Heterogeneous Specification and the Heterogeneous Tool Set

Habilitation thesis

Till Mossakowski

February 9th, 2005

Abstract

Formal specification of software systems has become more and more important, especially in safety-critical areas where one cannot take the risk of malfunction. CASL, the Common Algebraic Specification Language, is a standard for axiomatic specification of conventional software; and several extensions of CASL deal with temporal, reactive, higher-order etc. aspects. That is, we are faced with a multitude of specification languages and underlying logics.

In this work, we argue that for the specification of large software systems, heterogeneous multi-logic specifications are needed, since complex problems have different aspects that are best specified in different logics. A combination of all the used logics would become too complex in many cases. Moreover, using heterogeneous specifications, different approaches being developed at different sites can be related, i.e. there is a formal interoperability among languages and tools. In many cases, specialized languages and tools often have their strengths in particular aspects. Using heterogeneous specification, these strengths can be combined with comparably small effort.

The specification language CASL is an expressive specification language. At the level of basic specifications, it provides first-order logic with induction, powerful datatype constructs, subsorting and partial functions. CASL also provides constructs for structuring specifications-in-the-large. Several extensions of CASL (concurrent, modal-temporal, coalgebraic and higher-order) are formalized as so-called institutions, which means that CASL's structuring constructs can also be used for these extensions.

We extend CASL's powerful logic-independent structuring constructs to heterogeneous specification, obtaining the specification language Heterogeneous CASL (HETCASL). HETCASL allows mixing specifications written in different logics (using translations between the logics). It extends CASL only at the level of structuring constructs, by adding constructs for choosing the logic and translating specifications among logics. HETCASL is needed when combining specifications written in CASL with specifications written in its sublanguages and extensions. HETCASL also allows the integration of logics that are completely different from the CASL logic.

Heterogeneous specification in HETCASL is based on an arbitrary but fixed graph of logics (formalized as institutions) and logic translations (formalized as various kinds of institution morphisms). We provide an initial logic graph covering a range of different specification paradigms, and then study heterogeneous specification in general.

In order to obtain a semantic foundation for heterogeneous specification, we extend Diaconescu's morphism-based Grothendieck institutions to the case of comorphisms. This is not just a dualization, because we obtain more general results, especially concerning amalgamation properties. We also introduce a proof calculus for structured heterogeneous specifications and study its soundness and completeness (where amalgamation properties play a rôle for obtaining the latter).

Last but not least, we show how this theory can be brought into practice. The Heterogeneous Tool Set (HETS). HETS provides an abstract interface for logics and provides a parser, static analysis and proof engine for heterogeneous CASL— based on corresponding tools for the logics involved in the given logic graph.

Acknowledgments

This work is a natural extension of the author's work within the Common Framework Initiative (CoFI). All the members of this initiative deserve my thanks for having created a project with a fruitful cooperation among different groups. From the long list of contributors to CoFI I will name here only Michel Bidoit, Peter D. Mosses, Don Sannella and Andrzej Tarlecki, with whom it was a great pleasure to cooperate during the production the CoFI books.

More specifically, this work grew out of two cooperations: with Andrzej Tarlecki on the semantic aspects and with Serge Autexier and Dieter Hutter on the proof theoretic aspects of structured specifications. All these three deserve my thanks. Further thanks go to Andrzej Tarlecki, as well as to Joseph Goguen, Răzvan Diaconescu, José Meseguer and Grigore Roşu, for very fruitful discussions about institutions and their (co)morphisms.

Concerning the initial logic graph, which of course is crucial in order to show applicability of the abstract theory, I wish to thank the participants of CoFI for the fruitful cooperation concerning the design of CASL, and in particular Maura Cerioli, Anne Haxthausen and Bernd Krieg-Brückner for cooperation on the design of the subsorted CASL institution. Moreover, I wish to thank Lutz Schröder for the cooperation on HasCasl, Horst Reichel, Lutz Schröder, Markus Roggenbach and Daniel Hausmann for cooperation on CoCasl, Markus Roggenbach for cooperation on Csp-Casl, and Stefano Borgo and Claudio Masolo for valuable hints for ModalCasl. Lutz Schröder deserves special thanks for the numerous interesting discussions we had (and have), eased by the fact that we are roommates – the whiteboard in our room is always full of mysterious symbols (some of them being interpretable in arbitrary institutions)

Bernd Krieg-Brückner deserves my thanks for creating an excellent research environment, and for numerous discussions about design and tool issues, also concerning the design of the heterogeneous HetCasl.

The heterogeneous tool set Hets, which shows that the theory outlined in this work can also be brought to practice, would not have possible without cooperation with Christian Maeder and Klaus Lüttich. Besides the author, the following people have been involved in the implementation of Hets: Katja Abu-Dib, Carsten Fischer, Jorina Freya Gerken, Sonja Gröning, Wiebke Herding, Heng Jiang, Tina Krausser, Martin Kühl, Mingyi Liu, Klaus Lüttich, Christian Maeder, Maciek Makowski, Daniel Pratsch, Felix Reckers, Markus Roggenbach, Pascal Schmidt and Paolo Torrini. A precursor of Hets, namely Cats, has been developed with help of Mark van den Brand, Bartek Klin, Kolyang, Pascal Schmidt and Frederic Voisin. I also want to thank Agnes Arnould, Thibaud Brunet, Pascale LeGalle, Kathrin Hoffmann, Katiane Lopes, Klaus Lüttich, Christian Maeder, Stefan Merz, Maria Martins Moreira, Christophe Ringeissen, Markus Roggenbach, Dmitri Schamschurko, Lutz Schröder, Konstantin Tchekine and Stefan Wölfl for giving feedback about Cats, HOL-CASL and Hets. Finally, special thanks to Christoph Lüth and George Russell for help with connecting Hets to their UniForM workbench.

Lutz Schröder and Andrzej Tarlecki proofread part of this work. Erwin R. Catesbeiana helped with the remaining inconsistencies.

This work has been supported by the project MULTIPLE of the *Deutsche Forschungsgemeinschaft* under Grants KR 1191/5-1 and KR 1191/5-2, and by the CoFI Working Group (ESPRIT WG 29432).

Contents

1 Introduction and Motivation **7**
1.1 Examples of Sub-/Superlanguage and Wide-Spectrum Specifications 10
1.2 Example of a Viewpoint Specification: Temporal Properties of Reactive Systems . . 10
1.3 Example of a Specification Refinement 16
1.4 Overview of the Thesis 18

2 Institutions and Logics **20**
2.1 Institutions 20
2.2 Logical Consequence and Theories 22
2.3 Amalgamation and Craig Interpolation 22
2.4 Entailment Systems and Logics 24
2.5 Institution Morphisms 24
2.6 Institution Comorphisms 26
2.7 Simple Theoroidal (Co)Morphisms 27
2.8 Intersections of Subinstitutions 28
2.9 Adjointness Between Morphisms and Comorphisms 28
2.10 A Taxonomy of (Co)Morphisms 29
2.11 Properties of Institution Comorphisms 31
2.12 Institution (Co)Morphism Modifications 34
2.13 Institutions as Functors 36
2.14 Colimits in Hom-Categories 37
2.15 Polymorphism in an Arbitrary Institution 39
2.15.1 Failures of the Satisfaction Condition 39
2.15.2 Generic Polymorphism 41
2.15.3 A Generic Institutionalization 42
2.15.4 Semantic Consequence for Generic Polymorphism 44
2.15.5 Model-Theoretic Conservativity 45
2.16 Bibliographical Notes 46

3 Some Institutions for Specification of Software Systems **47**
3.1 CASL 47
3.1.1 Partial First-Order Logic 48
3.1.2 Subsorted Partial First-Order Logic 53
3.1.3 CASL Language Constructs 55
3.1.4 Proof Calculus 58
3.1.5 Checking Conservativity in CASL 58
3.1.6 Colimits of CASL signatures 58
3.1.7 Amalgamation in CASL 62
3.1.8 Craig Interpolation in CASL 63
3.1.9 Subinstitutions of CASL 64
3.1.10 Second-Order Logic 68
3.2 Modal CASL 68

3.2.1 Signatures . . . 68
3.2.2 Models . . . 70
3.2.3 Sentences . . . 70
3.2.4 Satisfaction . . . 71
3.2.5 Sublanguages of ModalCasl . . . 72
3.2.6 Amalgamation . . . 73
3.3 CoCasl . . . 73
3.3.1 Generation and cogeneration constraints . . . 76
3.3.2 Free types and cofree cotypes . . . 77
3.3.3 Modal logic . . . 80
3.3.4 The CoCasl Institution . . . 82
3.3.5 Amalgamation . . . 84
3.4 HasCasl . . . 84
3.4.1 The partial λ-calculus . . . 84
3.4.2 Product types . . . 88
3.4.3 Signatures . . . 88
3.4.4 Models . . . 94
3.4.5 Translation of HasCasl signatures into partial λ-theories . . . 94
3.4.6 Sentences . . . 95
3.4.7 Satisfaction . . . 96
3.4.8 The Internal Logic . . . 96
3.4.9 Proof Calculus . . . 98
3.4.10 Amalgamation in HasCasl . . . 98
3.4.11 HasCasl Language Constructs . . . 99
3.4.12 Functional Programs: Haskell . . . 100
3.5 Csp-Casl . . . 101
3.6 Bibliographical Notes . . . 103
3.6.1 Casl . . . 103
3.6.2 ModalCasl . . . 103
3.6.3 CoCasl . . . 104
3.6.4 HasCasl . . . 105
3.6.5 CspCASL . . . 105

4 An Initial Logic Graph **106**
4.1 Comorphisms Among Subinstitutions of Casl . . . 106
4.1.1 The First-Order Level . . . 106
4.1.2 The Positive Conditional Level . . . 114
4.2 Relating modal logics and Casl . . . 118
4.2.1 The standard translation . . . 119
4.2.2 Indexed Propositional Modal Logic . . . 120
4.3 Relating Casl and CoCasl . . . 120
4.4 Relating Casl into HasCasl . . . 121
4.4.1 Liberality . . . 121
4.4.2 Sort generation constraints . . . 122
4.5 Csp-Casl . . . 122
4.6 Bibliographical Notes . . . 124

5 Structured Specification and Development Graphs **125**
5.1 Specifications Over an Arbitrary Institution . . . 125
5.2 Borrowing . . . 126
5.2.1 Borrowing For Structured Specifications . . . 127
5.3 A Proof Calculus for Structured Specifications . . . 129
5.4 Development Graphs . . . 131

5.5 Verification Semantics for Structured Specifications . . . 136
5.6 Proof Rules for Development Graphs . . . 137
5.6.1 Faithful Extension Rule . . . 137
5.6.2 Hiding Decomposition Rules . . . 138
5.6.3 Conservativity rules . . . 141
5.6.4 Simple Structural Rules . . . 144
5.6.5 Soundness and Completeness . . . 145
5.7 A Sample Derivation in the Development Graph Calculus . . . 147
5.8 Bibliographical Notes . . . 148

6 Foundations of Heterogeneous Specification 149
6.1 Comorphism-Based Grothendieck Institutions . . . 149
6.2 Amalgamation and Exactness . . . 152
6.3 Grothendieck Logics and Heterogeneous Borrowing . . . 160
6.4 Heterogeneous Proofs . . . 161
6.5 A Sample Heterogeneous Proof . . . 165
6.6 Heterogeneous Bridges . . . 168
6.7 Morphism-Based Grothendieck Institutions . . . 171
6.8 The Bi-Grothendieck Institution . . . 172
6.9 Inducibility . . . 172
6.10 Spans of Comorphisms . . . 175
6.11 The Heterogeneous Verification Semantics . . . 179
6.12 Representation Maps . . . 182
6.13 Bibliographical Notes . . . 185

7 The Heterogeneous Tool Set (HETS) 186
7.1 Genericity Versus Heterogeneity . . . 189
7.2 The Type Class Logic . . . 189
7.3 Implementing the Grothendieck Logic . . . 191
7.4 Heterogeneous Parsing . . . 194
7.5 Heterogeneous Static Analysis . . . 194
7.6 Heterogeneous Proofs . . . 198
7.7 Overview of HETS module structure . . . 198

8 Conclusion 200
8.1 Future Work . . . 201
8.1.1 Management Of Change . . . 201
8.1.2 Reduction Strategies . . . 202
8.1.3 Flattening out Heterogeneity . . . 202
8.1.4 Interface for Theorem Provers . . . 202
8.1.5 Heterogeneous Architectural Specifications and Heterogeneous Refinement . . 203

A Heterogeneous CASL (HETCASL) Language Summary 215
A.1 Heterogeneous Concepts . . . 215
A.1.1 Institutions . . . 215
A.1.2 Institution Morphisms and Comorphisms . . . 216
A.1.3 Logic Graphs . . . 217
A.2 Heterogeneous Constructs . . . 218
A.2.1 The Current Logic . . . 218
A.2.2 Heterogeneous Structured Specifications . . . 219
A.2.3 Translations . . . 222
A.2.4 Fitting Arguments . . . 222
A.2.5 Heterogeneous Architectural Specifications . . . 222
A.2.6 Heterogeneous Specification Libraries . . . 222

A.3 Abstract Syntax . . . 223
A.3.1 Structured Specifications . . . 223
A.3.2 Specification Libraries . . . 224
A.4 Abbreviated Abstract Syntax . . . 224
A.4.1 Structured Specifications . . . 224
A.4.2 Specification Libraries . . . 225
A.5 Concrete Syntax . . . 225
A.5.1 Context-Free Syntax . . . 225
A.5.2 Structured Specifications . . . 226
A.5.3 Specification Libraries . . . 227
A.5.4 Lexical Syntax . . . 227

B Selected Code of the Heterogeneous Tool Set HETS **228**
B.1 Haskell Code of Type Class Logic . . . 228
B.2 Haskell Code of Type Class Comorphism . . . 235
B.3 Haskell Code of Grothendieck Logic . . . 239
B.4 Haskell Data Structure for Heterogeneous Development Graphs . . . 253

C Free and Cofree Specifications **256**
C.1 Free extensions and liberality . . . 256
C.2 Borrowing For Structured Specifications (Including **free**) . . . 259
C.3 Preservation of Freeness . . . 261
C.4 Rules for Free Theorem Links in Development Graphs . . . 263
C.5 Cofree specifications . . . 266

Chapter 1

Introduction and Motivation

> "Specification of a complex program necessarily involves presentation, often separate presentation, of many aspects of its behaviour; practically useful methodologies must support this. A successful example here is UML, where a good dozen of various kinds of diagrams are used. Diagrams of each kind present a different program view, and in fact (leaving aside any doubts about the formal underpinnings) offer a formalism to deal with one particular kind of program properties. None of these individual views captures all the aspects of the program in question, and only considering them together may lead to an adequate overall view." [Tar04]

> "...it is at present extremely difficult to integrate in a rigorous way different formal descriptions, and to reason across such descriptions. This situation is very unsatisfactory, and presents one of the biggest obstacles to the use for formal methods in software engineering because, given the complexity of large software systems, it is a fact of life that no single perspective, no single formalization of level of abstraction suffices to represent a system and reason about its behaviour." [Mes98b]

> "There is no λ-calculus of concurrency." [Abr03]

> "As can be seen, a plethora of formalisms for the verification of programs, and, in particular, for the verification of concurrent programs has been proposed. Up to now, their relationship is almost clear and for many different formalisms we already know if translations between them exist and how to translate them efficiently. ...the most important classical formalisms, namely μ-calculus, ω-automata, temporal logics, and predicate logic are considered and their relationship is outlined in detail. ...*there are good reasons to consider all the mentioned formalisms, and to use whichever one best suits the problem.*" [Sch04] (italics in the original)

Formal specification of software and hardware systems has become more and more important, especially in safety-critical areas where one cannot take the risk of malfunction. There has been a proliferation of logics covering the different aspects of such systems: datatypes, temporal and reactive behaviour, higher-order functions etc. That is, we are faced with a multitude of specification languages and underlying logics.

Abramsky [Abr03] has posed the question: Will there be a single unified theory of computer science, as there was in physics? Even in physics (with a much more well-defined object of research) the attempts to take the theories of the four forces to and combine them into a Grand unified theory has not been successful so far. The situation in computer science is much more scattered. This has been called the "next 700" syndrome [Lan66, Pau90]. In the area of formal specification, we have

- logics for specification of datatypes,
- process calculi and logics for the description of concurrent and reactive behaviour,

- logics for specifying security requirements and policies,
- logics for spatial reasoning,
- description logics for knowledge bases in artifical intelligence and for the semantic web,
- logics capturing the control of name spaces and administrative domains (e.g. the ambient calculus), etc.

Indeed, at present, it is not imaginable that a combination of all these (and other) logics would be feasible or even desirable — even if it existed, the combined formalism would lack manageability, if not become inconsistent. Often, even if a combined logic exists, for efficiency reasons, it is desirable to single out sublogics and study translations between these (cf. e.g. [Sch04]).

This means that for the specification of large software systems, heterogeneous multi-logic specifications are needed, since complex problems have different aspects that are best specified in different logics. Moreover, heterogeneous specifications additionally have the benefit that different approaches being developed at different sites can be related, i.e. there is a formal interoperability among languages and tools. In many cases, specialized languages and tools often have their strengths in particular aspects. Using heterogeneous specification, these strengths can be combined with comparably small effort.

In the literature, several approaches to heterogeneous specification have been developed [BCL96, CBL99, Dia02, Mos02b, Tar00, AC94, Dia98, Bor99, Dia02, Dia]. The most prominent approach is CafeOBJ with its cube of eight logics and twelve projections (formalized as *institution morphisms*) among them [DF96], having a semantics based on the notion of Grothendieck institution [Dia02]. However, not only projections between logics, but also logic encodings (formalized as so called *comorphisms*) are relevant to heterogeneous specification [Tar00, Mos02b]. Moreover, besides these model theoretic approaches, also the need of integrating different proof calculi via "bridges" has been stressed [CBL99]. The goal of the present work is to extend the Grothendieck institution approach to cover these aspects.

But let us stay away from the more technical side for a moment and take a more methodological view. There are a number of situations when heterogeneity arises:

- Specification involving different languages due to different *skills and customs* of the different specifiers involved.
- *Sub- and superlanguages*: e.g. first-order specifications with *some* higher-order part (e.g. real numbers).
- *Viewpoint specifications* [BBD$^+$00] combine specifications expressing different viewpoints on a common system, e.g. data types, process algebra and temporal logic.
- *Wide-spectrum* specifications involving specifications and programs, hence avoiding the need of special wide-spectrum languages.
- Specifications involving *different modalities* (e.g. deontic and temporal modalities).
- Specifications involving different input languages for *tools*.
- Specifications involving switches between "black box" and "glass box" views of a system.
- Combination of *formal ontologies* that are axiomatized in different logics.

Let us now have a closer look to some of these situations, and discuss some examples. For an introduction to the syntax of these examples, see the CASL user manual [BM04] and reference manual [CoF04], as well as Chaps. 3 and 5.

logic CASL
spec BOOL = **free type** $Bool ::= True \mid False$
spec LIST[**sort** $Elem$] =
free type $List[Elem] ::= nil \mid cons(Elem;\ List[Elem])$
logic HASCASL
spec FILTER = BOOL **and** LIST[**sort** $Elem$]
then
op $filter : (Elem \to Bool) \to List[Elem] \to List[Elem]$
$var\ p : Elem \to Bool;\ x : Elem;\ l : List[Elem]$
. $filter\ p\ [] = []$
. $filter\ p\ (x :: l) = x :: filter\ p\ l\ if\ p\ x = True$
. $filter\ p\ (x :: l) = filter\ p\ l\ if\ p\ x = False$

Figure 1.1: Specification of lists and a filter function in CASL and HASCASL.

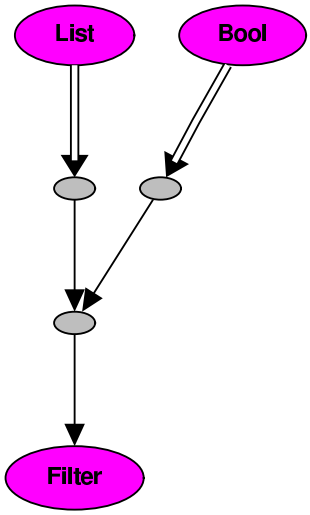


Figure 1.2: Development graph for specification of filter in CASL and HASCASL.

logic *Haskell*
spec FILTERIMP = PRELUDE
then {

```
data Elem
filter :: (Elem -> Bool) -> [Elem] -> [Elem]
filter p [] = []
filter p (x:l) | p x == True = x:filter p l
               | p x == False = filter p l
```

}
end

view CORRECT : FILTER **to** FILTERIMP
end

Figure 1.3: Implementation of filter function in Haskell.

1.1 Examples of Sub-/Superlanguage and Wide-Spectrum Specifications

Among sub- and superlanguages, there is always a trade-off between ease to learn and tool-support on the one hand and expressivity and conciseness on the other hand. Hence, it very well makes sense to use heterogeneous specifications with parts written in different sublanguages, even if in principle the whole specification could be expressed in one and the same superlanguage.

There are basically two different kinds of heterogeneous sub-/superlanguage specifications. They differ in whether the main specification is written in the sub- or in the superlanguage.

One of these situations (namely, main specification written in the superlanguage) is illustrated in Fig. 1.1: the standard datatypes of Booleans and lists are specified in the specification language CASL (a first-order language that also supports the specification of inductive datatypes). This specification then is extended with some higher-order filter function that is specified in HASCASL (the higher-order extension of CASL). The lists and Booleans typically will come from a library of CASL specifications (with theorems proved about the specifications), such that it would make not much sense to use HASCASL variants of lists and Booleans here.

Fig. 1.2 shows the development graph for the specification in Fig. 1.1. Development graphs are a formalism for the management of proofs in heterogeneous specifications. Nodes corresponds to the basic building blocks of specifications, while the arrows (links) indicate their relations. Solid arrows denote an import of another specification. Double arrow indicate imports that are heterogeneous, i.e. such that the logic changes along the arrow (here, it changes from CASL to HASCASL). Finally, Fig. 1.3 shows an implementation of the filter function in Haskell, and a so-called view that states the correctness of the Haskell program with respect to the HASCASL specification.

A similar, but more complex, specification formalizes the process algebra CCS in COCASL (see Fig. 1.6, and the development graph in Fig. 1.7). Full details of this specification are explained in [MSRR].

An example for the other kind (i.e. main specification in the sublanguage) is given in Fig. 1.4. Here, the real numbers (which can be thoroughly specified only in higher-order logic) are imported from a HASCASL library, while a graphics package using the reals is otherwise specified in first-order CASL. In such situations, one typically wants (for the sake of readability, better tool support etc.) to stay in the sublanguage as long as possible, and involve the superlanguage only at certain specific points. The development graphs of this example are shown in Fig. 1.5. Indeed, since the specifications are distributed over several libraries, we show one development graph per library, with rectangle nodes denoting imports from other libraries. The second development graph shows the structure of the projection of the real numbers from HASCASL onto CASL. Actually, two heterogeneous arrows[1] jointly denote the logic projection. The reason for this is explained in Sect. 6.10 and 6.11.

1.2 Example of a Viewpoint Specification: Temporal Properties of Reactive Systems

"Three types of formal models are used:

1. Firstly, we require an *executable model* (written in LOTOS using an object-based style) which is useful for constructing an executable model for validation.
2. Secondly, we have a *logical model* (based on the B method) which is used to verify the state invariant properties of our system (statically).
3. Finally, we use TLA to provide semantics for a *static analysis of liveness and fairness properties*. No one model can treat each of these aspects, yet each of these

[1] One of them is a *hiding* arrow, indicated by a different colour. In Chap. 5, we will use an h to indicate hiding arrows. The projection corresponds to an institution morphism, and the verification semantics in Sect. 5.5 translates this into two heterogeneous links via comorphisms, one of which is a hiding link.

library HasCASL/Real

from Basic/RelationsAndOrders **get** RichTotalOrder

from Basic/Algebra_I **get** Field

logic Casl
spec OrderedField =
    Field
and
    RichTotalOrder
then
    vars a, b, c: *Elem*
    • $(a + c) < (b + c)$ *if* $a < b$
    • $(a * c) < (b * c)$ *if* $a < b \wedge c > 0$
end

logic HasCasl
spec Real =
    OrderedField **with** *Elem*$\mapsto$ *Real*
then
    ops $\_<\_$: *Pred* (*Real* * *Pred* (*Real*));
        $\_<\_$: *Pred* (*Pred* (*Real*) * *Real*);
        isBounded : *Pred* (*Pred* (*Real*)); *inf*, *sup* : *Pred* (*Real*) $\rightarrow ?$ *Real*
    $\forall$ r, s: *Real*; M: *Pred* (*Real*)
    • $M < r \Leftrightarrow (\forall\ s$: *Real* • $M(s) \Rightarrow s < r)$
    • $r < M \Leftrightarrow (\forall\ s$: *Real* • $M(s) \Rightarrow r < s)$
    • $inf(M) = r \Leftrightarrow r < M \wedge (\forall\ s$: *Real* • $s < M \Rightarrow s < r)$
    • $sup(M) = r \Leftrightarrow M < r \wedge (\forall\ s$: *Real* • $M < s \Rightarrow r < s)$
    • $isBounded(M) \Leftrightarrow (\exists\ ub, lb$: *Real* • $lb < M \wedge M < ub)$
    • $isBounded(M) \Rightarrow def\ inf(M) \wedge def\ sup(M)$ %(completeness)%
end

library HasCASL/Graphics

from HasCASL/Real **get** Real

logic Casl
spec Graphics =
    Real **hide logic** $\rightarrow$ Casl
then
    free type *Coordinate* ::= $C(x, y$:*Real*)
    sort *Screen*
    op *move* : *Coordinate* $\times$ *Coordinate* $\rightarrow$ *Screen*
end

Figure 1.4: Specification of real numbers and graphics in Casl and HasCasl.

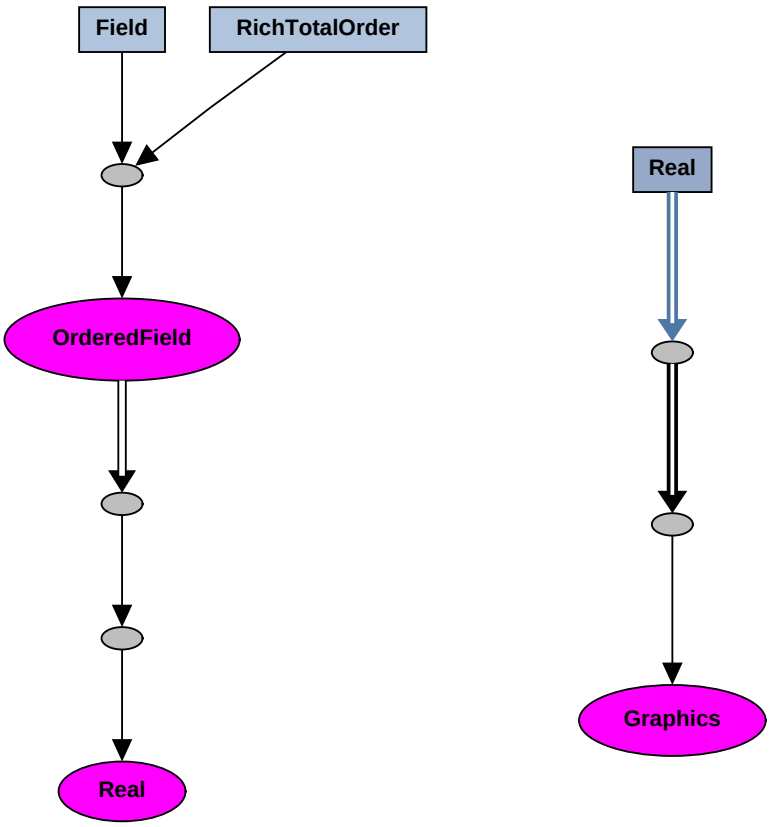


Figure 1.5: Development graphs for specification of real numbers and graphics in CASL and HAS-CASL.

library CoCASL/CCS_Automaton
...
logic CoCasl
spec FinalNonDeterministicAutomaton [Action] =
  cofree {Set [**sort** *State*] **then**
    cotype *State* ::= (*next* : *Act* → *Set*[*State*]) }
end
spec Zero = FinalNonDeterministicAutomaton [Action]
then op *zero* : *State*
  ∀ *a*: *Act* • *next*(*a*, *zero*) = {}
end
spec ActionPrefixing = FinalNonDeterministicAutomaton [Action]
then op _→_ : *Act* × *State* → *State*
  ∀ *x*, *y*: *Act*; *s*: *State* • *next*(*x*, *y* → *s*) = { *s* } *when* *x* = *y* *else* {}
end
spec Summation = Zero **and**
logic Casl : BinRelFun [**sort** *State* **op** _+_ : *State* × *State* → *State*]
then ∀ *a*: *Act*; *s1*, *s2*: *State*
  • *next*(*a*, *s1* + *s2*) = *pplus*(*zero* ∗ *next*(*a*, *s2*) *union* *next*(*a*, *s1*) ∗ *zero*)
end
spec Composition = FinAct **and** FinalNonDeterministicAutomaton [Action] **and**
logic Casl : BinRelFun [**sort** *State* **op** __||__ : *State* × *State* → *State*]
  with *pplus* ↦ *ppar*
and logic Casl : ExtSet [**sort** *State*]
and logic Casl : ExtSet [**sort** *Act*]
then ops _||_ : *State* × *State* → *State*; *h* : *State* × *State* × *Set*[*Label*] → *Set*[*State*]
  ∀*l* : *Label*; *s1*, *s2* : *State*; *set1*, *set2* : *Set*[*Label*]
  • *h*(*s1*, *s2*, {}) = {}
  • *h*(*s1*, *s2*, { *l* }) = *next*(*l*, *s1*) *intersection* *next*(*bar*(*l*), *s2*)
  • *h*(*s1*, *s2*, *set1* *union* *set2*) = *h*(*s1*, *s2*, *set1*) *union* *h*(*s1*, *s2*, *set2*)
  • *next*(*l*, *s1* || *s2*) = *ppar*(*next*(*l*, *s1*) ∗ *s2* *union* *s1* ∗ *next*(*l*, *s2*))
  • *next*(*tau*, *s1* || *s2*) = *ppar*(*next*(*tau*, *s1*) ∗ *s2* *union* *s1* ∗ *next*(*tau*, *s2*)) *union* *h*(*s1*, *s2*, *actions*)
end
spec Hiding = FinalNonDeterministicAutomaton [Action] **and** ActionRelabelling
then op _−_ : *State* × *Set*[*Label*] → *State*
  ∀ *l*: *Label*; *s*: *State*; *L*: *Set*[*Label*]
  • *next*(*l*, *s* − *L*) = {} *when* *l* *isIn* *L* = *True* *else* *next*(*l*, *s*)
  • *next*(*tau*, *s* − *L*) = *next*(*tau*, *s*)
end
spec Relabelling = FinalNonDeterministicAutomaton [Action] **and**
logic Casl : ExtSet [**sort** *Act*] **and** ActionRelabelling
then op *rel* : *State* × *Relabelling* → *State*
  ∀ *l*: *Label*; *s*: *State*; *f*: *Relabelling*
  • *next*(*l*, *rel*(*s*, *f*)) = *next*(*eval*(*f*, *l*), *s*)
  • *next*(*tau*, *rel*(*s*, *f*)) = *next*(*tau*, *s*)
end
spec CCS_Coalgebraic_Semantics = FinAct **and** Zero **and** ActionPrefixing
 and Summation **and** Composition **and** Hiding **and** Relabelling **and** CCS
end

Figure 1.6: Specification of CCS in Casl and CoCasl.

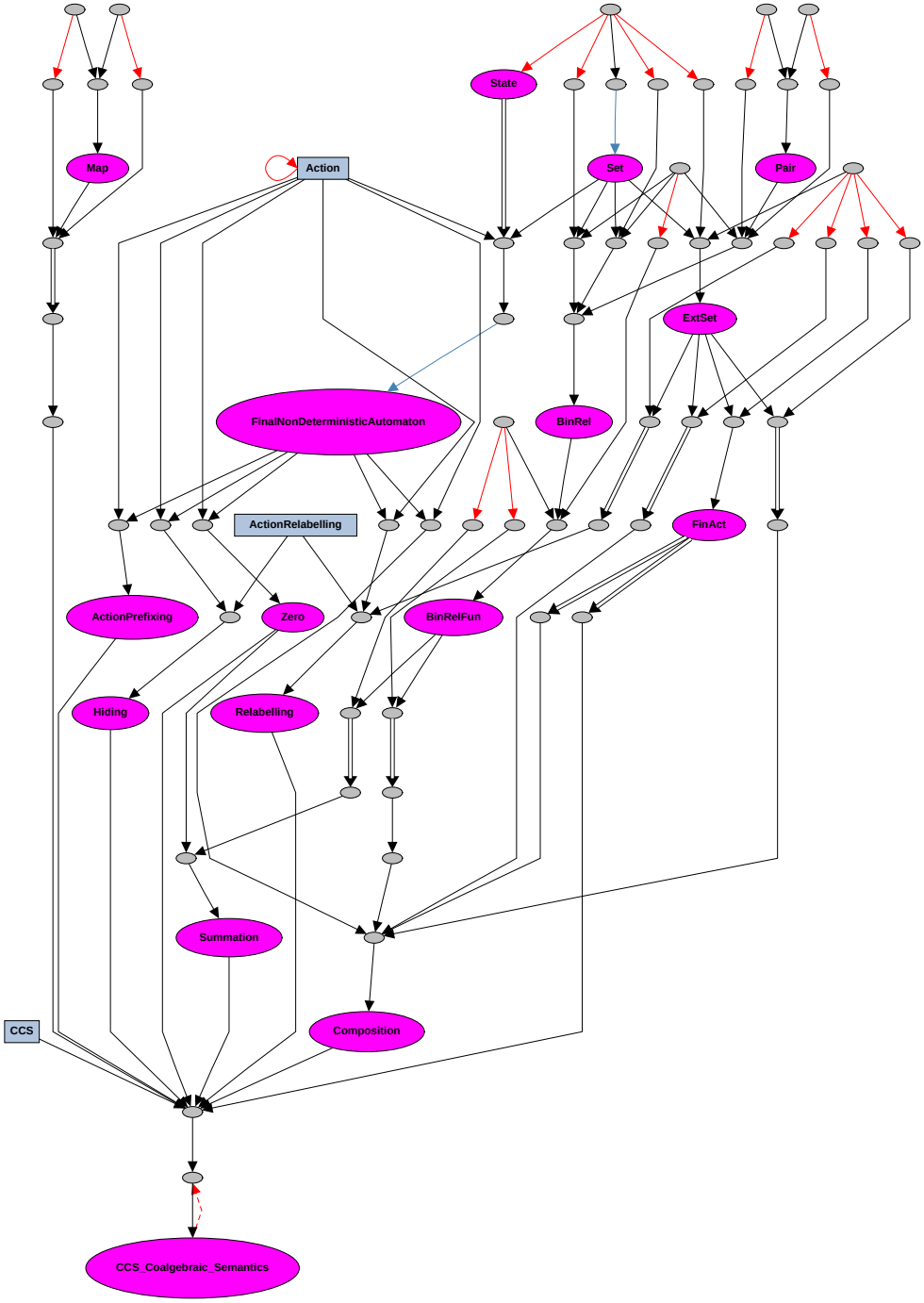


Figure 1.7: Development graph for specification of CCS in CASL and COCASL.

```
logic CSP-CASL
spec BUFFER =
    data LIST
    channels read, write : Elem
    process let Buf(l : List[Elem]) =
                    read?x → Buf(cons(x, nil))
                    □ if l = nil then STOP
                        else write!last(l) → Buf(rest(l))
            in Buf(nil)
    with logic → MODALCASL
    then %implies • A G F ∃x : Elem . ⟨write.x⟩ true
end
```

Figure 1.8: A specification of fair buffers in CASL, CSP-CASL and MODALCASL.

> aspects of the conceptualization are necessary in the formal development of features." [GMM97]

Here, we provide a similar scenario of heterogeneity arising in the specification of reactive systems: some equational or first-order logic is used to specify the data (here, lists over arbitrary elements), some process algebra (here, CSP) is used to describe the system (here, a buffer implemented as a list), and some temporal logic is used to state fairness or eventuality properties that go beyond the expressiveness of the process algebra (here, we express the fairness property that the buffer cannot read infinitely often without writing). A corresponding heterogeneous specification is given in Fig. 1.8, the corresponding development graph in Fig. 1.9. The dotted arrow in the development graph is a so-called *local theorem link* (indicated with a different colour), denoting a proof obligation that is caused by the **%implies** annotation.

Actually, one should add that the process Buf does not meet the fairness constraint, since it can read infinitely often without ever writing. However, a simplistic buffer such as

$$Copy = read?x \rightarrow write!x \rightarrow Copy$$

satisfies the fairness constraint, and so does a buffer using bounded lists.

We now briefly introduce the several languages involved; a more detailed definition has to wait until Chapt. 3.

CASL [CoF04] is based on an extension of *first-order logic* with partial functions and subsorted. Moreover, sort generation constraints allow expressing that sorts are term generated, which is needed for the specification of inductive datatypes like lists.

CSP-CASL [Rog] combines CASL with the process algebra CSP. CSP processes may involve data terms from CASL for their communications.

MODALCASL is an extension of CASL with first-order modal logic with Kripke and neighbourhood semantics. Also, the computation tree logic CTL^* [vL90] with its path quantifiers is supported. In the example, $A\ G\ F\ \varphi$ means: for all "execution paths" (through the Kripke models), it is always the case, that eventually ϕ holds.

Among these languages, we now introduce some translations; these are formally defined in Chap. 4:

- An inclusion translation from CASL to CSP-CASL, which is implicitly invoked with the **data** keyword (indicating the CASL data specification that is used for the subsequent CSP-CASL process specification).

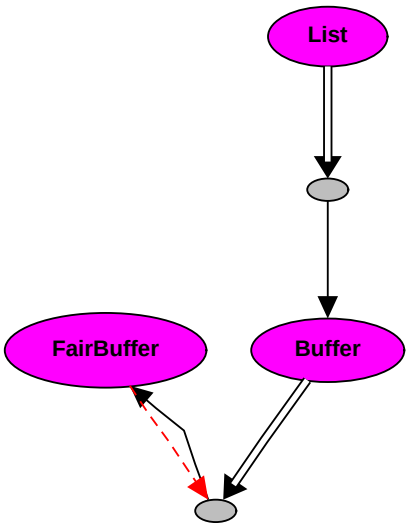


Figure 1.9: Development graph for fair buffer specification in CASL, CSP-CASL and MODALCASL

- A translation $Csp2Modal$: CSP-CASL $\longrightarrow$ MODALCASL, mapping the labelled transition system obtained by the CSP-CASL semantics to MODALCASL. This translation is invoked with the syntax **with logic** $\rightarrow$ MODALCASL.
- An inclusion translation CASL2$Modal$: CASL $\longrightarrow$ MODALCASL.

1.3 Example of a Specification Refinement

Consider the following specification of sets in CASL:

spec SET [**sort** *Elem*] =
  generated type $Set[Elem] ::= empty \mid insert(Elem; Set[Elem])$
  pred $\_is\_in\_ : Elem \times Set[Elem]$
  $\forall\ e, e' : Elem; S, S' : Set[Elem]$
  • $\neg\ e\ is\_in\ empty$
  • $e\ is\_in\ insert(e', S) \Leftrightarrow e = e' \vee e\ is\_in\ S$
  • $S = S' \Leftrightarrow (\forall\ x : Elem \bullet x\ is\_in\ S \Leftrightarrow x\ is\_in\ S')$ %(equal_sets)%
end

If we want to implement sets as ordered lists, we need to assume some total ordering:

spec TOTALORDER =
  sort *Elem*
  pred $\_\leq\_ : Elem \times Elem$
  $\forall\ x, y, z : Elem$
  • $x \leq x$ %(reflexive)%
  • $x \leq z\ if\ x \leq y \wedge y \leq z$ %(transitive)%
  • $x = y\ if\ x \leq y \wedge y \leq x$ %(antisymmetric)%
  • $x \leq y \vee y \leq x$ %(dichotomous)%

end

spec LIST [TOTALORDER] =
  free type $List[Elem] ::= [] \mid \_::\_(head{:}?Elem;\ tail{:}?List[Elem])$
then
  preds $\_is\_in\_ : Elem \times List[Elem]$;
    $is\_ordered : List[Elem]$
  vars x, y: $Elem$; $L, L1, L2$: $List[Elem]$
  • $\neg\ x\ is\_in\ []$
  • $x\ is\_in\ (y :: L) \Leftrightarrow x = y \vee y\ is\_in\ L$
  • $is\_ordered([])$
  • $is\_ordered(x :: [])$
  • $is\_ordered(x :: y :: L) \Leftrightarrow x \leq y \wedge is\_ordered(y :: L)$
end

spec SORTEDLIST [TOTALORDER] =
  LIST [TOTALORDER]
then
  sort $SortedList[Elem] = \{l{:}\ List[Elem] \bullet is\_ordered(l)\}$
  ops $[] : SortedList[Elem]$;
    $insert : Elem \times SortedList[Elem] \to SortedList[Elem]$;
    $head : SortedList[Elem] \to? Elem$;
    $tail : SortedList[Elem] \to? SortedList[Elem]$
  pred $\_is\_in\_ : Elem \times SortedList[Elem]$
  $\forall\ x, y$: $Elem$; L: $SortedList[Elem]$
  • $insert(x, []) = x :: []$
  • $insert(x, L) = x :: L\ when\ x \leq head(L)$
    $else\ head(L) :: insert(x, tail(L))\ as\ SortedList[Elem]$
  hide $List[Elem]$
end

In the refinement language of [MSA05], we can now express that ordered lists are an implementation of sets:

spec NAT = ...

refinement R =
  SET [NAT]
  behaviourally refined via $Set[Nat] \mapsto SortedList[Nat]$, $empty \mapsto []$
  to SORTEDLIST [NAT]
end

The refinement expresses that each model of the target specification is also (when sufficiently translated) a model of the source specification [MSA05]. The keyword **behaviourally** [MA] here indicates that the refinement is only up to observable behaviour. I.e. the specification SET[NAT] need not be satisfied literally, but only up to the behaviour observable by the SET[NAT] -operations.

Although all these specifications are written in CASL, the example involves some kind of heterogeneity. This is because in the framework of development graphs, behavioural refinement can be best expressed by using some behavioural quotient construction (in the example, this construction basically throws out all lists that are not sorted, and identifies any two sorted lists that contain the same elements, i.e. duplicates are ignored). This behavioural quotient can be seen as a construction on the CASL logic. Hence, the framework of heterogeneous specifications is well-suited to treat such examples. The corresponding development graph is shown in Fig. 1.10. The heterogeneous links

here are caused by the translation of CASL into itself.[2] The link between the lower two nodes is a *global theorem link* (indicated with a different colour) denoting a proof obligation corresponding to the refinement.[3]

1.4 Overview of the Thesis

The logics of the various specifications formalisms are formalized as so-called *institutions*, and related with various kinds of institutions morphisms and comorphisms. These notions are the subject of Chap. 2, which also discusses some properties of institutions that important for structured and heterogeneous specification, such as amalgamation and Craig interpolation.

Chap. 3 then presents a number of institutions in detail. Most of them are extensions of the Common algebraic specification language CASL. CASL is an expressive language based on first-order logic with induction, that provides powerful constructs for the specification of datatypes. Partial functions and subsorts are supported as well. The modal logic extension MODALCASL allows for specification of labelled transition systems (=Kripke models) in modal, temporal and dynamic logic. COCASL is a co-algebraic extension of CASL suitable for the specification of process types. HASCASL is a higher-order extension of CASL that is geared towards the development of functional programs, particular ones written in the programming language Haskell. Note that it is also possible formalize Haskell as an institution. Finally, CSP-CASL is an extension of CASL with CSP processes terms that describe concurrent processes (with CASL data elements being communicated over the CSP channels).

The relations and translations among these different formalisms are studies in Chap. 4, where an initial logic graph of CASL and its extensions is built. This logic graph will be a reference example for the theory developed in later chapters. We again stress that this theory is not limited at all to CASL and its extensions; one can choose to work with completely different logic graphs.

Chap. 5 covers structured specifications over an arbitrary but fixed institution. Indeed, the constructs for structuring large specifications can be defined and studied completely independent of the underlying institution. Actually, we present two institution independent structuring formalisms: term-like structured specifications and development graphs. For both structuring formalisms, proof calculi are given, and their soundness and completeness is discussed.

Chap. 6 contains the core of the theory of heterogeneous specification. It starts with Diaconescu's notion of Grothendieck institution, which is a kind of flattening of a graph of institutions and institution morphisms. We here dualize the construction to comorphisms. Amalgamation properties for this Grothendieck institution, which are studied subsequently, play the same central role as they play for individual institutions. They build the necessary prerequisite for heterogeneous proof systems, and the completeness proof for the proof system for heterogeneous development graphs. While the latter are just ordinary development graphs over a Grothendieck institutions, there are some aspects that need to be covered specifically for the heterogeneous situation. Finally, the approach is extended to deal with other types of institution translation than just with comorphisms.

All this theory also has been implemented: in the Heterogeneous Tool Set, which is the subject of Chap. 7. We first explain the structure of HETS by explaining some toy version for some toy heterogeneous language, and then give a short overview of the full system.

Chap. 8 contains conclusions and discusses related and future work.

Appendix A contains a detailed summary of the heterogeneous language HETCASL, which is the input language for HETS, and which also has been used for the examples above.

Appendix B shows some central modules of HETS in full detail. Actually, they can be read as a formalization of the theory in Haskell.

Appendix C covers freeness and cofreeness structuring constructs.

[2] Actually, the behavioural quotient operation corresponds to an institution semi-comorphism (see Chap. 2), and the verification semantics in Sect. 5.5 translates this into two heterogeneous links via comorphisms, one of which is a hiding link. The latter is indicated with a different colour.

[3] There are further theorem links corresponding to instantiations of parameterized specifications.

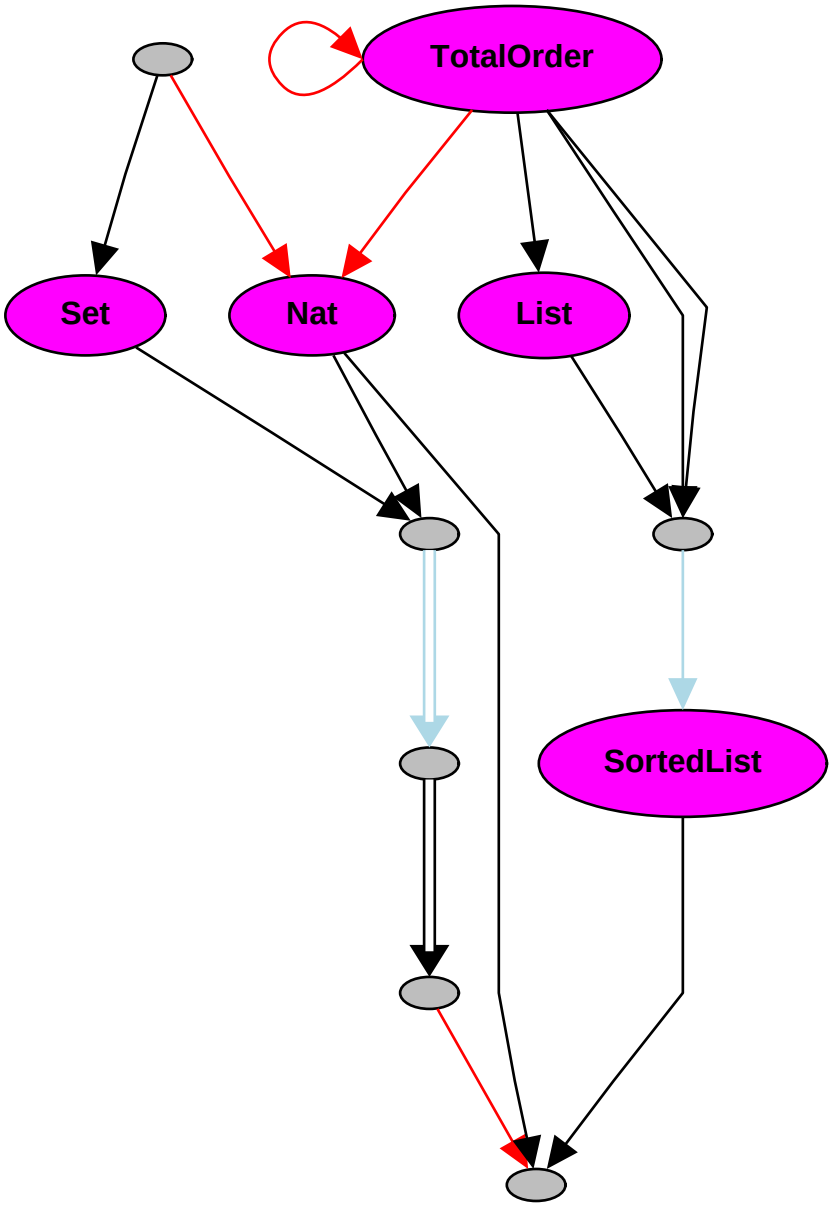


Figure 1.10: Development graph for a refinement

Chapter 2

Institutions and Logics

> "There is a population explosion among the logical systems used in computer science. Examples include first order logic, equational logic, Horn clause logic, higher order logic, infinitary logic, dynamic logic, intuitionistic logic, order-sorted logic, and temporal logic; moreover, there is a tendency for each theorem prover to have its own idiosyncratic logical system. We introduce the concept of *institution* to formalize the informal notion of 'logical system'." [GB92]

Institutions are the central abstract notion that is the basis for a theory of structured specification and proving independent of the underlying logical system. Naturally, this notion is also the basis for heterogeneous specification. While institutions capture model theory, entailment systems are a related abstract notion capturing proof theory. Finally, an institution equipped with an entailment is called a *logic*.

Many different logics, including first-order [GB92], higher-order [Bor99], polymorphic [NP86, SML05], modal [Cîr02, SSCM00, Dia], temporal [FC96], process [FC96], behavioural [BHa], and object-oriented [SCS94, GD94, LF97, SS93, Ala02] logics have been shown to be institutions. Recently, there have even institutions for XML [Ala02] and databases [Gog] have been examined.

2.1 Institutions

A specification formalism is usually based on some notion of signature, model, sentence and satisfaction. These are the usual ingredients of Barwise's abstract model theory [Bar74]. Contrary to Barwise's notions, *institutions* of Goguen and Burstall [GB92] do not assume that signatures are algebraic signatures and thus cover a much larger variety of logics. Indeed, the theory of institutions assumes nothing about signatures except that they form a class and that there are *signature morphisms*, which can be composed in some way. This amounts to stating that signatures form a *category*.

There is also nothing special assumed about the form of the *sentences* and *models*. Given a signature Σ, the Σ-sentences form just a set, while the Σ-models form a category (taking into account that there may be *model morphisms*).

Signature morphisms lead to *translations* of sentences and of models (thus, the assignments of sentences and of models to signatures are functors). There is a contravariance between the sentence and the model translation: sentences are translated *along* signature morphisms, while models are translated *against* signature morphisms.

Informally, this can be motivated as follows. Forget for a moment the above generality and think of signatures as of sets of certain symbols. Think of sentences over a signature Σ as derivation trees over some grammar, decorated at the nodes with the symbols from Σ. Then sentence translation along a signature morphism $\sigma: \Sigma \longrightarrow \Sigma'$ keeps the structure of the derivation tree, but replaces the

symbols decorating the nodes, using σ. This explains why sentences are translated *along* signature morphisms.

Concerning models over a signature: they have to interpret the symbols from the signature somehow. Thus, a Σ-model can be seen as a map M going from the symbols of Σ to some semantical domain. Now given a Σ'-model M' and a signature morphism $\sigma: \Sigma \longrightarrow \Sigma'$, by composing the interpretation map M' with σ we get a new interpretation map, let us call it $M'|_\sigma$, which is a Σ-model! ($M'|_\sigma$ is also called the *σ-reduct* of M'.) This explains why models are translated *against* signature morphisms.

Of course, these explanations just have motivating purpose: there can be institutions with a completely different view of signatures, models and sentences. However, they shed some light on how many typical institutions work.[1]

Finally, institutions have a *satisfaction relation* between models and sentences, which has to be invariant under the simultaneous translation of sentences and models w.r.t. a given signature morphism.

This leads to the following formal definition. Let $\mathcal{CAT}$ be the category of categories and functors.[2]

Definition 2.1 An *institution* $I = (\mathbf{Sign}^I, \mathbf{Sen}^I, \mathbf{Mod}^I, \models^I)$ consists of

- a category $\mathbf{Sign}^I$ of *signatures*,
- a functor $\mathbf{Sen}^I: \mathbf{Sign}^I \longrightarrow \mathbf{Set}$ giving, for each signature Σ, the set of *sentences* $\mathbf{Sen}^I(\Sigma)$, and for each signature morphism $\sigma: \Sigma \longrightarrow \Sigma'$, the *sentence translation map* $\mathbf{Sen}^I(\sigma): \mathbf{Sen}^I(\Sigma) \longrightarrow \mathbf{Sen}^I(\Sigma')$, where often $\mathbf{Sen}^I(\sigma)(\varphi)$ is written as $\sigma(\varphi)$,
- a functor $\mathbf{Mod}^I: (\mathbf{Sign}^I)^{op} \longrightarrow \mathcal{CAT}$ giving, for each signature Σ, the category of *models* $\mathbf{Mod}^I(\Sigma)$, and for each signature morphism $\sigma: \Sigma \longrightarrow \Sigma'$, the *reduct functor* $\mathbf{Mod}^I(\sigma): \mathbf{Mod}^I(\Sigma') \longrightarrow \mathbf{Mod}^I(\Sigma)$, where often $\mathbf{Mod}^I(\sigma)(M')$ is written as $M'|_\sigma$,
- a satisfaction relation $\models^I_\Sigma \subseteq |\mathbf{Mod}^I(\Sigma)| \times \mathbf{Sen}^I(\Sigma)$ for each $\Sigma \in \mathbf{Sign}^I$,

such that for each $\sigma: \Sigma \longrightarrow \Sigma'$ in $\mathbf{Sign}^I$ the following *satisfaction condition* holds:

$$M' \models^I_{\Sigma'} \sigma(\varphi) \Leftrightarrow M'|_\sigma \models^I_\Sigma \varphi$$

for each $M' \in \mathbf{Mod}^I(\Sigma')$ and $\varphi \in \mathbf{Sen}^I(\Sigma)$. □

We will omit the index I when it is clear from the context.

We now informally present some examples. They will be (more) formally introduced in Sect. 3.1.

Example 2.2 The institution $Eq^=$ of equational logic. Signatures are many-sorted algebraic signatures consisting of a set of sorts and a set of function symbols (where each function symbol has a string of argument sorts and a result sort). Signature morphisms map sorts and function symbols in a compatible way. Models are just many-sorted algebras, i.e. each sort is interpreted as a carrier set, and each function symbol is interpreted as a function between the carrier sets specified by the argument and result sorts. Reducts are constructed as sketched above. Sentences are equations between many-sorted terms, and sentence translation means replacement of the translated symbols. Finally, satisfaction is the usual satisfaction of an equation in an algebra. □

Example 2.3 The institution $FOL^=$ of many-sorted first-order logic with equality. Signatures are many-sorted first-order signatures, i.e. many-sorted algebraic signatures enriched with predicate symbols. Models are many-sorted first-order structures. Sentences are first-order formulas, and again sentence translation means replacement of the translated symbols. Satisfaction is the usual satisfaction of a first-order sentence in a first-order structure. □

[1] Indeed, the above explanation has been formalized as so-called *parchments* [Mos96d].

[2] Strictly speaking, $\mathcal{CAT}$ is not a category but only a so-called quasicategory, which is a category that lives in a higher set-theoretic universe [HS73].

Example 2.4 The institution $PFOL^{=}$ of partial first-order logic with equality. Signatures are many-sorted first-order signatures enriched by partial function symbols. Models are many-sorted partial first-order structures. Sentences are first-order formulas containing existential equations, strong equations, definedness statements and predicate applications as atomic formulas. Satisfaction is defined using total valuations of variables, while valuation of terms is partial due to the existence of partial functions. An existential equation holds if both sides are defined and equal, whereas a strong equation also holds if both sides are undefined. A definedness statement holds if the term is defined. A predicate application holds if the terms contained in it are defined, and the corresponding tuple of values is in the interpretation of the predicate. This is extended to first-order formulas as usual. □

2.2 Logical Consequence and Theories

Within an arbitrary but fixed institution, we can easily define the usual notion of *logical consequence* or *semantical entailment*: Given a set of Σ-sentences Ψ and a Σ-sentence φ, we say

$$\Psi \models_\Sigma \varphi \text{ (φ follows from Ψ)}$$

iff for all Σ-models M, we have

$$M \models_\Sigma \Psi \text{ implies } M \models_\Sigma \varphi.$$

Here, $M \models_\Sigma \Psi$ means that $M \models_\Sigma \psi$ for each $\psi \in \Psi$.

In an arbitrary institution I, a *theory* is a pair $T = \langle \Sigma, \Psi \rangle$, where $\Sigma \in \mathbf{Sign}$ and $\Psi \subseteq \mathbf{Sen}(\Sigma)$ (we set $Sig(T) = \Sigma$ and $Ax(T) = \Psi$).[3] *Theory morphisms* $\sigma\colon \langle \Sigma, \Psi \rangle \longrightarrow \langle \Sigma', \Psi' \rangle$ are those signature morphisms $\sigma\colon \Sigma \longrightarrow \Sigma'$ for which $\Psi' \models_{\Sigma'} \sigma(\Psi)$, that is, axioms are mapped to logical consequences. By inheriting composition and identities from **Sign**, we obtain a category **Th** of theories. The category **Pres** of *presentations* (also called *flat specifications*) is just the full subcategory of theories having finite sets of axioms.

It is easy to extend **Sen** and **Mod** to start from **Th** by putting $\mathbf{Sen}(\langle \Sigma, \Psi \rangle) = \mathbf{Sen}(\Sigma)$ and letting $\mathbf{Mod}(\langle \Sigma, \Psi \rangle)$ be the full subcategory of $\mathbf{Mod}(\Sigma)$ induced by the class of those models M satisfying Ψ. In this way, we get the *institution of theories* $I^{th} = (\mathbf{Th}, \mathbf{Sen}, \mathbf{Mod}, \models)$ over I.

A theory morphism $\sigma\colon \langle \Sigma, \Psi \rangle \longrightarrow \langle \Sigma', \Psi' \rangle$ is *conservative*, if each $\langle \Sigma, \Psi \rangle$ has a σ-expansion to a $\langle \Sigma', \Psi' \rangle$-model; it is *monomorphic*, if each model has such an expansion that is unique up to isomorphism, and it is *definitional*, if each model has a unique such expansion.

We will also freely use other standard logical terminology when working within an arbitrary but fixed institution.

2.3 Amalgamation and Craig Interpolation

The amalgamation property (called 'exactness' in [DGS91]) is a major technical assumption in the study of specification semantics [ST88a] and is important in many respects. To give a few examples: it allows the computation of normal forms for specifications [BHK90, Bor00], and it is a prerequisite for good behaviour w.r.t. parameterization [EM90b] and conservative extensions [DGS91]. A Z-like state based language has been developed over an arbitrary institution with amalgamation [Bau98]. Here, we mainly will need amalgamation for using the proof system for development graphs with hiding (see Sect. 5.6). Craig interpolation plays a similar role for proof systems for structured specifications, see e.g. [Bor02] and Sect. 5.3.

In the sequel, fix an arbitrary institution $I = (\mathbf{Sign}, \mathbf{Sen}, \mathbf{Mod}, \models)$. We now recall the institution independent definitions of amalgamation and Craig interpolation properties that will become important for (proving in) structured specification (see Chap. 5).

[3] Note that the theories introduced here are *presentations* of theories. We follow here the terminology of Meseguer's general logics [Mes89a] instead of Goguen and Burstall's original definition [GB92]. In what follows, when we talk about a theory (Σ, Ψ) we shall mean a theory presentation.

Definition 2.5 The institution I has the *Craig interpolation property*, if for any pushout in **Sign**

$$\begin{array}{ccc} \Sigma & \xrightarrow{\sigma_1} & \Sigma_1 \\ \downarrow{\scriptstyle \sigma_2} & & \downarrow{\scriptstyle \theta_2} \\ \Sigma_2 & \xrightarrow{\theta_1} & \Sigma' \end{array}$$

any Σ_1-sentence φ_1 and any Σ_2-sentence φ_2 with

$$\theta_2(\varphi_1) \models \theta_1(\varphi_2),$$

there exists a Σ-sentence φ (called the *interpolant*) such that

$$\varphi_1 \models \sigma_1(\varphi) \text{ and } \sigma_2(\varphi) \models \varphi_2.$$

Definition 2.6 A cocone for a diagram in **Sign** is called *(weakly) amalgamable* if it is mapped to a (weak) limit under **Mod**. I (or **Mod**) admits *(finite) (weak) amalgamation* if (finite) colimit cocones are (weakly) amalgamable, i.e. if **Mod** maps (finite) colimits to (weak) limits. An important special case is pushouts: I is called *(weakly) semi-exact*, if it admits (weak) amalgamation for pushout diagrams.

Pushouts in the signature category are prominently used for instance in instantiations of parameterized specifications. (Recall also that finite limits can be constructed from pullbacks and terminal objects, so that finite amalgamation reduces to preservation of pullbacks and terminal objects — dually: pushouts and initial objects). Here, the (weak) amalgamation property requires that a pushout

$$\begin{array}{ccc} \Sigma & \longrightarrow & \Sigma_1 \\ \downarrow & & \downarrow \\ \Sigma_2 & \longrightarrow & \Sigma_R \end{array}$$

in **Sign** is mapped by **Mod** to a (weak) pullback

$$\begin{array}{ccc} \mathbf{Mod}(\Sigma) & \longleftarrow & \mathbf{Mod}(\Sigma_1) \\ \uparrow & & \uparrow \\ \mathbf{Mod}(\Sigma_2) & \longleftarrow & \mathbf{Mod}(\Sigma_R) \end{array}$$

of categories. Explicitly, this means that any pair $(M_1, M_2) \in \mathbf{Mod}(\Sigma_1) \times \mathbf{Mod}(\Sigma_2)$ that is *compatible* in the sense that M_1 and M_2 reduce to the same Σ-model can be *amalgamated* to a unique (or weakly amalgamated to a not necessarily unique) Σ_R-model M (i.e., there exists a (unique) $M \in \mathbf{Mod}(\Sigma_R)$ that reduces to M_1 and M_2, respectively), and similarly for model morphisms.

More generally, given a diagram $D\colon J \longrightarrow \mathbf{Sign}^I$, a family of models $(M_j)_{j\in|J|}$ is called *D-consistent* if $M_k|_{D(\delta)} = M_j$ for each $\delta\colon j \longrightarrow k \in J$. A cocone $(\Sigma, (\mu_j)_{j\in|J|})$ over the diagram in $D\colon J \longrightarrow \mathbf{Sign}^I$ is called *weakly amalgamable* if for each D-consistent family of models $(M_j)_{j\in|J|}$, there is a Σ-model M with $M|_{\mu_j} = M_j$ $(j \in |J|)$. If this model is unique, the cocone is called *amalgamable*.

Proposition 2.7 An institution admits (weak) amalgamation iff each colimiting cocone is (weakly) amalgamable.

The notion of weakly amalgamable cocone leads to further weakenings of the above notions. They will become important in Chapters 5 and 6.

Definition 2.8 Call an institution I *quasi-exact* if for each diagram $D\colon J \longrightarrow \mathbf{Sign}^I$, there is some weakly amalgamable cocone over D. *Quasi-semi-exactness* is the restriction of this notion to diagrams of shape $\bullet \longleftarrow \bullet \longrightarrow \bullet$.

2.4 Entailment Systems and Logics

Coming to proofs, a *logic* extends an institution with proof-theoretic entailment relations that are compatible with semantic entailment.

Definition 2.9 A *logic* $\mathcal{LOG} = (\mathbf{Sign}, \mathbf{Sen}, \mathbf{Mod}, \models, \vdash)$ is an institution $(\mathbf{Sign}, \mathbf{Sen}, \mathbf{Mod}, \models)$ equipped with an *entailment system* $\vdash$, that is, a relation $\vdash_\Sigma \subseteq \mathcal{P}(\mathbf{Sen}(\Sigma)) \times \mathbf{Sen}(\Sigma)$ for each $\Sigma \in |\mathbf{Sign}|$, such that the following properties are satisfied:

1. *reflexivity:* for any $\varphi \in \mathbf{Sen}(\Sigma)$, $\{\varphi\} \vdash_\Sigma \varphi$,
2. *monotonicity:* if $\Psi \vdash_\Sigma \varphi$ and $\Psi' \supseteq \Psi$ then $\Psi' \vdash_\Sigma \varphi$,
3. *transitivity:* if $\Psi \vdash_\Sigma \varphi_i$ for $i \in I$ and $\Psi \cup \{\varphi_i \mid i \in I\} \vdash_\Sigma \psi$, then $\Psi \vdash_\Sigma \psi$,
4. $\vdash$-*translation:* if $\Psi \vdash_\Sigma \varphi$, then for any $\sigma: \Sigma \longrightarrow \Sigma'$ in $\mathbf{Sign}$, $\sigma(\Psi) \vdash_{\Sigma'} \sigma(\varphi)$,
5. *soundness:* if $\Psi \vdash_\Sigma \varphi$ then $\Psi \models_\Sigma \varphi$.

A logic is *complete* if, in addition, $\Psi \models_\Sigma \varphi$ implies $\Psi \vdash_\Sigma \varphi$.

It is easy to obtain a complete logic from an institution by simply defining $\vdash$ as $\models$. Hence, $\vdash$ might appear to be redundant. However, the point is that $\vdash$ will typically be defined via a system of finitary *derivation rules*. This gives rise to a notion of *proof* that is absent when the institution is considered on its own, even if the relation that results coincides with semantic entailment which is defined in terms of the satisfaction relation.

Example 2.10 Many-sorted first-order logic with equality ($FOL^=$) can be easily seen to be a logic, using some standard entailment system. While $FOL^=$ is known to be complete, many-sorted second-order logic with equality ($SOL^=$) only is complete if we work with Henkin-models. The specification language CASL has an underlying institution $SubPCFOL^=$ (subsorted partial first-order logic with generation constraints and equality), which is usually translated to some other institution for doing proofs, hence, it is just an institution and not a logic in the first place.

Examples of logics that can be formalized in this sense are many-sorted equational logic, many-sorted first-order logic, higher-order logic, various lambda calculi, various modal, temporal, and object-oriented logics etc.

2.5 Institution Morphisms

Institution morphisms relate two given institutions. A typical situation is that an institution morphism expresses the fact that a "larger" institution *is built upon* a "smaller" institution by *projecting* the "larger" institution onto the "smaller" one.

An institution morphism from an institution I to an institution J consists of the following components:

- a translation of I-signatures to J-signatures. The idea is here that an I-signature Σ is *projected* to a J-signature $\Phi(\Sigma)$ by leaving out all features from I that have no counterpart in J. For example, if I has signatures with both partial and total function symbols, while J has just total function symbols, then the projection would just remove the partial function symbols from a signature.
- a translation of J-sentences to I-sentences. Note that we cannot expect to be able to translate Σ-sentences in I to $\Phi(\Sigma)$-sentences in J: if an Σ-sentence is thought to be a derivation tree decorated with symbols from Σ, it is not clear what to do with those symbols that are just thrown out during the projection. Rather, it is easy to translate $\Phi(\Sigma)$-sentences to Σ-sentences: since the $\Phi(\Sigma)$ can be thought to be a subset of the Σ-symbols, basically just keep sentences as they are.

- a translation of I-models to J-models. This translation basically should translate a Σ-model in I to a $\Phi(\Sigma)$-model in J by just leaving out the semantical interpretation of all those symbols that are left out when going from Σ to $\Phi(\Sigma)$.

There is a condition imposed on institution morphisms that is analogous to the satisfaction condition institutions: we require that a translated model satisfies a sentence iff the original model satisfies the translated sentence.

After these rather informal motivating explanations (which explain what is going on in typical, but not necessarily in all cases), we now come to the formal definition:

Given institutions I and J, an *institution morphism* $\mu = (\Phi, \alpha, \beta)\colon I \longrightarrow J$ consists of

- a functor $\Phi\colon \mathbf{Sign}^I \longrightarrow \mathbf{Sign}^J$,
- a natural transformation $\alpha\colon \mathbf{Sen}^J \circ \Phi \longrightarrow \mathbf{Sen}^I$ and
- a natural transformation $\beta\colon \mathbf{Mod}^I \longrightarrow \mathbf{Mod}^J \circ \Phi^{op}$,

such that the following *satisfaction condition* is satisfied for all $\Sigma \in \mathbf{Sign}^I$, $M \in \mathbf{Mod}^I(\Sigma)$ and $\varphi' \in \mathbf{Sen}^J(\Phi(\Sigma))$:

$$M \models^I_\Sigma \alpha_\Sigma(\varphi') \Leftrightarrow \beta_\Sigma(M) \models^J_{\Phi(\Sigma)} \varphi'$$

In more detail, the above definition means that each signature $\Sigma \in \mathbf{Sign}^I$ is translated to a signature $\Phi(\Sigma) \in \mathbf{Sign}^J$, and each signature morphism $\sigma\colon \Sigma \longrightarrow \Sigma' \in \mathbf{Sign}^I$ is translated to a signature morphism $\Phi(\sigma)\colon \Phi(\Sigma) \longrightarrow \Phi(\Sigma') \in \mathbf{Sign}^J$. Moreover, for each signature $\Sigma \in \mathbf{Sign}^I$, we have a sentence translation map $\alpha_\Sigma\colon \mathbf{Sen}^J(\Phi(\Sigma)) \longrightarrow \mathbf{Sen}^I(\Sigma)$ and a model translation functor $\beta_\Sigma\colon \mathbf{Mod}^I(\Sigma) \longrightarrow \mathbf{Mod}^J(\Phi(\Sigma))$. Naturality of α and β means that for any signature morphism $\sigma\colon \Sigma \longrightarrow \Sigma' \in \mathbf{Sign}^I$,

$$\begin{array}{ccc}
\mathbf{Sen}^I(\Sigma) & \xleftarrow{\alpha_\Sigma} & \mathbf{Sen}^J(\Phi(\Sigma)) \\
\downarrow{\scriptstyle \mathbf{Sen}^I(\sigma)} & & \downarrow{\scriptstyle \mathbf{Sen}^J(\Phi(\sigma))} \\
\mathbf{Sen}^I(\Sigma') & \xleftarrow{\alpha_{\Sigma'}} & \mathbf{Sen}^J(\Phi(\Sigma'))
\end{array}$$

and

$$\begin{array}{ccc}
\mathbf{Mod}^I(\Sigma) & \xrightarrow{\beta_\Sigma} & \mathbf{Mod}^J(\Phi(\Sigma)) \\
\uparrow{\scriptstyle \mathbf{Mod}^I(\sigma)} & & \uparrow{\scriptstyle \mathbf{Mod}^J(\Phi(\sigma))} \\
\mathbf{Mod}^I(\Sigma') & \xrightarrow{\beta_{\Sigma'}} & \mathbf{Mod}^J(\Phi(\Sigma'))
\end{array}$$

commute.

Example 2.11 There is an institution morphism going from first-order logic with equality to equational logic. A first-order signature is translated to an algebraic signature by just forgetting the set of predicate symbols; similarly, a first-order model is turned into an algebra by forgetting the predicate. Sentence translation is just inclusion of equations into first-order sentences. □

Let $\mu = (\Phi, \alpha, \beta)\colon I \longrightarrow I'$ and $\mu' = (\Phi', \alpha', \beta')\colon I' \longrightarrow I''$ be two institution morphisms. Then the composition $\mu'' = \mu' \circ \mu\colon I \longrightarrow I''$ consists of the following components:

- $\Phi'' = \Phi' \circ \Phi$
- $\alpha''_\Sigma = \alpha_\Sigma \circ \alpha'_{\Phi\,\Sigma}$ $(\Sigma \in |\mathbf{Sign}^I|)$

- $\beta''_\Sigma = \beta'_{\Phi\,\Sigma} \circ \beta_\Sigma$ $(\Sigma \in |\mathbf{Sign}^I|)$

Together with obvious identities, this gives us the category **Ins** of institutions and institution morphisms.

2.6 Institution Comorphisms

Somewhat dually to institution morphisms, institution comorphisms allow to express that fact that one institution I is *included* into an institution J. An *institution comorphism* from an institution I to an institution J consists of the following components:

- a translation Φ of I-signatures to J-signatures. Given an I-signature Σ, the task is to find a J-encoding $\Phi(\Sigma)$ of Σ in some way. In particular, the model category of $\Phi(\Sigma)$ should approximate the model category of Σ somehow.
- a translation α of I-sentences to J-sentences. The reason why the sentence translation goes *along* with the signature translation is similar to the reason why the sentence translation *within* an institution goes along with the signature morphism. Namely, if a signature Σ in I is encoded by the signature $\Phi(\Sigma)$ in J, it is expected that each symbol in Σ is translated to some corresponding symbol in $\Phi(\Sigma)$. Now if we assume that a Σ-sentence φ is a derivation tree decorated with some symbols from Σ, the translation $\alpha_\Sigma(\varphi)$ just keeps the structure of the tree and translates the symbols according to the correspondence of symbols in Σ and $\Phi(\Sigma)$.
- a translation β of J-models to I-models, giving the relation between Σ-models in I and $\Phi(\Sigma)$-models in J. Here, we again have the contravariance of the model translation, as in the definition of institution. Often it happens that there is also a model translation γ in the opposite direction. However, while β is formalized as a natural transformation, γ is not always natural (see [KM95] for a counterexample). Naturality of β is essential for heterogeneous specification, see Chap. 6.

We impose a satisfaction condition on comorphisms as well: we require that a translated model satisfies a sentence iff the original model satisfies the translated sentence.

Definition 2.12 Given institutions I and J, an *institution comorphism* $\rho = (\Phi, \alpha, \beta)\colon I \longrightarrow J$ consists of

- a functor $\Phi\colon \mathbf{Sign}^I \longrightarrow \mathbf{Sign}^J$,
- a natural transformation $\alpha\colon \mathbf{Sen}^I \longrightarrow \mathbf{Sen}^J \circ \Phi$,
- a natural transformation $\beta\colon \mathbf{Mod}^J \circ \Phi^{op} \longrightarrow \mathbf{Mod}^I$

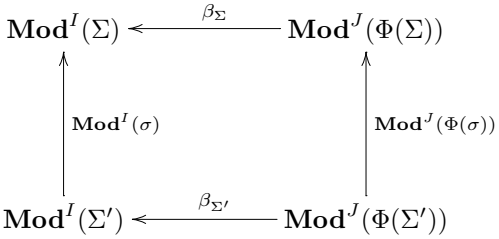
such that the following *satisfaction condition* is satisfied for all $\Sigma \in \mathbf{Sign}^I$, $M' \in \mathbf{Mod}^J(\Phi(\Sigma))$ and $\varphi \in \mathbf{Sen}^I(\Sigma)$:

$$M' \models^J_{\Phi(\Sigma)} \alpha_\Sigma(\varphi) \Leftrightarrow \beta_\Sigma(M') \models^I_\Sigma \varphi.$$

In more detail, this means that each signature $\Sigma \in \mathbf{Sign}^I$ is translated to a signature $\Phi(\Sigma) \in \mathbf{Sign}^J$, and each signature morphism $\sigma\colon \Sigma \longrightarrow \Sigma' \in \mathbf{Sign}^I$ is translated to a signature morphism $\Phi(\sigma)\colon \Phi(\Sigma) \longrightarrow \Phi(\Sigma') \in \mathbf{Sign}^J$. Moreover, for each signature $\Sigma \in \mathbf{Sign}^I$, we have a sentence translation map $\alpha_\Sigma\colon \mathbf{Sen}^I(\Sigma) \longrightarrow \mathbf{Sen}^J(\Phi(\Sigma))$ and a model translation functor $\beta_\Sigma : \mathbf{Mod}^J(\Phi(\Sigma)) \longrightarrow \mathbf{Mod}^I(\Sigma)$. Naturality of α and β means that for any signature morphism $\sigma\colon \Sigma \longrightarrow \Sigma' \in \mathbf{Sign}^I$,

$$\begin{array}{ccc} \mathbf{Sen}^I(\Sigma) & \xrightarrow{\alpha_\Sigma} & \mathbf{Sen}^J(\Phi(\Sigma)) \\ \downarrow{\scriptstyle \mathbf{Sen}^I(\sigma)} & & \downarrow{\scriptstyle \mathbf{Sen}^J(\Phi(\sigma))} \\ \mathbf{Sen}^I(\Sigma') & \xrightarrow{\alpha_{\Sigma'}} & \mathbf{Sen}^J(\Phi(\Sigma')) \end{array}$$

and



commute.

Example 2.13 There is an institution comorphism going from equational logic to first-order logic with equality. An algebraic signature is translated to a first-order signature by just taking the set of predicate symbols to be empty. Sentence translation is just inclusion of equations into first-order sentences. A first-order model with empty set of predicates is translated by just considering it as an algebra. □

Let $\rho = (\Phi, \alpha, \beta)\colon I \longrightarrow I'$ and $\rho' = (\Phi', \alpha', \beta')\colon I' \longrightarrow I''$ be two institution comorphisms. Then the composition $\rho'' = \rho' \circ \rho\colon I \longrightarrow I''$ consists of the following components:

- $\Phi'' = \Phi' \circ \Phi$
- $\alpha''_\Sigma = \alpha'_{\Phi\,\Sigma} \circ \alpha_\Sigma$ $(\Sigma \in |\mathbf{Sign}^I|)$
- $\beta''_\Sigma = \beta_\Sigma \circ \beta'_{\Phi\,\Sigma}$ $(\Sigma \in |\mathbf{Sign}^I|)$

Together with obvious identities, this gives us the category **CoIns** of institutions and institution comorphisms.

2.7 Simple Theoroidal (Co)Morphisms

Morphisms and comorphisms also come in a variant that maps signatures to theories.

Definition 2.14 A *simple theoroidal institution (co)morphism* from I to J is an institution (co)morphism (Φ, α, β) from I to J^{th}, where J^{th} is the institution of theories over J (see Sect. 2.2). If we want to stress that an institution (co)morphism is an ordinary one, we call it *plain*.

It is easy of extend $(\_)^{th}$ to a functor on **CoIns**, mapping a comorphism (Φ, α, β) to $(\Phi^\alpha, \alpha', \beta')$. Here, Φ^α is the extension of Φ to theories (using α for sentence translation), and α' and β' are suitable variants of α and β indexed by theories instead of signatures. Then, there is an institution comorphism $\eta_I\colon I \longrightarrow I^{th}$ mapping Σ to $(\Sigma, \emptyset)$ and an institution comorphism $\xi_I\colon (I^{th})^{th} \longrightarrow I^{th}$ mapping $((\Sigma, \Psi_1), \Psi_2)$ to $(\Sigma, \Psi_1 \cup \Psi_2)$. Both are the identity on sentences and models. It is rather straightforward to show that this yields a monad on **CoIns**, and using f Kleisli composition, institutions and simple theoroidal comorphisms form a category, see [RG04, Mos96b] for details.

On the other hand, simple theoroidal *morphisms* form a category only under additional assumptions (this is because Φ and α go in opposite directions, and hence $(\_)^{th}$ is not a functor on **Ins** from the outset). In the sequel, we will work with simple theoroidal comorphisms only: they capture the intuition of *encoding* an institution into another one (while examples for simple theoroidal morphisms are less frequent). Obviously, any institution comorphism also is a simple theoroidal one.

Definition 2.15 A simple theoroidal comorphism $\rho = (\Phi, \alpha, \beta)\colon I \longrightarrow J$ is said to be a *subinstitution comorphism* if Φ is an embedding of categories, α is a pointwise injection, and β is a natural isomorphism. I is said to be a *subinstitution* of J if there is a subinstitution comorphism from I to J. □

Example 2.16 The institution comorphism from Example 2.13 is a subinstitution comorphism. □

Example 2.17 Simple theoroidal institution comorphisms capture the encoding of a richer institution into a poorer one, as the following example shows: Define a simple theoroidal institution comorphism going from partial first-order logic with equality to first-order logic with equality as follows: A partial first-order signature is translated to a total one by encoding each partial function symbol as a total one, plus a (new) unary predicate D ("definedness") and a (new) function symbol $\bot$ ("undefined") for each sort (this means that $\bot$ and D are heavily overloaded). Furthermore, we add axioms stating that D does not hold on $\bot$, and that (encoded) total functions preserve ("totality") and reflect ("strictness") D, while partial functions only reflect D (and the holding of predicates implies D to hold on the arguments). Sentence translation is done by replacing all partial function symbols by the total functions symbols encoding them, replacing strong equations $t = u$ by $(D(t) \vee D(u)) \Rightarrow t = u$, existence equations by conjunctions of the equation and the definedness (using D) of one of the sides of the equation, replacing definedness with D, and leaving predicate symbols as they are. For a given total model of the translated signature, we just take as carriers of the partial model the interpretations of the definedness predicates in the total model, while the total functions are restricted to these new carriers, yielding partial functions. □

2.8 Intersections of Subinstitutions

In Sect. 3.1.9 below, we want to define subinstitutions of the CASL institution by removing different features from it. Therefore, we need *intersections* of subinstitutions.

Given a family $(\rho_k = (\Phi_k, \alpha^k, \beta^k)\colon J_k \longrightarrow J)_{k\in K}$ of plain subinstitution comorphisms, with $J_k = (\mathbf{Sign}^k, \mathbf{Sen}^k, \mathbf{Mod}^k, \models^k)$, their *intersection* I is defined as follows:

Signatures $\mathbf{Sign}^I := \bigcap_{k\in K} \Phi_k(\mathbf{Sign}^k)$. This is a category, since the Φ_k are embeddings, and categories are closed under taking the image along embeddings and under intersection.

Models $\mathbf{Mod}^I(\Sigma) := \mathbf{Mod}^J(\Sigma)$.

Sentences $\mathbf{Sen}^I(\Sigma) := \bigcap_{k\in K} \alpha^k_{\Phi_k^{-1}(\Sigma)}(\mathbf{Sen}^k(\Phi_k^{-1}(\Sigma)))$, and $\mathbf{Sen}^I(\sigma\colon \Sigma \longrightarrow \Sigma') : \mathbf{Sen}^I(\Sigma) \longrightarrow \mathbf{Sen}^I(\Sigma')$ is the domain-codomain-restriction of $\mathbf{Sen}^J(\sigma)$, which exists by naturality of α. The injection of $\mathbf{Sen}^I(\Sigma)$ into $\mathbf{Sen}^k(\Phi_k^{-1}(\Sigma))$ given by the appropriate restriction of $(\alpha^k_{\Phi_k^{-1}(\Sigma)})^{-1}$ is denoted by ι^k_Σ.

Satisfaction $M \models^I_\Sigma \varphi$ iff $M \models^J_\Sigma \varphi$ (hence, the satisfaction condition is inherited from J). □

Note that the intersection is a limit in the category **CoIns**, with limit projections $(\Phi_k^{-1}, \iota, (\beta^k \circ \Phi_k^{-1})^{-1})\colon I \longrightarrow J_k$ (well, injections would be a better name here, since they inject the intersection I in the J_k).

2.9 Adjointness Between Morphisms and Comorphisms

Institution morphisms and comorphisms are related by an adjunction.

Proposition and Definition 2.18 Given institutions I and J and functors $\Phi\colon \mathbf{Sign}^I \longrightarrow \mathbf{Sign}^J$ and $\Psi\colon \mathbf{Sign}^J \longrightarrow \mathbf{Sign}^I$ such that Φ is left adjoint to Ψ, there is a one-to-one correspondence between institution comorphisms $\rho = (\Phi, \alpha, \beta)\colon I \longrightarrow J$ and institution morphisms $\mu = (\Psi, \bar\alpha, \bar\beta)\colon J \longrightarrow I$. ρ and μ are called *adjoint* if they are in this correspondence.

For example, the institution morphism from Example 2.11 is adjoint to the institution comorphism from Example 2.13.

PROOF: If η is the unit and ε is the counit of the adjunction, then the one-to-one correspondence is given by

$$\bar{\alpha} = (\mathbf{Sen}^J \cdot \varepsilon) \circ (\alpha \cdot \Psi) \qquad \alpha = (\bar{\alpha} \cdot \Phi) \circ (\mathbf{Sen}^I \cdot \eta)$$
$$\bar{\beta} = (\beta \cdot \Psi^{op}) \circ (\mathbf{Mod}^J \cdot \varepsilon) \qquad \beta = (\mathbf{Mod}^I \cdot \eta) \circ (\bar{\alpha} \cdot \Phi^{op})$$

□

2.10 A Taxonomy of (Co)Morphisms

The notion of institution morphism can be varied in several ways by changing the directions of the arrows or even, in the case of semi-morphisms, omitting some arrows.

	morphism	comorphism	
Sign	$\longrightarrow$	$\longrightarrow$	$\mathbf{Sign}'$
Sen	$\longleftarrow$	$\longrightarrow$	$\mathbf{Sen}' \circ \Phi$
Mod	$\longrightarrow$	$\longleftarrow$	$\mathbf{Mod}' \circ \Phi$

	forward morphism	forward comorphism	
Sign	$\longrightarrow$	$\longrightarrow$	$\mathbf{Sign}'$
Sen	$\longrightarrow$	$\longleftarrow$	$\mathbf{Sen}' \circ \Phi$
Mod	$\longrightarrow$	$\longleftarrow$	$\mathbf{Mod}' \circ \Phi$

	semi morphism	semi comorphism	
Sign	$\longrightarrow$	$\longrightarrow$	$\mathbf{Sign}'$
Sen			$\mathbf{Sen}' \circ \Phi$
Mod	$\longrightarrow$	$\longleftarrow$	$\mathbf{Mod}' \circ \Phi$

The respective satisfaction conditions are quite obvious (note that for semi-(co)morphisms, none is required). The naming scheme follows a model-theoretic view — the name main part of the name is determined by the action on models:

- morphisms have model translations along the signature translation, while
- comorphisms have it against the signature translation.

The sentence translation is determined by the qualification:

- no qualification means that sentence translation is done against the model translation,
- "forward" means that it is done along the model translation, and
- "semi" means that there is no sentence translation at all.

Let **semi-Ins** (**semi-coIns**) be the category of institutions and semi-morphisms (semi-comorphisms). These notions are not just a formal game, but are relevant in practice. Let us now examine in more detail typical practical situations.

Let two institutions (for specification purposes) be given, a "poorer" one P and a "richer" one R having some more features. Then typically, there are four different types of translations between them, serving different purposes:

1. trivial inclusions $\rho: P \longrightarrow R$, which are usually comorphisms. In heterogeneous CASL (see Chap. 6 and Appendix A), we provide a construct SP **with logic** ρ allowing to move a specification into the richer logic for the purpose of heterogeneous specification;
2. trivial projections $\mu: R \longrightarrow P$ forgetting R's additional features (they are usually morphisms). In heterogeneous CASL, we provide a construct SP **hide logic** μ allowing to project a specification into the poorer logic. Typically, such a μ is adjoint (in the sense of Proposition and Definition 2.18) to some inclusion ρ;

3. codings $\rho: R \longrightarrow P$ encoding R's features within P, typically coming as simple theoroidal comorphisms or as forward morphisms. The corresponding construct SP **with logic** ρ typically is not used for writing heterogeneous specifications, but for exploiting borrowing (see Sect. 5.2, 6.3) during heterogeneous proofs;

4. "feature-interaction" either comes as a projection $\mu: R \longrightarrow P$ projecting R onto P while keeping R's additional features within P by making them explicit (i.e. by providing explicit signature elements in P for features that are implicit, or "logical", in R), or as a coding of R's additional features in P. These come as morphisms, and provide a form of *feature interaction* during heterogeneous specification (which initially seems only to put logics side by side).

When also considering institutions for programming languages, two additional situations may arise: Firstly, a *semi-morphism* going from a more implementation-oriented institution (e.g. a programming language) to a more specification-oriented institution, allows to express *implementations* [ST88c]. Secondly, sometimes it is also possible to automatically translate an executable sublanguage of a specification language to a programming language; this situation can be modeled by comorphisms that are similar to the inclusion comorphisms above.

We illustrate our taxonomy with some simple examples. Let FOL be first-order logic and $FOL^=$ be first-order logic with equality (a detailed formalization as institution can be found in [GB92]).

The trivial inclusion of FOL into $FOL^=$ is easily formalized as an institution comorphism, while the trivial projection from $FOL^=$ to FOL is a institution morphism. Both are the identity on signatures and models, and include the set of sentences without equality into the set of sentences with equality.

Concerning coding comorphisms, consider the simple theoroidal comorphism $\rho: FOL^= \longrightarrow FOL$. For signatures, it adds predicate symbols $\equiv: s \times s$ (overloaded for each sort s) that are axiomatized to be congruences. Models are translated by factoring their carriers w.r.t. $\equiv$. Sentence translation is done by replacing $=$ with $\equiv$.

Finally, there is a feature interaction institution morphism μ going from first-order logic with equality $FOL^=$ to first-order logic FOL defined in [RG04]. At the signature level, it adds a binary equality predicate eq_s for each sort s, which is interpreted as the equality relation when translating models. In sentences, the explicit equality predicate eq_s (in FOL) is translated to the built-in equality $=$ in $FOL^=$. So this morphism makes equality explicit.

These four types of translations all deal with the feature of equality: inclusions add it, projections forget it, codings make it explicit by axiomatizing it, and feature interaction translations make it explicit by naming it and keeping it in the models. We have chosen these examples involving FOL and $FOL^=$ because they are simple but quite typical. For example, between CASL and CASL-LT, the same four kinds of translations exists, some of them are sketched in [Mos02b]. E.g. the feature interaction morphism makes the labeled transition structure explicit.

In the literature, a whole bunch of different types of translations has been used. The following table partitions them by some informal classification scheme (a "th" stands for the simple theoroidal case, "semi" denotes semi-morphisms, and an "x" stand for folklore knowledge or trivialities):

	(semi) morphism	(simple theoroidal) comorphism	(simple theoroidal) forward morphism	forward comorphism
Inclusion	[STb]	[AF96, Mes89b]	x	x
Coding	([STb, Ala02])[4]	th [AC92, Cer93] [CM97, KM95] [MOM95, Mes89b] [Mes92, Mos96a] [Mos02, MKK98] [Tar87]	th [BHa] [WDC+95, SS93] [SS96, Sco93])[5]	
Projection	[AF96, STb, DF02]			
Feature interaction	[Mos02b]	x/th		
Implementation	semi [ST88c] [Tar96, STb]	x	[WDC+95]	

2.11 Properties of Institution Comorphisms

We now introduce various properties of institution comorphisms that will play a rôle for re-use of proof calculi (so-called Borrowing, Sect. 5.2) and heterogeneous specification (Chap. 6). A first property, useful for borrowing for flat specifications, is the model expansion property:

Definition 2.19 An institution (co)morphism (Φ, α, β) *admits model expansion* if β is pointwise surjective on objects (i.e., each β_Σ is surjective on objects). □

Example 2.20 The institution comorphisms from Examples 2.13 and 2.17 admit model expansion. For the former one, this is trivial. For the latter one, any partial model can be completed to a total model by adding one element to each carrier (as interpretation of $\bot$), representing "undefined", which is a fixpoint of all functions, while predicates do not hold on it. Then, this one-point completion just generates the original model via the model translation. □

Amalgamation resp. exactness can be lifted to comorphisms as follows:

Definition 2.21 Let $\rho = (\Phi, \alpha, \beta): I \longrightarrow J$ be an institution comorphism and let $\mathcal{D}$ be a class of signature morphisms in I. Then ρ is said to have the *(weak) $\mathcal{D}$-amalgamation property*, if for each signature morphism $\sigma: \Sigma_1 \longrightarrow \Sigma_2 \in \mathcal{D}$, the diagram

$$\begin{array}{ccc} \mathbf{Mod}^I(\Sigma_2) & \xleftarrow{\beta_{\Sigma_2}} & \mathbf{Mod}^J(\Phi(\Sigma_2)) \\ \downarrow{\scriptstyle \mathbf{Mod}^I(\sigma)} & & \downarrow{\scriptstyle \mathbf{Mod}^J(\Phi(\sigma))} \\ \mathbf{Mod}^I(\Sigma_1) & \xleftarrow{\beta_{\Sigma_1}} & \mathbf{Mod}^J(\Phi(\Sigma_1)) \end{array}$$

admits (weak) amalgamation, i.e. any for any two models $M_2 \in \mathbf{Mod}^I(\Sigma_2)$ and $M_1' \in \mathbf{Mod}^J(\Phi(\Sigma_1))$ with $M_2|_\sigma = \beta_{\Sigma_1}(M_1')$, there is a unique (not necessarily unique) $M_2' \in \mathbf{Mod}^J(\Phi(\Sigma_2))$ with $\beta_{\Sigma_2}(M_2') = M_2$ and $M_2'|_{\Phi(\sigma)} = M_1'$. In case that $\mathcal{D}$ consists of all signature morphisms, the (weak) $\mathcal{D}$-amalgamation property is also called (weak) *exactness*. The corresponding notions for institution morphisms are defined similarly. □

[4] It is not entirely clear whether these should be really called encodings, since —unlike the other codings in this row— it is not clear that they are suitable for re-use of theorem provers.

[5] Salibra and Scollo introduce a relaxed kind of forward morphism mapping models to sets of models.

Example 2.22 The institution comorphism from Example 2.13 trivially satisfies the weak $\mathcal{D}$-amalgamation property, where $\mathcal{D}$ is the class of all signature morphisms in $Eq^{=}$, since the model translations β_Σ are isomorphisms. □

Example 2.23 Let $\mathcal{D}$ be the class of all injective signature morphisms in $PFOL^{=}$. Weak $\mathcal{D}$-amalgamation for the institution comorphism from Example 2.17 can be seen as follows: Let $\sigma: \Sigma_1 \longrightarrow \Sigma_2 \in \mathcal{D}$, let M_2 be a Σ_2-model and M_1' be a $\Phi(\Sigma_1)$-model such that $M_2|_\sigma = \beta_{\Sigma_1}(M_1')$. Extend M_1' to a $\Phi(\Sigma_2)$-model M_2' as follows: For any sort s not in the image of σ, let the carrier for sort $\sigma(s)$ in M_2' just be $(M_2)_s \uplus \{*\}$, and let $D_{M_2'}$ hold everywhere except on $*$. $\bot$ is interpreted as $*$ in M_2'. Given a function symbol f in Σ outside the image of σ, $\sigma(f)$ is interpreted in M_2' to be f_{M_2}, except that the interpretation of $\bot$ is delivered if the argument is outside M_2 or the result is not defined due to partiality of the function. Given a predicate symbol p in Σ outside the image of σ, $\sigma(p)$ is interpreted in M_2' to be p_{M_2}, except that it is false if the argument is outside M_2. Then, we have that $\beta_{\Sigma_2}(M_2') = M_2$ and $M_2'|_{\Phi(\sigma)} = M_1'$, showing weak $\mathcal{D}$-amalgamation. □

We will also need the following strengthening of the weak amalgamation property:

Definition 2.24 An institution (co)morphism $\mu = (\Phi, \alpha, \beta): I \longrightarrow J$ is said to be *model-bijective* if for each $\Sigma \in \mathbf{Sign}^I$, β_Σ is a bijection on objects. □

A property ensuring a good interaction with liberality of institutions (to be introduced in Def. C.4) is the following one, which we have introduced in [KM95] in a slightly stronger form under the name of categorical retractive simulation:

Definition 2.25 An institution comorphism $\rho = (\Phi, \alpha, \beta): I \longrightarrow J$ is called *persistently liberal* if for each $\Sigma \in \mathbf{Sign}^I$, β_Σ has a left adjoint γ_Σ such that also $\beta_\Sigma \circ \gamma_\Sigma \cong id$. If we have even $\beta_\Sigma \circ \gamma_\Sigma = id$, then ρ is called *strongly persistently liberal*. If moreover each β_Σ has a right adjoint δ_Σ, ρ is called *strongly persistently bi-liberal*. We write (ρ, γ) resp. (ρ, γ, δ) if we want to chose a particular γ and δ. Note that γ and δ are not required to be natural transformations. □

Example 2.26 The institution comorphism from Examples 2.13 is strongly persistently liberal: γ_Σ is just the inverse of the isomorphism β_Σ. □

Example 2.27 The institution comorphism from Example 2.17 is strongly persistently liberal: γ_Σ totalizes a partial model by adding "undefined" values freely (this is the free completion [Bur86, BLR02]). □

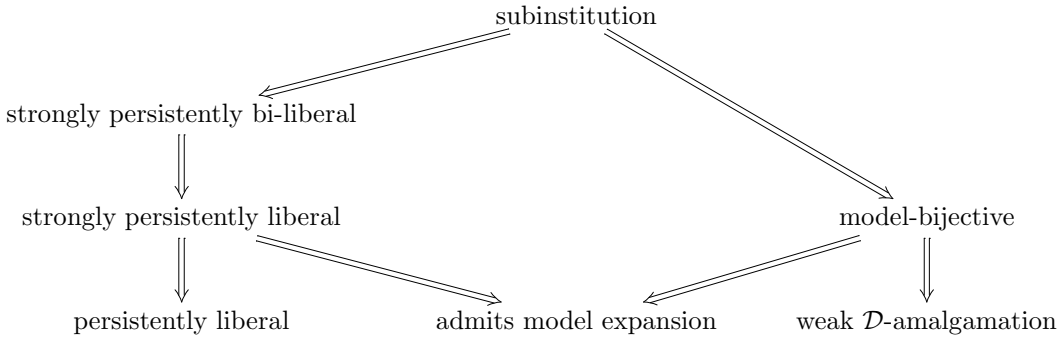
Proposition 2.28 Given a strongly persistently bi-liberal institution comorphism $((\Phi, \alpha, \beta), \gamma, \delta)$, we always have

$$\beta_\Sigma \circ \delta_\Sigma \cong id.$$

PROOF: By composability of adjoints, $\beta_\Sigma \circ \gamma_\Sigma$ is left adjoint to $\beta_\Sigma \circ \delta_\Sigma$. Now the identity functor is left adjoint to itself, and left adjoints are unique up to natural isomorphism. Hence, $\beta_\Sigma \circ \delta_\Sigma \cong id$. □

We now briefly compare the different notions of comorphisms that have been introduced so far.

Proposition 2.29 The implication shown below hold. In particular, any model-bijective institution comorphism admits model expansion and weak $\mathcal{D}$-amalgamation for arbitrary $\mathcal{D}$. Any subinstitution comorphism is model-bijective and strongly persistently bi-liberal such that both γ and δ are natural transformations.



□

We now present a number of counterexamples showing that the above diagram of implications is optimal.

Example 2.30 The institution comorphism from Examples 2.17 and 2.27 admits model expansion and it is strongly persistently liberal, but neither model-bijective admitting weak $\mathcal{D}$-amalgamation for $\mathcal{D}$ consisting of non-injective signature morphisms: for a given partial model, it is possible to add any number of "undefined" elements in a model representing it. □

Example 2.31 Modify the institution comorphism from Examples 2.17 and 2.27 by omitting the functions $\bot$ and the axioms involving $\bot$. Then the resulting institution comorphism is still strongly persistently liberal, but it does not admit weak $\mathcal{D}$-amalgamation, where $\mathcal{D}$ is the class of all signature morphisms adding a partial function symbol to a signature. Let $\sigma\colon \Sigma \longrightarrow \Sigma' \in \mathcal{D}$. Take any Σ'-model M containing a truly partial function and such that all functions in $M|_\sigma$ are total. Then $M|_\sigma$ can be represented by a model M' (i.e. $\beta_\Sigma(M') = M|_\sigma$) that has no "undefined" elements at all. However, it is not possible to represent M with an extension of M', since this would require the presence of some "undefined" element in some carrier of M' (and hence also M). □

Example 2.32 Modify the institution comorphism from Example 2.17 as follows: Add to $\Phi(\Sigma)$ a sentence $\neg\ x = \bot_s \Rightarrow\ D_s(x)$ (for each sort s in Σ). Thus, $\bot_s$ becomes the *unique* "undefined" element. Now a two-sided inverse of the model translation can be constructed as follows: For a partial Σ-structure M, form its *one-point completion* by just adding one element, $*$ (which is the interpretation of $\bot_s$), to all carriers, let all functions map $*$ to itself and behave as in M otherwise, where undefinedness of partial functions is mapped to $*$. Predicates are false on $*$. The interpretation of the predicates D is fixed by their defining axioms. This shows the comorphism to be model-bijective (and hence it also admits model expansion).

The above defined inverse to the model translation defines a bijection on model classes, but not an isomorphism of model categories. This is because a Σ-homomorphism need only preserve definedness of partial functions, while a $\Phi(\Sigma)$-homomorphism has to preserve *and reflect* "definedness" of the comorphisms of partial functions. Thus, the above inverse construction, being the unique inverse construction on models, cannot be extended to a functor. This shows that the above institution comorphism is neither a subinstitution comorphism nor persistently liberal.

Now consider the $PFOL^=$-signature Σ with one sort s and one partial function symbol $f\colon s \longrightarrow ?s$, and let σ be the inclusion of the signature consisting just of sort s into Σ. Given a set X, the σ-free Σ-model M over X is just X with f interpreted as the everywhere undefined function. The unique representation M' of M is X with one "undefined" element, and f yielding everywhere the "undefined" element. Now M' is not free over X, since any homomorphism starting from M' has to preserve the undefined element and thus there cannot be a homomorphism from M' into some model where f is defined at some point. This shows that the assumption of persistent liberality is really needed in Theorem C.8. □

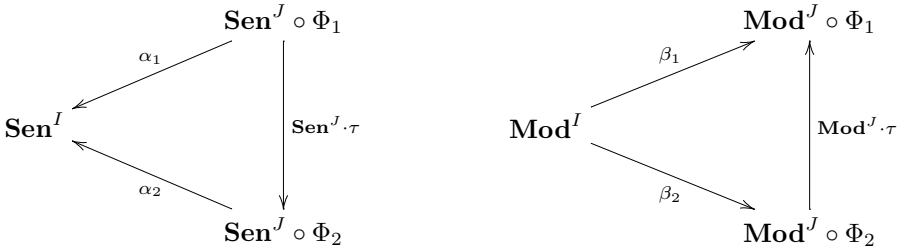
Example 2.33 Let $\mathcal{D}$ be the class of all signature isomorphisms in an institution I. Since functors preserve isomorphisms, any institution comorphism starting from I admits weak $\mathcal{D}$-amalgamation. Clearly, not every such comorphism admits model expansion. □

2.12 Institution (Co)Morphism Modifications

Sometimes it is useful to indicate that two institution (co)morphisms differ only in an inessential way. This in particular applies when the (co)morphisms arise as compositions of other (co)morphisms. We therefore introduce the notion of *modification*.

The main motivation behind the use of modifications is to identify comorphisms that are related by a modification for the purpose of heterogeneous specification (cf. Chap. 6). Indeed, this notion will later on serve to reduce the number of composite comorphisms shown to the user of the Heterogeneous Tool Set (cf. Chap. 7). Fig. 2.1 contains a menu of translations possible for a CASL specification; using modifications, the number of possible translations can be greatly reduced. A crucial property of modifications therefore is that in this identification process, they do not lead to identifications of different sentence or model translation maps. Hence, we strengthen the original notion from [Dia02] to *discrete* modifications:

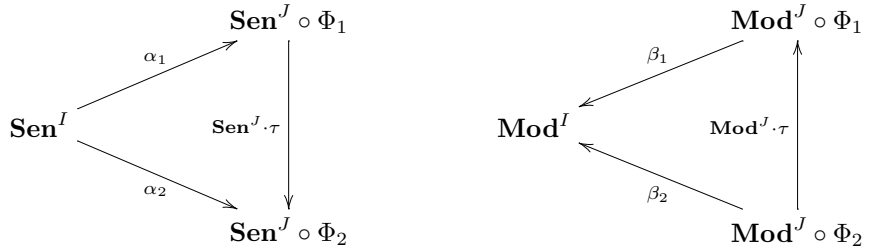
Definition 2.34 Given two institution morphisms $\mu_1, \mu_2 \colon I \longrightarrow J$ with $\mu_i = (\Phi_i, \alpha_i, \beta_i)$, a (discrete) *institution morphism modification* $\tau \colon \mu_1 \longrightarrow \mu_2$ is a natural transformation $\tau \colon \Phi_1 \longrightarrow \Phi_2$ such that



commute.

In [Dia02, Dia], the first condition is omitted, and the second condition is replaced by the existence of a natural transformation between β_1 and $(\mathbf{Mod}^J \cdot \tau) \circ \beta_2$. We have not found this extra generality of practical use and hence work with the above stronger notion of discrete modification. However, since we will not use any non-discrete modification, we will omit the qualification of being discrete henceforth.

Dually, given two institution comorphisms $\rho_1, \rho_2 \colon I \longrightarrow J$ with $\rho_i = (\Phi_i, \alpha_i, \beta_i)$, an *institution comorphism modification* $\tau \colon \rho_1 \longrightarrow \rho_2$ is a natural transformation $\tau \colon \Phi_1 \longrightarrow \Phi_2$ such that



commute.

Example 2.35 There are two ways to go from equational logic to first-order logic: one is the obvious subinstitution comorphism ρ_1 from Example 2.13, the other one is the composition ρ_2 of the obvious subinstitution comorphism from equational logic to partial first-order logic composed with the encoding of partial first-order logic into first-order logic from Example 2.17. (Actually, since

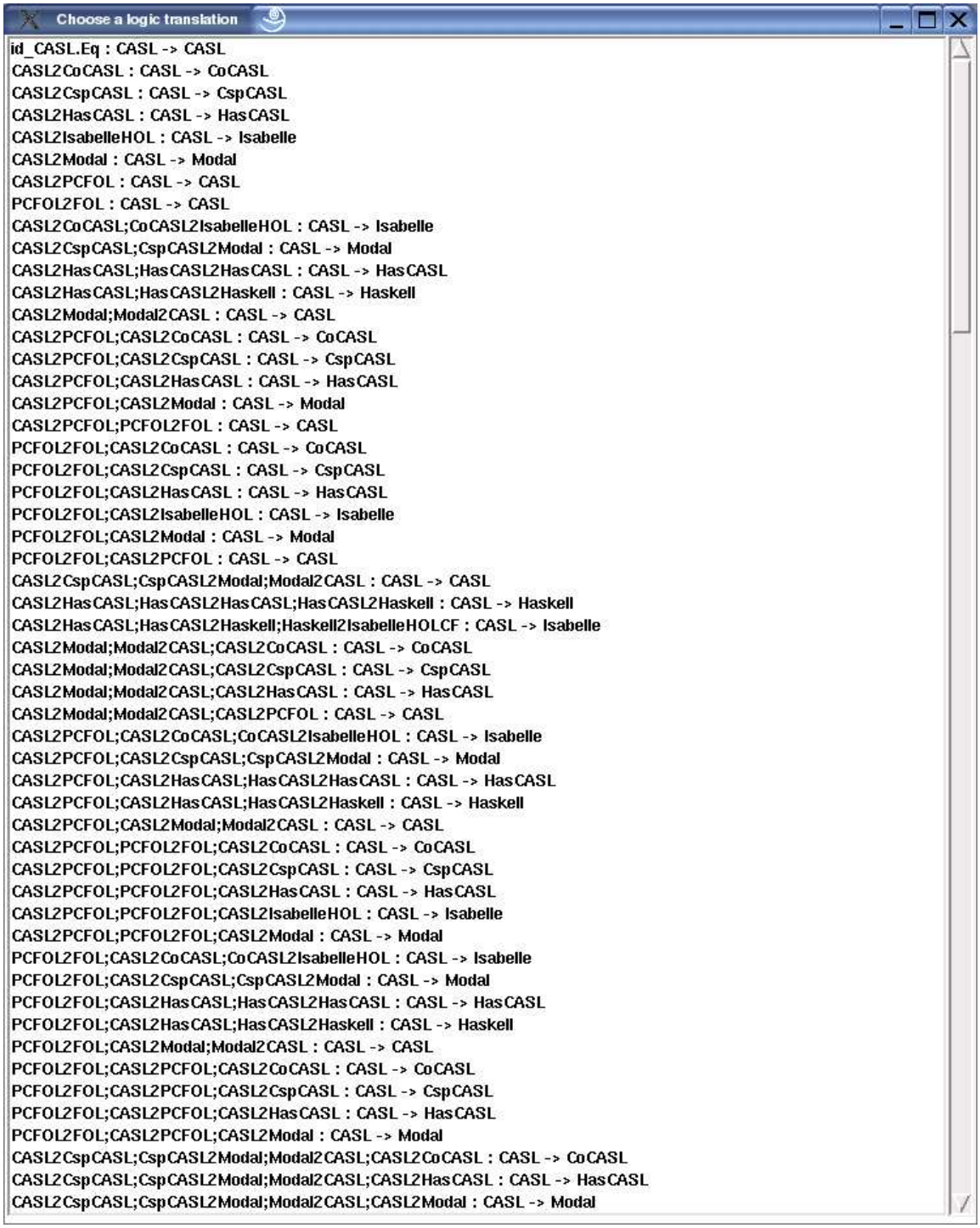
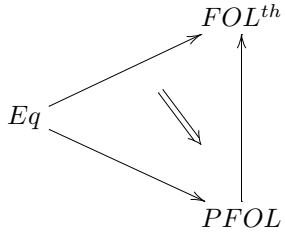


Figure 2.1: Dozens of translation possibilities for a CASL theory in HETS (from a logic graph without comorphism modifications; using modifications, the number of possible translations can be greatly reduced).

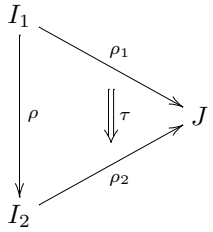
the latter is a simple theoroidal comorphism, we take both to end in FOL^{th}.) These comorphisms are different: ρ_2 adds some (superfluous) coding of partiality. The comorphism modification $\tau\colon \rho_1 \longrightarrow \rho_2$ is just the pointwise inclusion of an algebraic signature viewed as first-order signature into the theory coding a partial variant of that signature.



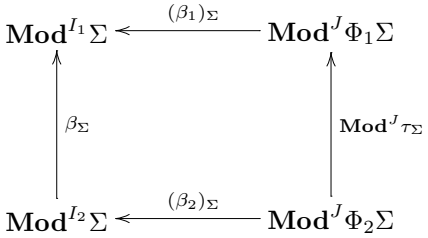
Together with obvious identities and compositions, modifications can serve as 2-cells, leading to 2-categories **Ins** and **CoIns**.

Modifications also interplay with amalgamation:

Definition 2.36 Given comorphisms $\rho = (\Phi, \alpha, \beta)\colon I_1 \longrightarrow I_2$, $\rho_1 = (\Phi_1, \alpha_1, \beta_1)\colon I_1 \longrightarrow J$ and $\rho_2 = (\Phi_2, \alpha_2, \beta_2)\colon I_2 \longrightarrow J$, a lax triangle



of institution comorphisms and modifications is called *(weakly) amalgamable*, if



is a (weak) pullback for each signature $\Sigma \in \mathbf{Sign}^I$.

2.13 Institutions as Functors

Institutions can be alternatively characterized as functors into a certain category of rooms. This view will be convenient in the treatment of heterogeneous specification in Chap. 6. The idea is to collect the satisfaction system local to a signature into a so-called *room*.

An *institution room* $(S, \mathcal{M}, \models)$ consists of

- a set of S of *sentences*,
- a category $\mathcal{M}$ of *models*, and
- a satisfaction relation $\models \subseteq |\mathcal{M}| \times S$.

Rooms are connected via corridors (which model change of notation within one logic, as well as translations between logics).

An *institution corridor* $(\alpha, \beta)\colon (S_1, \mathcal{M}_1, \models_1) \longrightarrow (S_2, \mathcal{M}_2, \models_2)$ consists of

- a sentence translation function $\alpha\colon S_1 \longrightarrow S_2$, and
- a model reduction functor $\beta\colon \mathcal{M}_2 \longrightarrow \mathcal{M}_1$, such that

$$M_2 \models_2 \alpha(\varphi_1) \Leftrightarrow \beta(M_2) \models_1 \varphi_1$$

holds for each $M_2 \in \mathcal{M}_2$ and each $\varphi_1 \in S_1$ (*satisfaction condition*).

Semantic entailment in an institution room is defined as usual: for $\Psi \subseteq S, \varphi \in S$, we write $\Psi \models \varphi$, if all models satisfying Ψ also satisfy φ.

A *logic room* $(S, \mathcal{M}, \models, \vdash)$ is an institution room $(S, \mathcal{M}, \models)$ equipped with an *entailment relation* $\vdash \subseteq \mathcal{P}(S) \times S$, such that the following conditions are satisfied:

1. *reflexivity:* for any $\varphi \in S$, $\{\varphi\} \vdash \varphi$,
2. *monotonicity:* if $\Psi \vdash \varphi$ and $\Psi' \supseteq \Psi$ then $\Psi' \vdash \varphi$,
3. *transitivity:* if $\Psi \vdash \varphi_i$, for $i \in I$, and $\Psi \cup \{\varphi_i \mid i \in I\} \vdash \psi$, then $\Psi \vdash \psi$,
4. *soundness:* for any $\Psi \subseteq S$ and $\varphi \in S$,

$$\Psi \vdash \varphi \text{ implies } \Psi \models \varphi.$$

A logic room will be called *complete* if, in addition, the converse of the above implication holds.

A *logic corridor* $(\alpha, \beta)\colon (S_1, \mathcal{M}_1, \models_1, \vdash_1) \longrightarrow (S_2, \mathcal{M}_2, \models_2, \vdash_2)$ consists of an institution corridor $(\alpha, \beta)\colon (S_1, \mathcal{M}_1, \models_1) \longrightarrow (S_2, \mathcal{M}_2, \models_2)$ such that if $\Psi \vdash_1 \varphi$, then $\alpha(\Psi) \vdash_2 \alpha(\varphi)$ ($\vdash$-*translation*). Together with obvious notions of composition and identity, this gives us categories **InsRoom** and **LogRoom**.

Now, an *institution* can be defined to be just a functor $I\colon \mathbf{Sign} \longrightarrow \mathbf{InsRoom}$ (where **Sign** is called the category of *signatures*), and a *logic* is a functor $L\colon \mathbf{Sign} \longrightarrow \mathbf{LogRoom}$. In terms of the standard notation, the institution room $I(\Sigma)$ is written as $(\mathbf{Sen}^I(\Sigma), \mathbf{Mod}^I(\Sigma), \models_\Sigma)$. $\mathbf{Mod}^I$ and $\mathbf{Sen}^I$ can easily be seen to be functors, and we arrive at the standard definition of institution as a quadruple $(\mathbf{Sign}, \mathbf{Sen}, \mathbf{Mod}, \models)$.

The definition of morphisms and comorphisms is very easy: Given institutions $I_1\colon \mathbf{Sign}_1 \longrightarrow \mathbf{InsRoom}$ and $I_2\colon \mathbf{Sign}_2 \longrightarrow \mathbf{InsRoom}$, an *institution morphism* $(\Psi, \mu)\colon I_1 \longrightarrow I_2$ consists of a functor $\Psi\colon \mathbf{Sign}_1 \longrightarrow \mathbf{Sign}_2$ and a natural transformation $\mu\colon I_2 \circ \Psi \longrightarrow I_1$. In contrast, an *institution comorphism* $(\Phi, \rho)\colon I_1 \longrightarrow I_2$ consists of a functor $\Phi\colon \mathbf{Sign}_1 \longrightarrow \mathbf{Sign}_2$ and a natural transformation $\rho\colon I_1 \longrightarrow I_2 \circ \Psi$. We have thus recovered the categories **Ins** and **CoIns**. Logic morphisms and comorphisms are defined analogously, leading to categories **Log** and **coLog**.

We also easily recast the notion of modification in the new setting: Given institution morphisms $(\Psi, \mu)\colon I_1 \longrightarrow I_2$ and $(\Psi', \mu')\colon I_1 \longrightarrow I_2$, an *institution morphism modification* $\theta\colon (\Psi, \mu) \longrightarrow (\Psi', \mu')$ is just a natural transformation $\theta\colon \Psi \longrightarrow \Psi'$ such that $\mu = \mu' \circ (I_2 \cdot \theta)$. Similarly, given institution comorphisms $(\Phi, \rho)\colon I_1 \longrightarrow I_2$ and $(\Phi', \rho')\colon I_1 \longrightarrow I_2$, an *institution comorphism modification* $\theta\colon (\Phi, \rho) \longrightarrow (\Phi', \rho')$ is a natural transformation $\theta\colon \Phi \longrightarrow \Phi'$ such that $(I_2 \cdot \theta) \circ \rho = \rho'$.

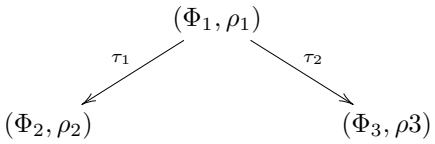
The corresponding notions for logics are entirely analogous.

2.14 Colimits in Hom-Categories

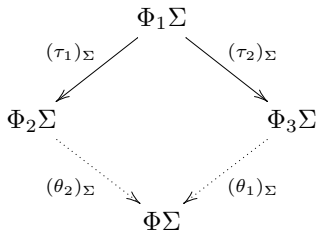
As a first application, we show that results about the 2-categorical structure of **CoIns** can be proved in a concise way using institutions as functors:

Proposition 2.37 Given two institutions I and J, if J has pushouts of signatures, then the Hom-category $\mathbf{CoIns}(I, J)$ has pushouts as well. This generalizes to arbitrary non-empty colimits.

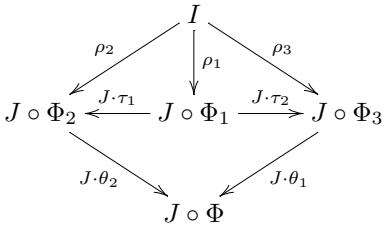
PROOF: Given comorphisms $(\Phi_i, \rho_i)\colon I \longrightarrow J$ $(i = 1, 2, 3)$ and a span of modifications



construct the signature component $\Phi\Sigma$ of the resulting comorphism as the pushout



By the universal property of the pushout, this extends to a functor $\Phi\colon \mathbf{Sign}^I \longrightarrow \mathbf{Sign}^J$ such that $\theta_1\colon \Phi_3 \longrightarrow \Phi$ and $\theta_2\colon \Phi_2 \longrightarrow \Phi$ become natural transformations.



We can then define room component of the pushout comorphism $\rho\colon I \longrightarrow J \circ \Phi$ to be $J \cdot \theta_2 \circ \rho_2 = J \cdot \theta_1 \circ \rho_3$, and the cocone consisting of $\theta_1\colon (\Phi_3, \rho_3) \Longrightarrow (\Phi, \rho)$ and $\theta_2\colon (\Phi_2, \rho_2) \Longrightarrow (\Phi, \rho)$ is easily seen to satisfy the universal property of a pushout.

The proof for coproducts, coequalizers or arbitrary non-empty colimits is very similar. □

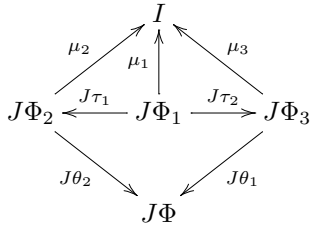
Note that initial objects in Hom-categories $\mathbf{CoIns}(I, J)$ generally do not exist: an initial comorphism from I to J would have to translate I-sentences to J-sentences over the initial signature, thereby losing any specific reference to the signature, which generally destroys the satisfaction condition.

The dual situation is better for initial objects:

Proposition 2.38 Given two institutions I and J, if J has an initial signature with empty set of sentences and terminal model category, then the Hom-category $\mathbf{Ins}(I, J)$ has an initial object.

PROOF: The initial institution morphism $(\Phi, \mu)\colon I \longrightarrow J$ is defined by letting $\Phi\Sigma$ be the initial signature, and μ_Σ consist of the empty map of sentences and the unique functor into the terminal model category. □

Concerning pushouts for $\mathbf{Ins}(I, J)$,



it is difficult to construct $\mu\colon J\Phi \longrightarrow I$. While the model translation component could be constructed using amalgamation, it is unclear how to map a J-sentence over the pushout signature $\Phi\Sigma$ to a

Σ-sentence in I, based on corresponding mappings for the signatures $\Phi_2\Sigma$ and $\Phi_3\Sigma$: generally, a sentence in $\Phi\Sigma$ may mix symbols of both $\Phi_2\Sigma$ and $\Phi_3\Sigma$. Hence, pushouts in $\mathbf{Ins}(I, J)$ seem to exist only under rather strong additional assumptions.

2.15 Polymorphism in an Arbitrary Institution

Type class polymorphism has been used in programming languages like Haskell [PJ03], as well as in the higher-order logic of Isabelle [Wen97]. It is one of the central features of the recently developed specification language HasCasl (see Sect. 3.4). Little attention has been paid in the literature to the question whether type class polymorphism can be formalized as an institution, the main problem here being that with the 'naive' semantics, the satisfaction condition fails in the sense that satisfaction of polymorphic axioms is preserved only by model reduction, not by model expansion, because expanded models may have more types. Thus, the naive semantics defines only a so-called rps preinstitution [SS93] rather than an institution.

The work of [NP86] is an initial attempt to define an institution for polymorphism but imposes severe restrictions on signature morphisms by simply ruling out the introduction of new types. For the case of polymorphism without type classes, one solution is to parameterize the notion of model by a fixed universe of types [Bor00, KBS88]; this solution, however, does not seem to be suitable for type class polymorphism.

The main goal here is to provide a semantics that avoids both problems, i.e. caters for type classes and works with the usual structured specification style where the signature is built up successively. In particular, we wish to avoid restrictions on *signature morphisms*; instead, we argue that the failure of the satisfaction condition points to a flaw in the notion of *model*. The key idea is to notice that polymorphic axioms are intended as statements about all types including those yet to be declared, and that therefore models should take into account future extensions. Starting from this observation, we obtain a general procedure that transforms a preinstitution into an institution, the so-called institution of *extended models*. This construction is employed in the semantics of HasCasl. It turns out that the notions of semantic consequence and model-expansive extension engendered by the construction agree with intuitive expectations, at least in sufficiently rich logics such as the logic of HasCasl.

More generally, HasCasl's treatment of polymorphic sentences can be subsumed under a definition of polymorphic formulae in institutions introduced here. Such generic polymorphic frameworks are perfect candidates for the extended model construction, and indeed it turns out that the notion of semantic consequence in the institution of extended models over a generic polymorphic framework is simpler and more natural than the original notion.

There are several other known examples of logical frameworks where the satisfaction condition fails unless restrictions are imposed. E.g. in observational logics, signature morphisms are usually not allowed to introduce new observers [BHb, GM00], precisely in order to rescue the satisfaction condition. Moreover, in the (non-)institution of SB-Casl [Bau01, BZ00], the satisfaction condition fails for signature morphisms that introduce additional state components [Bau01]. It turns out that our construction cannot be recommended for the observational case, since it suppresses coinduction, while the semantics obtained for SB-Casl arguably provides the 'right' notion of semantic consequence.

In HasCasl, polymorphic types, operators, and axioms are semantically coded out by collections of instances. That is, the effect of a polymorphic type is essentially just its contribution to the syntactic type universe; a polymorphic operator is interpreted as a family of operators, one for each instantiation of its type arguments; and a polymorphic axiom is understood as a collection of axioms, indexed over all types in the classes named in the quantifiers.

2.15.1 Failures of the Satisfaction Condition

There are various features in modern specification languages that tend to cause the satisfaction condition (cf. Sect. 2.1) to fail; besides polymorphism, this includes observational satisfaction and

dynamic equations between programs in states-as-algebras frameworks such as SB-CASL [BZ00]. Briefly, the reasons for the failures are as follows:

- **Parametric polymorphism:** if a signature morphism σ introduces additional types, then the translation of a polymorphic axiom φ may fail in a model M although φ holds in the reduct of M along σ, namely if φ holds for the 'old' types, but not for the newly introduced ones.
- **Observational equality:** if a signature morphism σ introduces additional observers, then observational equalities that hold in the reduct of a model M under σ may fail in M, since the new observers may detect previously unobservable differences.
- **dynamic equations:** if a signature morphism σ introduces additional state components (i.e. dynamic functions, predicates, or sorts), then dynamic equations $p = q$ between stateful program expressions [BZ00] that hold in the reduct $M|_\sigma$ of a model M may fail to hold in M, since the interpretations of p and q may differ on the new state components [Bau01].

In all these cases, only one direction of the satisfaction condition holds, so that logics with these features constitute proper rps preinstitutions; we explicitly repeat the definition [SS93]:

Definition 2.39 A *preinstitution* consists of a signature category equipped with model and sentence functors and a satisfaction relation in the same sense as an institution (cf. Sect. 2.1); these data are not, however, required to obey the satisfaction condition. A preinstitution is called an *rps preinstitution* ('reducts preserve satisfaction') if

$$M \models \sigma\varphi \quad \text{implies} \quad M|_\sigma \models \varphi$$

for all M, σ, φ, and an *eps preinstitution* ('extensions preserve satisfaction') if the reverse implication holds.

Let PI_1, PI_2 be preinstitutions. A *preinstitution comorphism* [Mos96c] $\mu : PI_1 \to PI_2$ consists of the same data (Φ, α, β) as an institution comorphism (in particular, sentence translation is covariant and model translation is contravariant), without however being required to obey the satisfaction condition as in Definition 2.12. A preinstitution comorphism μ is called *rps* if

$$M \models \alpha\varphi \quad \text{implies} \quad \beta M \models \varphi,$$

and *weakly eps* if a model M satisfies $\alpha\varphi$ whenever $\beta K \models \sigma\varphi$ for all K, σ such that $K|_{\Phi\sigma} = M$.

Thus, an institution is a preinstitution that is simultaneously rps and eps, and a preinstitution comorphism between two institutions is an institution comorphism iff it is rps and weakly eps.

The typical remedy used hitherto to obtain institutions in the presence of the mentioned features is to restrict signature morphisms to cases where the full satisfaction condition holds. This is not an acceptable solution for the case of polymorphism: one has to require that signature morphisms do not introduce additional types, a restriction that effectively prevents the use of structured specifications. We emphasize that this problem is *not* solved by treating quantified types as first-class types (higher rank polymorphism), even if one manages to work around the obstacle that the latter is inconsistent with higher order logic [Coq86]: e.g., the restriction that signature morphisms be surjective on types is imposed also in [NP86], where it is needed in order to ensure preservation of coherent families of domains in a semantics of higher rank polymorphism in the style of Reynolds. In other words, ensuring coherence of polymorphic operators model-theoretically is not a feasible option.

For plain shallow polymorphism without type classes, a further alternative is to interpret the range of quantification over type variables in a fixed universe of types, i.e. some collection of sets closed under a number of constructions, rather than in the syntactical universe of declared types. This is the approach taken e.g. in [Bor00, KBS88]; it is not apparently suitable for HASCASL and similar frameworks for two reasons:

- in connection with a Henkin style semantics of function types, it is unclear what closure of the type universe under function types means;

- the type universe does not give an indication of what the interpretation of type classes should be, in particular since type classes on the one hand can be entirely loose and on the other hand are meant to contain only explicitly declared instances rather than, say, all structures matching the interface.

Independently of these specific issues, a further general disadvantage of the universe approach is that the choice of a universe unduly influences semantic consequence — the particularities of the chosen universe may induce unintended semantic consequences in a rather unpredictable way, thus introducing an unnecessary degree of incompleteness of deduction. The solution chosen in the semantics of HasCasl is therefore to add a second level to the model semantics according to the general construction described below.

2.15.2 Generic Polymorphism

We now introduce a general notion of syntactic polymorphism in an institution which covers HasCasl's type class polymorphism as a special case. This construction provides a wide range of examples of rps preinstitutions. We will return to this example in Sect. 2.15.4, where we will show that the notion of semantic consequence between polymorphic formulae induced by our generic construction of institutions from preinstitutions is not only in accordance with intuitive expectations, but also greatly simplifies the original notion.

Our construction of polymorphic formulae is similar in spirit to the *open formulae* introduced in [Tar86]: given a signature Σ_1, an open Σ_1-formula is just a sentence ϕ in some extension Σ_2 of Σ_1, and a Σ_1-model M satisfies such a formula if all its expansions to Σ_2 satisfy ϕ. In typical algebraic settings, this produces exactly the right kind of first or higher order quantification if Σ_2 introduces only additional constants or function symbols, respectively; essentially, the new symbols then play the role of universally quantified variables. However, the given notion of satisfaction is rather too strong if Σ_2 introduces additional types; since new sorts and function symbols involving new sorts (including instances of polymorphic operators for new sorts) can be interpreted with arbitrary malevolence in extensions of M, most open formulae involving such a Σ_2 will in fact be unsatisfiable.

Thus, we need a relaxed notion of satisfaction in order to arrive at the right notion of universal quantification over types. The idea is to require satisfaction of ϕ as above not for *all* extensions of M, but only for *extensions by syntactic definition*, i.e. the new signature items in Σ_2 have to be interpreted in terms of the base signature Σ_1. Of course, the involved notion of interpretation will have to be sufficiently general. E.g., we will want to interpret function symbols by terms, type constants by composite types etc. — in other words, we will need to use derived signature morphisms. All this is formalized as follows.

Definition 2.40 An *institution with signature variables* is an institution I with a distinguished object-full subcategory **Var** of the signature category **Sign** (i.e. **Var** need not be full in **Sign**, but contains all objects of **Sign**) whose morphisms are called *signature variables*. Signature variables are assumed to be *pushout-stable*, i.e. pushouts of signature variables along **Sign**-morphisms exist and are signature variables. (Morphisms in **Sign** should be thought of as derived signature morphisms.)

In I, a *polymorphic formula* $\forall\sigma.\,\phi$ over a signature Σ_1 consists of a signature variable $\sigma : \Sigma_1 \to \Sigma_2$ and a Σ_2-sentence ϕ. A Σ_1-model M *satisfies* $\forall\sigma.\,\phi$ if

$$M \models \tau\phi \qquad \text{for all } \tau \text{ in } \mathbf{Sign} \text{ such that } \tau \circ \sigma = id.$$

A sentence $\tau\phi$ as above is called an *instance* of $\forall\sigma.\,\phi$. The *translation* $\rho(\forall\sigma.\,\phi)$ of $\forall\sigma.\,\phi$ along a signature morphism $\rho : \Sigma_1 \to \Sigma_3$ is defined to be $\forall\bar{\sigma}.\,\bar{\rho}\phi$, where

$$\begin{array}{ccc} \Sigma_2 & \xrightarrow{\bar{\rho}} & \bullet \\ {\scriptstyle\sigma}\uparrow & & \uparrow{\scriptstyle\bar{\sigma}} \\ \Sigma_1 & \xrightarrow[\rho]{} & \Sigma_3 \end{array}$$

is a pushout; note that $\bar{\sigma}$ is indeed a signature variable. (This definition determines the translation only up to isomorphism; for similar reasons as given in Remark 5.1. of [Tar86], this is not actually a problem.)

The *polymorphic preinstitution* Poly(I) *over* I is given as follows: the notions of signature, model, and model reduction are inherited from I; Σ-sentences are polymorphic formulae over Σ; satisfaction and sentence translation are as above.

The sentences of I can be coded in Poly(I): a Σ-sentence ϕ in I is equivalent to the polymorphic formula $\forall id_\Sigma . \phi$, where id_Σ is indeed a signature variable thanks to object-fullness of **Var** in **Sign**.

By the above example and Sect. 2.15.1, it is clear that the polymorphic preinstitution Poly(I) will in general fail to be an institution. However, we have

Theorem 2.41 The polymorphic preinstitution Poly(I) is an rps preinstitution.

2.15.3 A Generic Institutionalization

We now describe a general process that transforms preinstitutions into institutions. We begin with a heuristic observation regarding the intended meaning of polymorphic definitions. Consider the specification

spec COMPOSITION =
vars $a, b, c : \mathit{Type}$
op $\mathit{comp} : (b \to c) \to (a \to b) \to a \to c$
vars $f : b \to c;\ g : a \to b$
• $\mathit{comp}\ f\ g = \lambda x : a\,.\ f\ (g\ x)$

where for the sake of the argument we abuse HASCASL as a notation for the simply typed λ-calculus with shallow polymorphism in much the same sense as described in Sect. 3.4, the only real point of this being the assumption that, unlike in actual HASCASL, there is no unit type. On the first level of the semantics as described in Sect. 3.4, COMPOSITION is model-theoretically entirely vacuous, since the syntactic set of types is empty and hence the polymorphic axiom is trivially satisfied in 'all' models of the signature (there is in fact only one model, since the signature is effectively empty). This is clearly not the intention of COMPOSITION. Indeed this specification is necessarily meant as a building block for other specifications that import the polymorphic operator and its definition, which then induce instances according to the ambient signature. In other words, the real purpose of COMPOSITION is apparently to say something about the interpretation of *comp* at *all* types, even those not yet declared. Thus, a model of the specification should contain information not only about the interpretation of the presently declared signature, but also about all 'future' extensions of this interpretation. This is the motivation for the following definitions:

Definition 2.42 Let PI be a preinstitution. An *extended model* of a signature Σ_1 is a pair (N, σ), where $\sigma : \Sigma_1 \to \Sigma_2$ is a signature morphism and N is a Σ_2-model in PI. The *reduct* $(N, \sigma)|_\tau$ of (N, σ) along a signature morphism τ is $(N, \sigma \circ \tau)$. The extended model (N, σ) *satisfies* a sentence φ if

$$N \models \sigma\varphi$$

in PI.

We record explicitly

Theorem and Definition 2.43 The extended models, together with the original notions of signature and sentence from PI, form an institution, called the *institution of extended models* and denoted Ext(PI).

PROOF: Functoriality of reduction is easy to see. To check the satisfaction condition, let $\tau : \Sigma_1 \to \Sigma_2$ be a signature morphism, let φ be a Σ_1-sentence, and let (N, σ) be an extended Σ_2-model. Then $(N, \sigma) \models \tau\varphi$ in Ext(PI) iff $N \models \sigma\tau\varphi$ in PI iff $(N, \sigma)|_\tau = (N, \sigma \circ \tau)$ satisfies φ in Ext(PI). □

The semantic consequence relation in Ext(PI) is precisely as expected:

Proposition 2.44 A Σ_1-sentence ψ is a semantic consequence of a set Φ of Σ_1-sentences in Ext(PI) iff

$$\sigma\Phi \models \sigma\psi$$

in PI for each signature morphism $\sigma : \Sigma_1 \to \Sigma_2$.

PROOF: *'If':* trivial.

'Only if': let $\sigma : \Sigma_1 \to \Sigma_2$ be a signature morphism, and let N be a Σ_2-model such that $N \models \sigma\Phi$ in PI. Then the extended model (N, σ) satisfies Φ and hence also ψ, i.e. we have $N \models \sigma\psi$. □

That is, a formula is a semantic consequence of a specification $Sp = (\Sigma, \Phi)$ (where Φ is a set of Σ-sentences) iff this is the case, in PI, in all extensions of Sp.

Example 2.45 In the example specification COMPOSITION from Sect. 2.15.1, all formulae are semantic consequences on the first level, i.e. in PI, since all formulae are vacuously true. This pathology disappears in Ext(PI), where semantic consequences of the specification are only those formulae that follow from the definition of composition independently of how many types are introduced, such as e.g. associativity of composition. Thus, the notion of semantic consequence at the second level, unlike the one at the first level, conforms to intuitive expectations. We will make this more precise in Sect. 2.15.4.

One can give a concise description of extensions in Ext(PI):

Lemma 2.46 The extensions of an extended model (N, τ) along a signature morphism σ are precisely the extended models (N, ρ) where $\tau = \rho \circ \sigma$.

We can represent PI in Ext(PI) by a preinstitution comorphism (cf. Definition 2.39)

$$\eta : PI \to \mathrm{Ext}(PI)$$

which is the identity on signatures and sentences, and takes every extended model to its base model.

Proposition 2.47 The comorphism η is weakly eps. Moreover, η is rps if PI is rps.

Remark 2.48 Interestingly, the concept of extended model is close to the *very abstract* or *hyper-loose semantics* as introduced in [CR98, Pep91], where models may interpret more symbols than just the ones named in their signature. This is used e.g. in the semantics of RSL [GHH+92].

There are two crucial differences here. The first is of motivational nature: the purpose of very abstract semantics is to ensure that refinement is model class inclusion; there is no intended connection with repairing the satisfaction condition, and in fact, the construction described in [CR98] is explicitly intended as a construction *on institutions* (one of the example applications given in [CR98, Pep91] is to the institution of many-sorted first order logic). Note that, when applied to institutions, the very abstract semantics is equivalent to the original semantics in terms of the engendered semantic consequence relation.

Secondly, at a more technical level, the phrase 'models may interpret additional symbols' means that very abstract semantics limits the notion of model to extended models with injective signature morphisms; the main technical content of [CR98] is to solve the difficulties caused by this restriction w.r.t. model reduction. For the purposes pursued here, the restriction to injective extensions is not only unnecessary, but would indeed invalidate our main result; i.e. for models of polymorphism modeled along the construction of [CR98], the satisfaction condition would still fail.

Taking PI as the first level of the HASCASL semantics (cf. Sect. 3.4), we define the *second level* of the semantics to be given by Ext(PI).

2.15.4 Semantic Consequence for Generic Polymorphism

We now investigate the implications of the extended model construction explained in Sect. 2.15.3 in relation to the generic polymorphism introduced in Sect. 2.15.2 — recall that generic polymorphism in general leads only to an rps preinstitution. For the remainder of this section, let I be an institution with signature variables, and let $\mathrm{Poly}(I)$ denote the polymorphic preinstitution over I as defined in Sect. 2.15.2.

Let $\forall\sigma.\,\phi$ and $\forall\rho.\,\psi$ be polymorphic formulae over a signature Σ_1. It is easy to check that $\forall\rho.\,\psi$ is a semantic consequence of $\forall\sigma.\,\phi$ in $\mathrm{Poly}(I)$ iff

$$\{\tau\phi \mid \tau \circ \sigma = id\} \models \pi\psi$$

in I for each signature morphism π such that $\pi \circ \rho = id$. This is rather unpleasant, since it means we have to prove a possibly infinite number of semantic consequences, one for each instance $\pi\psi$ of $\forall\rho.\,\psi$ in Σ_1. Fortunately, the (stronger) notion of semantic consequence in the institution $\mathrm{Ext}(\mathrm{Poly}(I))$ is much more tractable:

Theorem 2.49 In $\mathrm{Ext}(\mathrm{Poly}(I))$, $\forall\rho.\,\psi$ is a semantic consequence of $\forall\sigma.\,\phi$ iff

$$\rho(\forall\sigma.\,\phi) \models \psi$$

in $\mathrm{Poly}(I)$ (or, since ψ enjoys eps, equivalently in $\mathrm{Ext}(\mathrm{Poly}(I))$)

(Recall that $\rho(\forall\sigma.\,\phi) = \forall\bar\sigma.\,\bar\rho\phi$, where $(\bar\rho, \bar\sigma)$ is the pushout of (σ, ρ)). The above condition can be equivalently rephrased as the semantic consequence

$$\{\lambda\phi \mid \lambda \circ \sigma = \rho\} \models \psi \qquad (*)$$

in I. Thus, unlike proofs of semantic consequence in $\mathrm{Poly}(I)$ as described above, proofs in $\mathrm{Ext}(\mathrm{Poly}(I))$ are actually feasible, since we have to prove only a single generic instance of the goal, rather than all instances that exist in the base signature due to pure syntactic happenstance. Moreover,

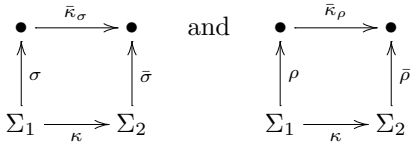
> *any sound and complete deduction system for I induces a sound and complete deduction system for* $\mathrm{Ext}(\mathrm{Poly}(I))$,

while for $\mathrm{Poly}(I)$, one will in general only obtain a sound but not complete deduction system.

The formulation of semantic consequence given in the theorem is exactly what one would intuitively expect: we fix the additional syntactic material quantified over by ρ and prove ψ only for this fixed instance; in the proof, we are allowed to make use of all instances of ϕ, including instances involving the new syntactic material. Proofs of polymorphic formulas e.g. in Isabelle [NPW02] work in precisely this way, which we have now provided with a semantic foundation.

Proof: [Theorem 2.49] *'Only If':* by Proposition 2.44, we have $\rho(\forall\sigma.\,\phi) \models \rho(\forall\rho.\,\psi)$, and ψ is an instance of $\rho(\forall\rho.\,\psi)$. The latter follows from the universal property of the pushout of ρ with itself.

'If': let Σ_1 be the base signature of $\forall\sigma.\,\phi$ and $\forall\rho.\,\psi$, let $\kappa : \Sigma_1 \to \Sigma_2$ be a signature morphism, and let



be the associated pushout diagrams. Then $\kappa(\forall\sigma.\,\phi) = \forall\bar\sigma.\,\bar\kappa_\sigma\phi$ and $\kappa(\forall\rho.\,\psi) = \forall\bar\rho.\,\bar\kappa_\rho\psi$. By Proposition 2.44, we thus have to prove

$$\forall\bar\sigma.\,\bar\kappa_\sigma\phi \models \forall\bar\rho.\,\bar\kappa_\rho\psi$$

in $\mathrm{Poly}(I)$, i.e. given a model M such that $M \models \forall\bar\sigma.\,\bar\kappa_\sigma\phi$ and τ such that $\tau\bar\rho = id$, we have to show $M \models \tau\bar\kappa_\rho\psi$ in I. Since semantic consequence in I is stable under translation, this reduces by $(*)$ above to showing $M \models \tau\bar\kappa_\rho\lambda\phi$ for all λ such that $\lambda \circ \sigma = \rho$. For such a λ, we have $\tau\bar\kappa_\rho\lambda\sigma = \tau\bar\kappa_\rho\rho = \tau\bar\rho\kappa = \kappa$, so that the pushout property yields ν such that $\nu\bar\sigma = id$ and $\nu\bar\kappa_\sigma = \tau\bar\kappa_\rho\lambda$. Then M satisfies the instance $\nu\bar\kappa_\sigma\phi$ of $\forall\bar\sigma.\,\bar\kappa_\sigma\phi$; but $\nu\bar\kappa_\sigma\phi = \tau\bar\kappa_\rho\lambda\phi$. □

2.15.5 Model-Theoretic Conservativity

While the semantic consequence relation engendered by the extended model construction is without further ado precisely the 'right' one, the issue of model expansion, i.e. of conservativity in the model-theoretic sense as used e.g. in CASL, is somewhat more subtle. We recall a few definitions:

Definition 2.50 A *theory* in a (pre-)institution is a pair $Sp = (\Sigma, \Phi)$ consisting of a signature Σ and a set Φ of Σ-sentences. A *model* of Sp is a Σ-model M such that $M \models \Phi$. A theory is *consistent* if it has a model. A signature morphism $\sigma : \Sigma_1 \to \Sigma_2$ is a *theory morphism* $(\Sigma_1, \Phi_1) \to (\Sigma_2, \Phi_2)$ if

$$\Phi_2 \models \sigma\Phi_1.$$

A theory morphism $\sigma : Sp_1 \to Sp_2$ is *model-theoretically conservative* or *model-expansive* if every model M of Sp_1 has an Sp_2-extension, i.e. a model N of Sp_2 such that $N|_\sigma = M$.

Notice that by Proposition 2.44 and Example 2.45, the notion of theory morphism in $\mathrm{Ext}(PI)$ is in general properly stronger than in PI.

Proposition 2.51 A theory is consistent in an rps preinstitution PI iff it is consistent in $\mathrm{Ext}(PI)$.

Typical extensions that would be expected to be model-expansive e.g. in HASCASL are (recursive) function definitions, loose declarations of new signature elements, and declarations of free datatypes. An apparent obstacle to model-expansivity of such extensions at the second level of the semantics is Part (i) of the following observation:

Proposition 2.52 Let PI be an rps preinstitution, and let $\sigma : (\Sigma_1, \Phi_1) \to (\Sigma_2, \Phi_2)$ be a theory morphism in $\mathrm{Ext}(PI)$. Then the following holds:

(i) If σ is model-expansive in $\mathrm{Ext}(PI)$ and (Σ_1, Φ_1) is consistent, then σ is a section as a signature morphism; i.e. there exists a signature morphism $\tau : \Sigma_2 \to \Sigma_1$ such that $\tau \circ \sigma = id$.

(ii) If σ is a section as a theory morphism in $\mathrm{Ext}(PI)$, i.e. there exists a theory morphism $\tau : (\Sigma_2, \Phi_2) \to (\Sigma_1, \Phi_1)$ such that $\tau \circ \sigma = id$, then σ is model-expansive.

PROOF: *(i):* By assumption and the rps condition, (Σ_1, Φ_1) has a model (M, id) in $\mathrm{Ext}(PI)$. By Lemma 2.46, existence of an extension of this model along σ implies that σ is a section.
(ii): Straightforward. □

When plain signature morphisms are used, which typically map type constants to type constants, operators to operators etc., then the necessary condition above is clearly too restrictive; essentially, the only model-expansive extensions one obtains are those that define symbols by other symbols already present. The solution to this is to use *derived* signature morphisms instead, which typically are allowed to map, say, type constants to composite types, operators to terms, and the like; by the sufficient condition (ii) above, one then obtains as model-expansive extensions all declarations and definitions which can be implemented by some composite object in the present theory.

With HASCASL derived signature morphism, model-expansive extensions behave as expected, see Sect. 3.4.3. In general, it depends on the expressive power of signatures and theories in the preinstitution at hand whether or not using derived signature morphisms leads to a satisfactory notion of model-expansivity. It should however be noted that there is usually quite some latitude in the definition of derived signature morphism; many forms of extensions can be made model-expansive by just giving a more liberal definition of what a derived signature morphism can do.

A drawback of the extended model construction is the general failure of even weak semi-exactness: two extended models can only be amalgamated in the rare case that there underlying models in PI are identical.

2.16 Bibliographical Notes

Institutions and institution morphisms have been introduced by Goguen and Burstall in a seminal paper [GB92], with [GB84] as a precursor. Meseguer [Mes89b] has extended institutions with entailment systems, arriving at the notion of logic. The notion of theory is not treated uniformly in the literature: while Goguen and Burstall require closedness under semantic consequence in [GB92], they do not require this elsewhere (see also [Mes89b]), such that theories closed under semantic consequence are called "closed theories". The treatment of institutions as functors is by Tarlecki, Goguen and Burstall [TBG91, GB92].

The term "institution comorphism" has been coined by Goguen and Rosu [RG04], but the notion has been introduced already under the names of plain map of institutions (by Meseguer [Mes89b]) and institution representation (by Tarlecki [Tar96]). Simple theoroidal comorphisms, introduced by Goguen and Rosu [RG04], have been called simple maps of institutions in [Mes89b]. The notion of subinstitution is due to Meseguer [Mes89b]. The adjunction between morphisms and comorphisms has been proved by Arrais and Fiadeiro [AF96] in the context of linear-time versus branching-time temporal logic. The model-expansion property of (co)morphisms has been studied for a long time. Weak amalgamation of comorphisms has been introduced in [Bor02], persistent liberality in [KM95, Dia98]. Bi-liberality is new.

Semi-morphisms have been introduced by Sannella and Tarlecki in the context of implementation [ST88d], and been dualized by Goguen and Rosu [RG04]. The first occurrence of the notion of forward (co)morphisms is in [Tar96], while the name has been introduced in [RG04].

Amalgamation resp. exactness of institutions has been introduced by Sannella and Tarlecki [ST88b] (see also [DGS91]), the extension to (co)morphisms is due to Borzyszkowski [Bor02].

The notion of institution morphism modification has been introduced by Diaconescu [Dia02]. Our motivation for using modifications is essentially the same as for using representation maps as introduced by Tarlecki [Tar96] (the latter are studied in Sect. 6.12). The results about colimits in Hom-categories are new.

The institutional treatment of polymorphism has been developed in [SML05]. It should not only provide a semantics for polymorphism in HasCasl, but also lead to a repairing of the satisfaction condition for SB-Casl [BZ00].

There is a comprehensive bibliography of the FLIRTS interest group (FLIRTS = *F*ormalism, *L*ogic, *I*nstitution — *R*elating, *T*ranslating, *S*tructuring) to be found at `http://www.tzi.de/flirts`.

Chapter 3

Some Institutions for Specification of Software Systems

> "My impression is that the need for different specification languages during the construction of complex system comes from the presence of features belonging to different domains, like, for instance, imperative, functional, non deterministic, temporal and real-time aspects. Most of those aspects are naturally described by languages (and logics) that *do not form an institution* in the obvious way, though they can be *(painfully) coded* to become an institution.
>
> Then, an institution based mechanism to compose heterogeneous specifications seems to me a fake, because the component specifications are not expressed in the "natural" language for the particular issue they are addressing, have to be already coded in the institution version of the "natural" language.
>
> I have the strong belief that the *heterogeneity* that can be naturally achieved within this kind of composition *is limited to different subsets of logical languages*."

This quote from an anonymous referee shows some typical reservations concerning the institutional approach to heterogeneous specification. In this chapter, we will demonstrate that a variety of features can be formalized in the institutional setting in a way enabling the use of heterogeneous specifications. Indeed, sometimes it is not so obvious how to formalize e.g. process algebras as institutions, and care has taken to choose a formalization that interacts well with the other institutions. However, in the end, we will see that heterogeneous specification is not at all limited to logical languages.

In the next section, we will start with introducing the institution underlying CASL, the Common Algebraic Specification Language. The other sections of this chapter will cover modal, coalgebraic, reactive and higher-order extensions of CASL. Note that this by no means implies that the heterogeneous framework introduced in Chap. 6 only applies to CASL and its extensions. However, since CASL itself is an extension of first-order logic, its extensions already cover a wide range of logics. This explains why we concentrate on CASL extensions in this work. But this does not preclude at all applications of the heterogeneous framework to completely different logics.

3.1 CASL

The most fundamental assumption underlying algebraic specification is that programs are modeled as algebraic structures that include a collection of sets of data values together with functions over those sets. This level of abstraction is commensurate with the view that the correctness of the input/output behaviour of a program takes precedence over all its other properties. Another common element is that specifications of programs consist mainly of logical *axioms*, usually in a logical system

in which equality has a prominent role, describing the properties that the functions are required to satisfy—often just by their interrelationship. This *property-oriented* approach is in contrast to so-called *model-oriented* specifications in frameworks like VDM [Jon90] which consist of a simple realization of the required behaviour. However, the theoretical basis of algebraic specification is largely in terms of constructions on algebraic models, so it is at the same time much more model-oriented than approaches such as those based on type theory (see e.g. [NPS90]), where the emphasis is almost entirely on syntax and formal systems of rules, and semantic models are absent or regarded as of secondary importance.

CASL, the Common Algebraic Specification Language, has been designed by CoFI, the international *Common Framework Initiative for algebraic specification and development* [Mos97, CoF], with the goal to subsume many previous algebraic specification languages and to provide a standard language for the specification and development of modular software systems. See the CASL user manual [BM04] and reference manual [CoF04] for detailed information about CASL.

Here, we concentrate on CASL *basic specifications*, designed for writing single specification modules. Structured specification will be covered in Chap. 5.

The logic of CASL basic specifications combines first-order logic and induction (the latter is expressed using so-called sort generation constraints, and needed for the specification of the usual inductive datatypes) with subsorts and partial functions. The institution underlying CASL is introduced in two steps [CoF99, CHKBM97]: first, we introduce many-sorted partial first-order logic with sort generation constraints and equality ($PCFOL^=$), and then, subsorted partial first-order logic with sort generation constraints and equality ($SubPCFOL^=$) is described in terms of $PCFOL^=$.

3.1.1 Partial First-Order Logic

Definition 3.1 The institution $PCFOL^=$.

Signatures A *many-sorted signature* $\Sigma = (S, TF, PF, P)$ in $PCFOL^=$ consists of

- a set S of *sorts*,
- two $S^*\times S$-sorted families $TF = (TF_{w,s})_{w\in S^*,s\in S}$ and $PF = (PF_{w,s})_{w\in S^*,s\in S}$ of *total function symbols* and *partial function symbols*, respectively, such that $TF_{w,s}\cap PF_{w,s} = \emptyset$, for each $(w, s) \in S^*\times S$ (constants are treated as functions with no arguments), and
- a family $P = (P_w)_{w\in S^*}$ of *predicate symbols*.

Note that function and predicate symbols may be overloaded, occurring in more than one of the above sets. To ensure that there is no ambiguity in sentences, however, symbols are always qualified by profiles when used. In the CASL language constructs (see Sect. 3.1.3), such qualifications may be omitted when these are unambiguously determined by the context.

We now introduce some notation. We write $f : w \to s \in TF$ for $f \in TF_{w,s}$ (including the special case $f : s$ for empty w), $f : w \to? s \in PF$ for $f \in PF_{w,s}$ (including the special case $f :\to? s$ for empty w) and $p : w \in P$ for $p \in P_w$. For a function symbol $f \in TF_{w,s}$ or $f \in PF_{w,s}$, we call $w \to s$ or $w \to? s$, resp., its *profile*, w its *argument sorts* and s its *result sort*. For predicate symbols $p : w$, we call w both its *profile* and its *argument sorts*. Given a function $f\colon A \longrightarrow B$, let $f^*\colon A^* \longrightarrow B^*$ be its extension to finite strings. Given a finite string $w = s_1 \ldots s_n$ and sets $M_{s_1}, \ldots, M_{s_n}$, we write M_w for the Cartesian product $M_{s_1} \times \ldots \times M_{s_n}$.

Given signatures $\Sigma = (S, TF, PF, P)$ and $\Sigma' = (S', TF', PF', P')$, a signature morphism $\sigma\colon \Sigma \longrightarrow \Sigma'$ consists of

- a map $\sigma^S\colon S \longrightarrow S'$,
- a map $\sigma^F_{w,s}\colon TF_{w,s} \cup PF_{w,s} \longrightarrow TF'_{\sigma^{S^*}(w),\sigma^S(s)} \cup PF'_{\sigma^{S^*}(w),\sigma^S(s)}$ preserving totality, for each $w \in S^*, s \in S$, and
- a map $\sigma^P_w\colon P_w \longrightarrow P'_{\sigma^{S^*}(w)}$ for each $w \in S^*$.

Identities and composition are defined in the obvious way. This gives us a category of $PCFOL^{=}$-signatures.

Models Given a many-sorted signature $\Sigma = (S, TF, PF, P)$, a *many-sorted Σ-model* M consists of:

- a non-empty carrier set M_s for each sort $s \in S$,
- a partial function $(f_{w,s})_M$ (also written just f_M) from M_w to M_s for each function symbol $f \in TF_{w,s} \cup PF_{w,s}$, the function being total if $f \in TF_{w,s}$, and
- a predicate $(p_w)_M$ (also written just p_M) $\subseteq M_w$ for each predicate symbol $p \in P_w$.

A *many-sorted Σ-homomorphism* $h : M \longrightarrow N$ consists of a family of functions $(h_s : M_s \longrightarrow N_s)_{s \in S}$ with the property that for all $f \in TF_{w,s} \cup PF_{w,s}$ and $(a_1, \ldots, a_n) \in M_w$ with $(f_{w,s})_M(a_1, \ldots, a_n)$ defined, we have

$$h_s((f_{w,s})_M(a_1, \ldots, a_n)) = (f_{w,s})_N(h_{s_1}(a_1), \ldots, h_{s_n}(a_n)),$$

and for all $p \in P_w$ and $(a_1, \ldots, a_n) \in M_w$,

$$(a_1, \ldots, a_n) \in (p_w)_M \text{ implies } (h_{s_1}(a_1), \ldots, h_{s_n}(a_n)) \in (p_w)_N.$$

Identities and composition are defined in the obvious way.

Concerning *reducts*, if $\sigma\colon \Sigma \longrightarrow \Sigma'$ is a signature morphism with $\Sigma = (S, TF, PF, P)$, and M' is a Σ'-model, then $M'|_\sigma$ is the Σ-model M with

- $M_s := M'_{\sigma^S(s)} \quad (s \in S)$
- $(f_{w,s})_M := (\sigma^F_{w,s}(f))_{M'} \quad (f \in TF_{w,s} \cup PF_{w,s})$
- $(p_w)_M := (\sigma^P_w(p))_{M'} \quad (p \in P_w)$

This is well-defined since $\sigma^F_{w,s}$ preserves totality.

Given a Σ'-homomorphism $h'\colon M'_1 \longrightarrow M'_2$, its reduct $h'|_\sigma\colon M'_1|_\sigma \longrightarrow M'_2|_\sigma$ is the homomorphism defined by

$$(h'|_\sigma)_s := h'_{\sigma^S(s)} \quad (s \in S).$$

It is easy to see that the reduct w.r.t. an identity is the identity, and that the reduct w.r.t. a composition is the composition of the reducts w.r.t. the signature morphisms that are composed. Thus, **Mod** is a functor.

Sentences Let a many-sorted signature $\Sigma = (S, TF, PF, P)$ be given. A *variable system over* Σ is an S-sorted, pairwise disjoint family of variables $X = (X_s)_{s \in S}$. Let such a variable system be given.

The sets $T_\Sigma(X)_s$ of *many-sorted Σ-terms* of sort s, $s \in S$, with variables in X are the least sets satisfying the following rules:

1. $x \in T_\Sigma(X)_s$, if $x \in X_s$,
2. $f_{w,s}(t_1, \ldots, t_n) \in T_\Sigma(X)_s$, if $t_i \in T_\Sigma(X)_{s_i} (i = 1, \ldots, n), f \in TF_{w,s} \cup PF_{w,s}, w = s_1 \ldots s_n$.

Note that each term has a unique sort.

$T_\Sigma(X)$ can be made into a many-sorted Σ-algebra by putting

- $(f_{w,s})_{T_\Sigma(X)}(t_1, \ldots, t_n) = f_{w,s}(t_1, \ldots, t_n)$ for $f \in TF_{w,s} \cup PF_{w,s}$,
- $(p_w)_{T_\Sigma(X)} = \emptyset$ for $p \in P_w$.

Variable assignments are total, but the value of a term w.r.t. a variable assignment may be undefined, due to the application of a partial function during the evaluation of the term. The evaluation of a term w.r.t. a variable assignment is defined as follows (see [CMR99]):

Given a variable valuation $\nu: X \longrightarrow M$ for X in M, the *term evaluation* $\nu^{\#}: T_\Sigma(X) \rightarrow ? M$ is inductively defined by:

- $\nu^{\#}_s(x) = \nu(x)$ for all $x \in X_s$ and all $s \in S$,
- $\nu^{\#}_s(f_{w,s}(t_1, \ldots, t_n)) = \begin{cases} (f_{w,s})_M(\nu^{\#}_{s_1}(t_1), \ldots, \nu^{\#}_{s_n}(t_n)), \\ \quad \text{if } \nu^{\#}_{s_i}(t_i) \text{ is defined } (i = 1, \ldots, n) \text{ and} \\ \quad\quad (f_{w,s})_M(\nu^{\#}_{s_1}(t_1), \ldots, \nu^{\#}_{s_n}(t_n)) \text{ is defined} \\ \text{undefined, otherwise} \end{cases}$

  for all $f \in TF_{w,s} \cup PF_{w,s}$, where $w = s_1 \ldots s_n$, and $t_i \in T_\Sigma(X)_{s_i}$, for $i = 1, \ldots, n$.

Given a term t, we say that t is *ν-interpretable*, if $t \in$ dom $\nu^{\#}$ and in this case we call $\nu^{\#}(t) \in M_s$ the *value of t in M under the valuation ν*.

A many-sorted atomic Σ-formula with variables in X is either (1) an application of a qualified predicate symbol to terms of appropriate sorts, (2) an existential equation between terms of the same sort, (3) a strong equation between terms of the same sort, or (4) an assertion about definedness of a term:

The set $AF_\Sigma(X)$ of *many-sorted atomic Σ-formulas* with variables in X is the least set satisfying the following rules:

1. $p_w(t_1, \ldots, t_n) \in AF_\Sigma(X)$, if $t_i \in T_\Sigma(X)_{s_i}, p \in P_w, w = s_1 \ldots s_n \in S^*$,
2. $t \stackrel{e}{=} t' \in AF_\Sigma(X)$, if $t, t' \in T_\Sigma(X)_s, s \in S$ (existential equations),
3. $t = t' \in AF_\Sigma(X)$, if $t, t' \in T_\Sigma(X)_s, s \in S$ (strong equations),
4. *def* $t \in AF_\Sigma(X)$, if $t \in T_\Sigma(X)_s, s \in S$ (definedness assertions).

The set $FO_\Sigma(X)$ of *many-sorted first-order Σ-formulas* with variables in X is the least set satisfying the following rules:

1. $AF_\Sigma(X) \subseteq FO_\Sigma(X)$,
2. *false* $\in FO_\Sigma(X)$
3. $(\varphi \wedge \psi) \in FO_\Sigma(X)$ and $(\varphi \Rightarrow \psi) \in FO_\Sigma(X)$ for $\varphi, \psi \in FO_\Sigma(X)$,
4. $(\forall x : s \bullet \varphi) \in FO_\Sigma(X)$ for $\varphi \in FO_\Sigma(X \cup \{x : s\})$, $s \in S$.

We omit brackets whenever this is unambiguous and use the usual abbreviations: $\neg\varphi$ for $\varphi \Rightarrow$ *false*, $\varphi \vee \psi$ for $\neg(\neg\varphi \wedge \neg\psi)$, *true* for $\neg$ *false* and $\exists x : s \bullet \varphi$ for $\neg\forall x : s \bullet \neg\varphi$.

A *sort generation constraint* states that some set of sorts is generated by some set of functions. Technically, sort generation constraints also contain a signature morphism component; this is needed to be able to translate them along signature morphisms without sacrificing the satisfaction condition.

Formally, a sort generation constraint over a signature Σ is a triple $(\overset{\bullet}{S}, \overset{\bullet}{F}, \theta)$, where $\theta: \bar{\Sigma} \longrightarrow \Sigma$, $\bar{\Sigma} = (\bar{S}, \bar{TF}, \bar{PF}, \bar{P})$, $\overset{\bullet}{S} \subseteq \bar{S}$ and $\overset{\bullet}{F} \subseteq \bar{TF} \cup \bar{PF}$.

Now a *Σ-sentence* is a closed many-sorted first-order Σ-formula (i.e. a many-sorted first-order Σ-formula in the empty set of variables), or a sort generation constraint over Σ.

Given a signature morphism $\sigma: \Sigma \longrightarrow \Sigma'$ and variable system X over Σ, we can get a variable system $\sigma(X)$ over Σ' by putting

$$\sigma(X)_{s'} = \bigcup_{\sigma^S(s) = s'} X_s$$

Since the term algebra is total, the inclusion $\zeta_{\sigma,X}: X \longrightarrow T_{\Sigma'}(\sigma(X))|_\sigma$ (construed as a variable valuation) leads to a term evaluation function

$$\zeta^{\#}_{\sigma,X}: T_\Sigma(X) \longrightarrow T_{\Sigma'}(\sigma(X))|_\sigma$$

that is total as well. This can be inductively extended to a translation of Σ-first order formulas along σ:

- $\sigma(t) = \zeta^{\#}_{\sigma,X}(t)$, if t is a Σ-term in variables X,
- $\sigma(p_w(t_1, \ldots, t_n)) = \sigma^P_w(p)_{\sigma^{S^*}(w)}(\sigma(t_1), \ldots, \sigma(t_n))$,
- $\sigma(t \stackrel{e}{=} t') = \sigma(t) \stackrel{e}{=} \sigma(t')$,
- $\sigma(t = t') = \sigma(t) = \sigma(t')$,
- $\sigma(\mathit{def}\ t) = \mathit{def}\ \sigma(t)$,
- $\sigma(\mathit{false}) = \mathit{false}$,
- $\sigma(\varphi \wedge \psi) = \sigma(\varphi) \wedge \sigma(\psi)$,
- $\sigma(\forall x : s \bullet \varphi) = \forall x : \sigma^S(s) \bullet \sigma(\varphi)$.

The translation of a Σ-constraint $(\dot{S}, \dot{F}, \theta)$ along σ is the Σ'-constraint $(\dot{S}, \dot{F}, \sigma \circ \theta)$.

It is easy to see that the sentence translation along the identity signature morphism is the identity, and that the sentence translation along a composition of two signature morphisms is is the composition of the sentence translations along the individual signature morphisms. Hence, sentence translation functorial.

Satisfaction Relation Even though the evaluation of a term w.r.t. a variable assignment may be undefined, the evaluation of a formula is always defined (and it is either true or false). That is, we have a two-valued logic. The application of a predicate symbol p to a sequence of argument terms holds w.r.t. a valuation $\nu: X \longrightarrow M$ iff the values of all the terms are defined under $\nu^\#$ and give a tuple belonging to p_M. A definedness assertion concerning a term holds iff the value of the term is defined. An existential equation holds iff the values of both terms are defined and identical, whereas a strong equation holds also when the values of both terms are undefined; thus both notions of equation coincide for defined terms.

More formally, satisfaction of a formula $\varphi \in FO_\Sigma(X)$ by a variable valuation $\nu: X \longrightarrow M$ is defined inductively over the structure of φ:

- $\nu \Vdash_\Sigma p_w(t_1, \ldots, t_n)$ iff $\nu^\#(t_i)$ is defined $(i = 1, \ldots, n)$ and, moreover, $(\nu^\#(t_1), \ldots, \nu^\#(t_n)) \in (p_w)_M$
- $\nu \Vdash_\Sigma t_1 \stackrel{e}{=} t_2$ iff $\nu^\#(t_1)$ and $\nu^\#(t_2)$ are both defined and equal,
- $\nu \Vdash_\Sigma t_1 = t_2$ iff $\nu^\#(t_1)$ and $\nu^\#(t_2)$ are either both defined and equal, or both undefined,
- $\nu \Vdash_\Sigma \mathit{def}\ t$ iff $\nu^\#(t)$ is defined,
- not $\nu \Vdash_\Sigma \mathit{false}$
- $\nu \Vdash_\Sigma (\varphi \wedge \psi)$ iff $\nu \Vdash_\Sigma \varphi$ and $\nu \Vdash_\Sigma \psi$
- $\nu \Vdash_\Sigma (\forall x : s \bullet \varphi)$ iff for all valuations $\xi: X \cup \{x : s\} \longrightarrow M$ which extend ν on $X \setminus \{x : s\}$ (i.e., $\xi(x) = \nu(x)$ for $x \in X \setminus \{x : s\}$), we have $\xi \Vdash_\Sigma \varphi$.

A formula φ is satisfied in a model M (written $M \models \varphi$) iff it is satisfied w.r.t. all variable valuations into M.

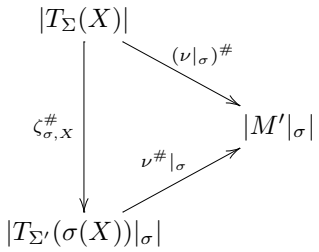
A Σ-constraint $(\dot{S}, \dot{F}, \theta)$ satisfied in a Σ-model M, if the carriers of $M|_\theta$ of the sorts in $\dot{S}$ are generated by the function symbols in $\dot{F}$, i.e. for every sort $s \in \dot{S}$ and every value $a \in (M|_\theta)_s$,

there is a $\bar{\Sigma}$-term t containing only function symbols from $\overset{\bullet}{F}$ and variables of sorts not in $\overset{\bullet}{S}$ such that $\nu^{\#}(t) = a$ for some assignment ν into $M|_\theta$.

For a sort generation constraint $(\overset{\bullet}{S}, \overset{\bullet}{F}, \theta)$ we can assume without loss of generality that all the result sorts of function symbols in $\overset{\bullet}{F}$ occur in $\overset{\bullet}{S}$. If not, we can just leave out from $\overset{\bullet}{F}$ those function symbols not satisfying this requirement. The satisfaction of the sort generation constraint in any model will not be affected by this: in the $\bar{\Sigma}$-term t witnessing the satisfaction of the constraint, any application of a function symbol with result sort outside $\overset{\bullet}{S}$ can just be replaced by a variable of that sort, which gets then as assigned value the evaluation of the function application.

Concerning the satisfaction condition, we need the following two lemmas:

Lemma 3.2 Let a signature morphism $\sigma: \Sigma \longrightarrow \Sigma'$, a Σ'-model M', and a variable system X over Σ be given. Then for each valuation $\nu: \sigma(X) \longrightarrow M'$, there is valuation $\nu|_\sigma: X \longrightarrow M'|_\sigma$ such that $\nu^{\#}|_\sigma \circ \zeta^{\#}_{\sigma,X} = (\nu|_\sigma)^{\#}$. Moreover, this is a one-one correspondence.



PROOF: We put

$$(\nu|_\sigma)_s(x) := \nu_{\sigma(s)}(x) \text{ for } x \in X_s.$$

The inverse is given by

$$((\_|_\sigma)^{-1}(\nu))_{s'}(x) := \nu_s(x),$$

where s is the unique $s \in S$ with $x \in X_s$ and $\sigma(s) = s'$.

The property $\nu^{\#}|_\sigma \circ \zeta^{\#}_{\sigma,X} = (\nu|_\sigma)^{\#}$ follows by induction over $T_\Sigma(X)$. □

Lemma 3.3 Given a signature morphism $\sigma: \Sigma \longrightarrow \Sigma'$, a Σ'-model M, a variable system X over Σ, and a formula $\varphi \in FO_\Sigma(X)$, we have

$$\nu|_\sigma \Vdash \varphi \text{ iff } \nu \Vdash \sigma(\varphi)$$

PROOF: Induction over φ.

E.g. for strong equations, we have $\nu|_\sigma \Vdash t_1 = t_2$
iff $(\nu|_\sigma)^{\#}(t_1) = (\nu|_\sigma)^{\#}(t_2)$ (or both sides are undefined)
iff (by Lemma 3.2) $\nu^{\#} \circ \zeta^{\#}_{\sigma,X}(t_1) = \nu^{\#} \circ \zeta^{\#}_{\sigma,X}(t_2)$ (or both sides are undefined)
iff (by definition) $\nu^{\#}(\sigma(t_1)) = \nu^{\#}(\sigma(t_2))$ (or both sides are undefined)
iff $\nu \Vdash \sigma(t_1 = t_2)$.

For quantifications, we have $\nu|_\sigma \Vdash \forall x : s \bullet \psi$ iff for all $\xi: X \cup \{x : s\} \longrightarrow M'|_\sigma$ extending $\nu|_\sigma$ on $X \setminus \{x : s\}$, $\xi \Vdash \psi$ iff (by induction hypothesis) for all $\xi: X \cup \{x : s\} \longrightarrow M'|_\sigma$ extending $\nu|_\sigma$ on $X \setminus \{x : s\}$, $(\_|_\sigma)^{-1}(\xi) \Vdash \sigma(\psi)$ iff (since $\_|_\sigma$ is one-one) for all $\rho: \sigma(X \cup \{x : s\}) \longrightarrow M'$ extending ν on $\sigma(X) \setminus \{x : \sigma(s)\}$, $\rho \Vdash \sigma(\psi)$ iff $\nu \Vdash \forall x : \sigma(s) \bullet \sigma(\psi)$.

The other cases are treated similarly. □

The satisfaction condition for first-order formulas now follows easily from Lemma 3.3 (noting that by Lemma 3.2 σ is surjective on valuations).

The satisfaction condition for sort generation constraints is obvious (indeed, the extra signature morphism component in the sort generation constraints has been introduced to make it work).

This completes the definition of the institution $PCFOL^=$. □

3.1.2 Subsorted Partial First-Order Logic

Subsorted partial first-order logic is defined in terms of partial first-order logic. The basic idea is to reduce subsorting to injections between sorts. While in the subsorted institution, these injections have to occur explicitly in the sentences, in the CASL language, they may be left implicit. Apart from the injections, one also has partial projection functions (one-sided inverses of the injections) and membership predicates.

Definition 3.4 The institution $SubPCFOL^=$.

Signatures The notion of subsorted signatures extends the notion of order-sorted signatures as given by Goguen and Meseguer [GM92], by allowing not only total function symbols, but also partial function symbols and predicate symbols:

A *subsorted signature* $\Sigma = (S, TF, PF, P, \leq_S)$ consists of a many-sorted signature (S, TF, PF, P) together with a reflexive transitive *subsort relation* $\leq_S$ on the set S of sorts. Note that $\leq_S$ is not required to be antisymmetric; this allows to declare isomorphic sorts, where the injections correspond to change of representations.

The relation $\leq_S$ extends pointwise to sequences of sorts. We drop the subscript S when obvious from the context.

For a subsorted signature, $\Sigma = (S, TF, PF, P, \leq_S)$, we define *overloading relations* (also called *monotonicity orderings*), $\sim_F$ and $\sim_P$, for function and predicate symbols, respectively:

Let $f : w_1 \longrightarrow s_1, f : w_2 \longrightarrow s_2 \in TF \cup PF$, then

$$f : w_1 \longrightarrow s_1 \sim_F f : w_2 \longrightarrow s_2$$

iff there exist $w \in S^*$ with $w \leq w_1$ and $w \leq w_2$ and $s \in S$ with $s_1 \leq s$ and $s_2 \leq s$.

Let $p : w_1, p : w_2 \in P$, then $p : w_1 \sim_P p : w_2$ iff there exists $w \in S^*$ with $w \leq w_1$ and $w \leq w_2$.

A *signature morphism* $\sigma : \Sigma \to \Sigma'$ is a many-sorted signature morphism that preserves the subsort relation and the overloading relations. Note that, due to preservation of subsorting, the preservation of the overloading relations can be simplified to:

$$f : w_1 \longrightarrow s_1 \sim_F f : w_2 \longrightarrow s_2 \text{ implies } \sigma^F_{w_1,s_1}(f) = \sigma^F_{w_2,s_2}(f)$$

$$p : w_1 \sim_P p : w_2 \text{ implies } \sigma^P_{w_1}(p) = \sigma^P_{w_2}(p)$$

With each subsorted signature $\Sigma = (S, TF, PF, P, \leq_S)$ we associate a many-sorted signature $\hat{\Sigma}$, which is the extension of the underlying many-sorted signature (S, TF, PF, P) with

- a total *injection* function symbol $\mathtt{inj} : s \to s'$, for each pair of sorts $s \leq_S s'$,
- a partial *projection* function symbol $\mathtt{pr} : s' \to ? s$, for each pair of sorts $s \leq_S s'$, and
- a unary *membership* predicate symbol $\in^s : s'$, for each pair of sorts $s \leq_S s'$.

We assume that the symbols used for injection, projection and membership are not used otherwise in Σ. In formulas, we also write $t \in s$ instead of $\in^s_{s'}(t)$ if s' is clear from the context.

Given a signature morphism $\sigma: \Sigma \longrightarrow \Sigma'$, we can extend it to a signature morphism $\hat{\sigma}: \hat{\Sigma} \longrightarrow \hat{\Sigma}'$ by just mapping the injections, projections and memberships in $\hat{\Sigma}$ to the corresponding injections, projections and memberships in $\hat{\Sigma}'$. This turns $\hat{\_}$ into a functor $\hat{\_}: \mathbf{Sign}^{SubPCFOL^=} \longrightarrow \mathbf{Sign}^{PCFOL^=}$.

Models Subsorted Σ-models are ordinary many-sorted $\hat{\Sigma}$-models satisfying the following set of axioms $\hat{J}(\Sigma)$ (where the variables are all universally quantified):

$\mathtt{inj}_{(s,s)}(x) \stackrel{e}{=} x$ (identity)

$\mathtt{inj}_{(s,s')}(x) \stackrel{e}{=} \mathtt{inj}_{(s,s')}(y) \Rightarrow x \stackrel{e}{=} y$ for $s \leq_S s'$ (embedding-injectivity)

$\mathtt{inj}_{(s',s'')}(\mathtt{inj}_{(s,s')}(x)) \stackrel{e}{=} \mathtt{inj}_{(s,s'')}(x)$ for $s \leq_S s' \leq_S s''$ (transitivity)

$\mathtt{pr}_{(s',s)}(\mathtt{inj}_{(s,s')}(x)) \stackrel{e}{=} x$ for $s \leq_S s'$ (projection)

$\mathtt{pr}_{(s',s)}(x) \stackrel{e}{=} \mathtt{pr}_{(s',s)}(y) \Rightarrow x \stackrel{e}{=} y$ for $s \leq_S s'$ (projection-injectivity)

$\in^s_{s'}(x) \Leftrightarrow def\ \mathtt{pr}_{(s',s)}(x)$ for $s \leq_S s'$ (membership)

$$\mathtt{inj}_{(s',s)}(f_{w',s'}(\mathtt{inj}_{(s_1,s'_1)}(x_1),\ldots,\mathtt{inj}_{(s_n,s'_n)}(x_n))) = \\ = \mathtt{inj}_{(s'',s)}(f_{w'',s''}(\mathtt{inj}_{(s_1,s''_1)}(x_1),\ldots,\mathtt{inj}_{(s_n,s''_n)}(x_n)))$$

for $f_{w',s'} \sim_F f_{w'',s''}$, where $w \leq w', w''$, $w = s_1 \ldots s_n$, $w' = s'_1 \ldots s'_n$, $w'' = s''_1 \ldots s''_n$ and $s', s'' \leq s$ (function-monotonicity)

$p_{w'}(\mathtt{inj}_{(s_1,s'_1)}(x_1),..,\mathtt{inj}_{(s_n,s'_n)}(x_n)) \Leftrightarrow p_{w''}(\mathtt{inj}_{(s_1,s''_1)}(x_1),..,\mathtt{inj}_{(s_n,s''_n)}(x_n))$

for $p_{w'} \sim_P p_{w''}$, where $w \leq w', w''$, $w = s_1 \ldots s_n$, $w' = s'_1 \ldots s'_n$, and $w'' = s''_1 \ldots s''_n$ (predicate-monotonicity)

Σ-homomorphisms are $\hat{\Sigma}$-homomorphisms.

Lemma 3.5 Let $\sigma: \Sigma \longrightarrow \Sigma'$ be a subsorted signature morphism. Then

$$\mathbf{Sen}^{PCFOL^=}(\hat{\sigma})(\hat{J}(\Sigma)) \subseteq \hat{J}(\Sigma')$$

PROOF: $\hat{\sigma}$ preserves subsorting, injections, projections, membership and the overloading relations $\sim_F$ and $\sim_P$. Now the sentences in $\hat{J}(\Sigma)$ and $\hat{J}(\Sigma')$ just correspond to these. □

To obtain a *reduct* of a subsorted Σ'-model M' along a subsorted signature morphism $\sigma: \Sigma \longrightarrow \Sigma'$, take the the many-sorted reduct $M'|_{\hat{\sigma}} = \mathbf{Mod}^{PCFOL^=}(\hat{\sigma})(M')$ of M' along $\hat{\sigma}: \hat{\Sigma} \longrightarrow \hat{\Sigma}'$. By definition of subsorted model, $M' \models^{PCFOL^=}_{\hat{\Sigma}'} \hat{J}(\Sigma')$. By the lemma,

$$M' \models^{PCFOL^=}_{\hat{\Sigma}'} \mathbf{Sen}^{PCFOL^=}(\hat{\sigma})(\hat{J}(\Sigma)).$$

From this, we get by the satisfaction condition for $PCFOL^=$

$$\mathbf{Mod}^{PCFOL^=}(\hat{\sigma})(M') \models^{PCFOL^=}_{\hat{\Sigma}} \hat{J}(\Sigma).$$

Thus, $\mathbf{Mod}^{PCFOL^=}(\hat{\sigma})(M')$ is a subsorted Σ-model, and hence, we can define $\mathbf{Mod}^{SubPCFOL^=}(\sigma)(M')$ to be $\mathbf{Mod}^{PCFOL^=}(\hat{\sigma})(M')$.

Sentences *Subsorted Σ-sentences* are ordinary many-sorted $\hat{\Sigma}$-sentences. Sentence translation along a subsorted signature morphism σ is just sentence translation along the many-sorted signature morphism $\hat{\sigma}$.

Satisfaction Since models and sentences are taken from $PCFOL^=$, satisfaction, as well as the satisfaction condition, can also be inherited from $PCFOL^=$.

This completes the definition of the institution $SubPCFOL^=$. □

```
%list [__], nil, __ :: __
%prec { __ :: __ } < { __ ++ __ }

spec LIST [sort Elem] =
  free type List[Elem] ::= nil | __ :: __(head :? Elem; tail :? List[Elem]);
  sort NEList[Elem] = {L : List[Elem] • ¬L = nil};
  op __ ++ __ : List[Elem] × List[Elem] → List[Elem];
  forall e : Elem; K, L : List[Elem]
      • nil ++ L = L                                   %(concat_nil)%
      • (e :: K) ++ L = e :: K ++ L                    %(concat_cons)%
end
```

Figure 3.1: Specification of lists over an arbitrary element sort in CASL.

3.1.3 CASL Language Constructs

Since the level of constructs will be treated only informally in this work, we just give a brief overview of the constructs for writing basic specifications (i.e. specifications in-the-small) in CASL. A detailed description can be found in the CASL Language Summary and the CASL semantics (see the CASL reference manual [CoF04]; the CASL user manual [BM04] provides a gentle introduction).

The CASL language provides constructs for writing sort, subsort, operation[1] and predicate declarations that contribute to the signature in the obvious way. Operations, predicates and subsorts can also be defined; this leads to a corresponding declaration plus a defining axiom.

Operation and predicate symbols may be overloaded; this may lead to ambiguities in the formulas. A formula is well-formed only if it has a unique fully-qualified expansion up to equivalence w.r.t. the overloading relations $\sim_F$ and $\sim_P$.

For operations and predicates, a mixfix syntax is provided. Precedence and associativity annotations may help to disambiguate terms containing mixfix symbols. There is also a syntax for literals such as numbers and strings, which allows the specification of the usual datatypes purely in CASL, without the need of magic built-in modules.

Binary operations can be declared to be associative, commutative, idempotent, or to have a unit. This leads to a corresponding axiom, and, in the case of associativity, to an associativity annotation.

The **type**, **free type** and **generated type** constructs allow the concise description of datatypes. They are expanded into the declaration of the corresponding constructor and selector operations and axioms relating the selectors and constructors. In the case of generated and free datatypes, also a sort generation constraint is produced. Free datatypes additionally lead to axioms that state the injectivity of the constructors and the disjointness of their images.

A typical CASL specification is shown in Fig. 3.1. Its translation to a presentation in $SubPCFOL^=$, the institution underlying CASL, is shown in Fig. 3.3. The translation has been generated with the Heterogeneous Tool Set (see Chap. 7) and uses the CASL notation to display theories in $SubPCFOL^=$. However, the notations are so close to each other that it should be easy to understand the specification in Fig. 3.3 as a theory in $SubPCFOL^=$. A CASL specification with loose types and mutually recursive free types is given in Fig. 3.2, the corresponding $SubPCFOL^=$-theories are shown in Fig. 3.4. The translation of CASL constructs to the underlying mathematical concepts is formally defined in the CASL semantics [CoF04]. An important concept for this translation is that of a *local environment* (which is given to the analysis of a specification): it is just the signature containing the previously declared items.

[1] At the level of constructs, functions are called operations.

spec CONTAINER [**sort** *Elem*] =
  type
    Container ::= *empty* | *insert*(*first* :? *Elem*; *rest* :? *Container*);

spec NTREE [**sort** *Elem*] =
  free types
    NTree ::= *fork*(*Elem*; *Forest*)
    Forest ::= *null* | *grow*(*NTree*; *Forest*)

Figure 3.2: Some **type** definition in CASL .

sorts *Elem*, *List*[*Elem*], *NEList*[*Elem*]
sort *NEList*[*Elem*] < *List*[*Elem*]
op _++_ : *List*[*Elem*] × *List*[*Elem*] → *List*[*Elem*]
op _::_ : *Elem* × *List*[*Elem*] → *List*[*Elem*]
op *head* : *List*[*Elem*] →? *Elem*
op *nil* : *List*[*Elem*]
op *tail* : *List*[*Elem*] →? *List*[*Elem*]
• ∀ *X1*: *Elem*; *X2*: *List*[*Elem*] • *head*(*X1* :: *X2*) = *X1* %(ga_selector_head)%
• ∀ *X1*: *Elem*; *X2*: *List*[*Elem*] • *tail*(*X1* :: *X2*) = *X2* %(ga_selector_tail)%
• ∀ *X1*: *Elem*; *X2*: *List*[*Elem*]; *Y1*: *Elem*; *Y2*: *List*[*Elem*]
  • *X1* :: *X2* = *Y1* :: *Y2* ⇔ *X1* = *Y1* ∧ *X2* = *Y2* %(ga_injective__::__)%
• ∀ *Y1*: *Elem*; *Y2*: *List*[*Elem*] • ¬ *nil* = *Y1* :: *Y2* %(ga_disjoint_nil__::__)%
• ¬ *def* *head*(*nil*) %(ga_selector_undef_head_nil)%
• ¬ *def* *tail*(*nil*) %(ga_selector_undef_tail_nil)%
generated {**sort** *List*[*Elem*]
  op __::__ : *Elem* × *List*[*Elem*] → *List*[*Elem*]
  op *nil* : *List*[*Elem*] %(ga_generated_List[Elem])%}
• ∀ *L*: *List*[*Elem*] • *L* ∈ *NEList*[*Elem*] ⇔ ¬ *L* = *nil*
• ∀ *L*: *List*[*Elem*] • *nil* ++ *L* = *L* %(concat_nil)%
• ∀ *e*: *Elem*; *K*, *L*: *List*[*Elem*] • (*e* :: *K*) ++ *L* = *e* :: (*K* ++ *L*) %(concat_cons)%

Figure 3.3: Translation of the specification LIST to a theory in $SubPCFOL^{=}$.

sorts *Container*, *Elem*
op *empty* : *Container*
op *first* : *Container* →? *Elem*
op *insert* : *Elem* × *Container* → *Container*
op *rest* : *Container* →? *Container*
• ∀ *X1*: *Elem*; *X2*: *Container* • *first*(*insert*(*X1*, *X2*)) = *X1* %(ga_selector_first)%
• ∀ *X1*: *Elem*; *X2*: *Container* • *rest*(*insert*(*X1*, *X2*)) = *X2* %(ga_selector_rest)%

sorts *Elem*, *Forest*, *NTree*
op *fork* : *Elem* × *Forest* → *NTree*
op *grow* : *NTree* × *Forest* → *Forest*
op *null* : *Forest*
• ∀ *X1*: *Elem*; *X2*: *Forest*; *Y1*: *Elem*; *Y2*: *Forest*
  • *fork*(*X1*, *X2*) = *fork*(*Y1*, *Y2*) ⇔ *X1* = *Y1* ∧ *X2* = *Y2* %(ga_injective_fork)%
• ∀ *X1*: *NTree*; *X2*: *Forest*; *Y1*: *NTree*; *Y2*: *Forest*
  • *grow*(*X1*, *X2*) = *grow*(*Y1*, *Y2*) ⇔ *X1* = *Y1* ∧ *X2* = *Y2* %(ga_injective_grow)%
• ∀ *Y1*: *NTree*; *Y2*: *Forest* • ¬ *null* = *grow*(*Y1*, *Y2*) %(ga_disjoint_null_grow)%
generated {**sorts** *Forest*, *NTree*
op *grow* : *NTree* × *Forest* → *Forest*
op *null* : *Forest*
op *fork* : *Elem* × *Forest* → *NTree* %(ga_generated_Forest_NTree)%}

Figure 3.4: Translation of the specifications CONTAINER and NTREE to theories in $SubPCFOL^{=}$.

3.1.4 Proof Calculus

We have developed a proof calculus for CASL in Chap. IV:2 of [CoF04]. We do not show it here, because rather than using this calculus, we will rely on an encoding of CASL into higher-order logic (Sect. 4.1) instead.

3.1.5 Checking Conservativity in CASL

CASL has annotations expressing that an extension of a specification is *conservative*, *monomorphic* or *definitional*, meaning that every model of the 'small' specification can be expanded to some model, some model unique up to isomorphism, or some unique model, of the 'large' specification, respectively (cf. Sect. 2.2). Moreover, as can be seen from the calculi studied in Sect. 5.3 and 5.6, checks for conservative extensions already arise from the presence of hiding. Furthermore, during the development process, it may be desirable to check the specification for *consistency* at an early stage – and consistency is just conservativity over the empty specification. Finally, using consistency, also *non-consequence* can be checked: an axiom does *not* follow from a specification if the specification augmented by the negation of the axiom is consistent.

So far there is no hope to tackle these questions in an institution independent way. Therefore, in this section we deal with the specific institution of CASL only. However, unfortunately already for first-order logic, neither the check for conservative, nor monomorphic, nor definitional extension are recursively enumerable, which means that there cannot be a complete (recursively axiomatized) calculus for them. For conservativity, this follows from Theorem 5.23 and the proof of Theorem 5.22: a recursively axiomatized calculus for conservativity would provide the needed oracle for Theorem 5.23, contradicting the example from the proof of Theorem 5.22.

Although there is no general approach to verify that an extension of a specification is conservative (or monomorphic, or definitional), several schemes for extending specifications have been developed in the past which guarantee these properties by construction. We only very informally list some possible rules here:

- extensions declaring new signature elements are conservative, provided the new symbols are not constrained in any way (by axioms, by requirements on the subsort and overloading relations, etc.) to be related to old symbols, and the new symbols themselves are constrained by a positive theory (i.e. not involving negation),
- free datatypes are monomorphic extensions of the local environment in which they are introduced,
- structured free Horn theories are monomorphic extensions,
- subsort definitions are definitional extensions, and
- inductive definitions over free datatypes are definitional extensions.

Some of these checks have been implemented in the Heterogeneous Tool Set.

3.1.6 Colimits of CASL signatures

Theorem 3.6 $PCFOL^=$ has a cocomplete signature category. (If restricted to finite signatures, $PCFOL^=$ has a finitely cocomplete signature category.)

PROOF: See [CGRW95]. □

The proof of cocompleteness of the CASL signature category is a bit more involved, due to the presence of the overloading relations.

As an example, consider the specification of $\mathbf{Sign}^{SubPCFOL^=}$ within CASL itself given in Fig. 3.5.

Subsorted partial-first order logic with sort generation constraints ($SubPCFOL$) is the underlying institution of CASL. It is described in [CoF97, CHKBM97]. For our cocompleteness proofs, we will need the following proposition, which follows from Corollaries 15 and 17 in [Mos96a].

spec ORDEREDSTRINGS =

sorts S, S^*

preds $\_ \leq \_ : \P S \times S$

$\_ \leq \_ : \P S^* \times S^*$

ops $\lambda : S^*$

$\_\ \_ : S \times S^* \longrightarrow S^*$

vars $s, s_1, s_2, s_3 : S; \quad w_1, w_2 : S^*$

axioms $s \leq s$

$s_1 \leq s_2 \wedge s_2 \leq s_3 \Rightarrow s_1 \leq s_3$

$\lambda \leq \lambda$

$w_1 \leq w_2 \wedge s_1 \leq s_2 \Leftrightarrow s_1 w_1 \leq s_2 w_2$

$s_1 w_1 = s_2 w_2 \Rightarrow (s_1 = s_2 \wedge w_1 = w_2)$

$\lambda = sw \Rightarrow (s_1 = s_2 \wedge w_1 = w_2)$

spec CASLSIG =

ORDEREDSTRINGS **then**

sorts $FunProfiles, PredProfiles$

ops $arity : FunProfiles \longrightarrow S^*$

$coarity : FunProfiles \longrightarrow S$

$arity : PredProfiles \longrightarrow S^*$

preds $istotal : FunProfiles$

$\_ \sim_F \_ : FunProfiles \times FunProfiles$

$\_ \sim_P \_ : PredProfiles \times PredProfiles$

vars $fp_1, fp_2 : FunProfiles; \quad pp_1, pp_2 : PredProfiles; \quad s : S; \quad w : S^*$

axioms $(fp_1 \sim_F fp_2 \wedge arity(fp_1) = arity(fp_2)$

$\wedge\ coarity(fp_1) = coarity(fp_2))$

$\Rightarrow fp_1 = fp_2$

$(pp_1 \sim_P pp_2 \wedge arity(pp_1) = arity(pp_2)) \Rightarrow fp_1 = fp_2$

$\lambda = sw \Rightarrow (fp_1 = fp_2 \wedge pp_1 = pp_2 \wedge istotal(fp_1))$

Figure 3.5: A specification of the CASL signature category within CASL

Proposition 3.7 Let $\theta: T \longrightarrow T1$ be a theory morphism in $SubPCFOL$ between universal Horn theories which both avoid the use of subsorting and do use strong equalities only in the conclusions of Horn formulae.[2]

Then both $\mathbf{Mod}(T)$ and $\mathbf{Mod}(T1)$ are cocomplete, and the forgetful functor $\_|_\theta: \mathbf{Mod}(T1) \longrightarrow \mathbf{Mod}(T)$ has a left adjoint.[3] □

Some Categorical Tools for Proving Cocompleteness

In this section, we recall some results from category theory that comprise a useful toolkit for proving cocompleteness theorems.

Definition 3.8 ([AHS90], 13.17) A functor $F: \underline{A} \longrightarrow \underline{B}$ is said to *lift colimits*, if for every diagram $D: \underline{I} \longrightarrow \underline{A}$ and every colimit $\mathcal{C}$ of $F \circ D$ there exists a colimit $\mathcal{C}'$ of D with $F(\mathcal{C}') = \mathcal{C}$. □

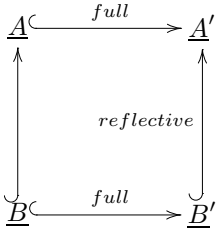
Definition 3.9 ([AHS90], 4.16) Let $\underline{A}$ be a subcategory of $\underline{B}$, and let B be a $\underline{B}$-object. An *$\underline{A}$-reflection arrow* for B is a morphism $r: B \longrightarrow A$ into an $\underline{A}$-object A having the following universal property:
For any morphism $f: B \longrightarrow A'$ from B into some $\underline{A}$-object A', there exists a unique $\underline{A}$-morphism $f': A \longrightarrow A'$ such that $f = f' \circ r$.

$\underline{A}$ is called a *reflective subcategory* of $\underline{B}$ provided that each $\underline{B}$-object has an $\underline{A}$-reflection.[4]

The dual notion is that of *co-reflective* subcategory. □

Proposition 3.10 ([Bor94], 3.5) If $\underline{A}$ is a full, reflective or coreflective, subcategory of $\underline{B}$, and $\underline{B}$ is cocomplete, then $\underline{A}$ is cocomplete as well. □

Proposition 3.11 Consider a commuting diagram of four subcategories



where $\underline{A}$ is a full subcategory of $\underline{A}'$, $\underline{B}$ is a full subcategory of $\underline{B}'$, and $\underline{B}'$ is a reflective subcategory of $\underline{A}'$.

If the $\underline{B}'$-reflection of an arbitrary $\underline{A}'$-object coming from $\underline{A}$ is already a $\underline{B}$-object, then $\underline{B}$ is a reflective subcategory of $\underline{A}$. □

Cocompleteness of the CASL Signature Category

We now come to the proof of cocompleteness of $\mathbf{Sign}^{SubPCFOL^=}$.

Consider the universal Horn CASL specification CASLSig given in Fig. 3.5. It might look a bit strange that we do not introduce a sort for names of functions and predicates, but only for profiles. We have done this because we want to capture CASL signature morphisms by CASLSig-homomorphisms. Now CASL signature morphisms can map different profiles for one and the same symbol name to profiles with different symbol names, while homomorphisms of course only have one choice for mapping elements of carriers. That is the reason why profiles are appropriate as carriers, and not symbol names. The identity of names can be recovered by using the overloading relations.

[2] The proposition can also be proved without the restriction that subsorting is not used; but forbidding strong equations in the premises of the Horn formulae is essential.

[3] For the purpose of this paper, we assume that empty carriers are allowed in the models of a CASL theory.

[4] Note that this is equivalent to the condition that the inclusion functor from $\underline{A}$ to $\underline{B}$ has a left adjoint, which then produces the reflection.

spec CASLSigGen =
CASLSig **then**
generated { **sort** S^*
ops $\lambda : S^*$
$\_\ \_ : S \times S^* \longrightarrow S^*$ }
vars $s : S;\ \ w : S^*$
axioms $\neg(\lambda = sw)$
$\neg(\lambda \leq sw)$
$\neg(sw \leq \lambda)$
vars $fp_1, fp_2 : FunProfiles;\ \ pp_1, pp_2 : PredProfiles$
axioms $fp_1 \sim_F fp_2 \Rightarrow \exists w : S^*;\ s : S \bullet$
$(w \leq arity(fp_1) \wedge w \leq arity(fp_2)$
$\wedge\ coarity(fp_1) \leq s \wedge coarity(fp_2) \leq s)$
$pp_1 \sim_P pp_2 \Rightarrow \exists w : S^* \bullet (w \leq arity(pp_1) \wedge w \leq arity(pp_2))$

Figure 3.6: A necessary refinement of CASLSig

The axioms starting with $\lambda = sw$ ensure that any confusion of elements in S^* enforces the model to be terminal.

By Prop. 3.7 we know that **Mod**(CASLSig), the model-category of CASLSig, is cocomplete. The intention is now that CASLSig specifies $\mathbf{Sign}^{SubPCFOL^=}$ somehow and thus $\mathbf{Sign}^{SubPCFOL^=}$ inherits cocompleteness.

Unfortunately, **Mod**(CASLSig) is not equivalent to $\mathbf{Sign}^{SubPCFOL^=}$: it is not guaranteed that there are no junk strings in the interpretation of S^*. Moreover, the profiles do not exactly capture function and predicate symbols. To repair this, we need the extension of CASLSig given in Fig. 3.6.

Lemma 3.12 **Mod**(CASLSigGen) is equivalent to $\mathbf{Sign}^{SubPCFOL^=}$.

Proof: A CASL-signature is mapped to a CASLSigGen-model by taking the obvious interpretations of the symbols. A CASLSigGen-model M is mapped to is the CASL-signature $(M_S, F, TF, P, \leq_M)$ which consists of

- the set of sorts M_S (without loss of generality, $(M_S)^*$ can be identified with M_{S^*}),
- the pre-order $\leq_M$,
- $FunNames = M_{FunProfiles}/(\sim_F)_M$,
- for each $w \in (M_S)^*$ and $s \in M_S$, $F_{w,s} = \{c \in FunNames \mid$ there is some $prof \in c$ with $arity_M(prof) = w, coarity_M(prof) = s\}$,
- for each $w \in (M_S)^*$ and $s \in M_S$, $c \in TF_{w,s}$ iff there is some $prof \in c$ with $arity_M(prof) = w, coarity_M(prof) = s$ and $istotal_M(prof)$,
- $PredNames = M_{PredProfiles}/(\sim_P)_M$,
- for each $w \in (M_S)^*$, $P_{w,s} = \{c \in PredNames \mid$ there is some $prof \in c$ with $arity_M(prof) = w\}$. □

To prove cocompleteness of **Mod**(CASLSigGen), we cannot apply Prop. 3.7, since CASLSigGen is not in Universal Horn form. But we can apply the following lemma:

Lemma 3.13 **Mod**(CASLSigGen) is a full coreflective subcategory of the category **Mod**(CASLSig).

Proof: The coreflection of a CASLSig-model M that is not terminal is the following submodel M' of M:

- $M'_S = M_S$,
- M'_{S^*} is the subset of M_{S^*} generated by λ_M and $\_\_\_M$,
- $\leq$ on M'_{S^*} is $\leq$ on M_{S^*} restricted to arguments of same length (otherwise, it yields false),
- $FunProfiles_{M'} = \{prof \in FunProfiles_M \mid arity_M(prof) \in M'_{S^*}\}$,
- $prof_1(\sim_F)_{M'} prof_2$ iff $prof_1(\sim_F)_M prof_2$ and there exist some $w \in M'_{S^*}$ and $s \in M'_S$ with $w \leq_M arity_{M'}(prof_1)$, $w \leq_M arity_{M'}(prof_2)$, $coarity_{M'}(prof_1) \leq_M s$ and $coarity_{M'}(prof_2) \leq_M s$,
- $PredProfiles_{M'} = \{prof \in PredProfiles_M \mid arity_M(prof) \in M'_{S^*}\}$,
- $prof_1(\sim_P)_{M'} prof_2$ iff $prof_1(\sim_P)_M prof_2$ and there exists some $w \in M'_{S^*}$ with $w \leq_M arity_{M'}(prof_1)$ and $w \leq_M arity_{M'}(prof_2)$,
- and the other operations and relations are inherited from M.

The coreflection property can be seen as follows: Consider a CASLSigGen-model $MGen$, a CASLSig-model M, and a CASLSig-homomorphism $\sigma: MGen \longrightarrow M$. Then σ is already a CASLSig-homomorphism into the coreflection of M, since σ preserves both the generatedness of elements of M_{S^*} and common super- and subsorts of arities and coarities, resp.

The coreflection of the terminal CASLSig-model is the terminal CASLSigGen-model consisting of one sort and, for each number of arguments, one total function and one predicate profile. □

We hence arrive at:

Theorem 3.14 *$SubPCFOL^=$ has a cocomplete signature category.*

Proof: Now cocompleteness of **Mod**(CASLSig) follows from Prop. 3.7, cocompleteness of **Mod**(CASLSigGen) with Lemma 3.13 and Prop. 3.10, and cocompleteness of $\mathbf{Sign}^{SubPCFOL^=}$ with Lemma 3.12. □

3.1.7 Amalgamation in Casl

Theorem 3.15 *$PCFOL^=$ admits amalgamation. (If restricted to finite signatures, $PCFOL^=$ admits finite amalgamation.)*

Proof: See [CGRW95]. □

Example 3.16 *$SubPCFOL^=$ fails to admit finite amalgamation, even to be semi-exact.*

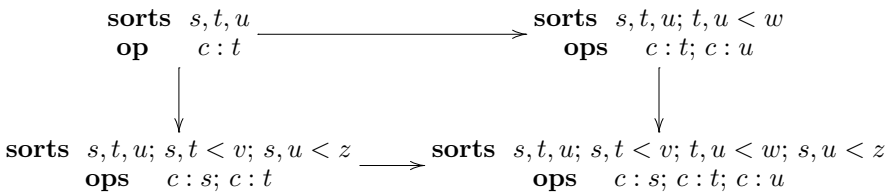
Let Σ be the signature with sorts s and t (and no operations), and let Σ_1 be the extension of Σ by the subsort relation $s < t$. Then the pushout

$$\begin{array}{ccc} \Sigma & \longrightarrow & \Sigma_1 \\ \downarrow & & \downarrow \\ \Sigma_1 & \longrightarrow & \Sigma_1 \end{array}$$

in $\mathbf{Sign}^{SubPCFOL^=}$ fails to be amalgamable (since two models of Σ_1, compatible w.r.t. the inclusion of Σ, may interpret the subsort injection differently).

Identifications of different paths of subsort injections is not the only source of failure of amalgamation in Casl.

Example 3.17 The pushout



fails to be amalgamable. The reason is that it is not guaranteed in the amalgamation that $c : s$ and $c : u$ are mapped to the same value of sort z. Though the overloading-equivalence between $c : s$ and $c : u$ can be traced back to a sequence of overloading-equivalences in the signatures of the diagram, the *specific way* of $c : s$ and $c : u$ being in the overloading relation in the pushout signature cannot be traced back.

3.1.8 Craig Interpolation in CASL

Theorem 3.18 *PFOL*$^{=}$ enjoys the Craig interpolation property for pushout squares where at least one of the signature morphisms of the diagram is injective on sorts.

PROOF: A proof of a slightly weaker result (with both signature morphisms of the diagram required to be injective on sorts) can be found in [Bor00]. The strengthening will be found in the forthcoming books [STa, Dia]. □

The counterexamples for amalgamation in CASL can easily be turned into counterexamples for Craig interpolation:

Example 3.19 *SubPFOL*$^{=}$ fails to have Craig interpolation even for injective signature morphisms.

Let Σ be the signature with sorts s and t and an operation symbol $f\colon s \longrightarrow t$, and let Σ_1 be the extension of Σ by the subsort relation $s < t$. Then the pushout

$$\begin{array}{ccc} \Sigma & \longrightarrow & \Sigma_1 \\ \downarrow & & \downarrow \\ \Sigma_1 & \longrightarrow & \Sigma_1 \end{array}$$

fails to admit Craig interpolation: for the entailment

$$\forall x : s\,.\ f_{s,t}(x) = \mathtt{inj}_{(s,t)}(x) \models_{\Sigma_1} \forall x : s\,.\ f_{s,t}(x) = \mathtt{inj}_{(s,t)}(x),$$

there is no Σ-interpolant.

It is not difficult to extend Example 3.17 to a further counterexample for Craig interpolation in *SubPFOL*.

Finally, also sort generation constraints can lead to failure of Craig interpolation:

Example 3.20 *PCFOL*$^{=}$ fails to have Craig interpolation even for injective signature morphisms.

Let Σ have a sorts Nat and s and operations $0 : Nat$ and $succ\colon Nat \longrightarrow Nat$. Let Σ_1 be Σ extended with an operation $c : s$, and Let Σ_1 be Σ extended with an operation $d : s$. If we take $\Sigma' = \Sigma_1 \cup \Sigma_2$, then clearly the diagram built with the obvious inclusions

$$\begin{array}{ccc} \Sigma & \xrightarrow{\sigma_1} & \Sigma_1 \\ \downarrow{\scriptstyle \sigma_2} & & \downarrow{\scriptstyle \theta_2} \\ \Sigma_2 & \xrightarrow{\theta_1} & \Sigma' \end{array}$$

is a pushout. Let φ_1 be the Σ_1-sentence $(\{Nat, s\}, \{0, succ, c\}, id)$ and φ_2 the Σ_2-sentence $(\{Nat, s\}, \{0, succ, d\}, id)$. Both sentences express that Nat is generated by 0 and $succ$ and that the sort s is interpreted as a singleton set. Hence

$$\theta_2(\varphi_1) \models \theta_1(\varphi_2).$$

Assume that φ is an interpolant, i.e. a Σ-sentence such that

$$\varphi_1 \models \sigma_1(\varphi) \text{ and } \sigma_2(\varphi) \models \varphi_2.$$

Since generatedness of Nat is not first-order expressible, φ cannot be a first-order sentence. Hence, it must be a sort generation constraint. However, a sort generation constraint over Σ cannot constrain sort s at all, and hence it is impossible that $\sigma_2(\varphi) \models \varphi_2$. We arrive at a contradiction. Thus, there is no interpolant.

Indeed, the above counterexample can be repaired if we allow a *set* if interpolants. In this case, set set would be

$$\{(\{Nat\}, \{0, succ\}, id), \forall x, y : s \,.\, x = y\}$$

We conjecture that $PCFOL^=$ restricted to sort-injective signature morphisms has Craig interpolation with a set of interpolants. The reason is that basically, the only ways in which two sort generation constraints over different signatures can interact are (1) via the shared signature symbols and (2) via sorts that are constrained to be singletons.

Another possibility of repairing the counterexample would be to close $PCFOL^=$ under conjunction. However, this adds the possibility of interaction between sort generation constraints and first-order formulas within the sentences to be interpolated, and we conjecture that this again leads to a failure of interpolation.

3.1.9 Subinstitutions of CASL

Despite being first-order, CASL is a quite rich and complex language. When relating CASL to other languages, it is quite useful to single out sublanguages of CASL. Here, we define a number of subinstitutions of the CASL institution $SubPCFOL^=$. The corresponding restriction of the CASL language is then straightforward, see [Mos97].

Below, we define subinstitutions of $SubPCFOL^=$ by just imposing restrictions on the signatures and/or axioms. This implicitly means that we take the full subcategory of signatures satisfying the restriction, and restrict the model and sentence functors and the satisfaction relation to this signature subcategory. A restriction on sentences further leads to the replacement of the sentence functor by a subfunctor. In each case, it is quite obvious to construct the corresponding subinstitution comorphism into $SubPCFOL^=$.

A Number of Features of CASL

In this section, we describe a number of CASL's features *negatively* by specifying, for each feature, the subinstitution of CASL that leaves out exactly that feature. This is possible since CASL is already the combination of all its features. A combination of only some of CASL's features can then be obtained by intersecting (in the sense of Sect. 2.8) all those subinstitutions that exclude exactly one of the undesired features. (Note that the combination of features from scratch is far more complicated [MTP98].)

Partiality The institution $SubCFOL^=$ is the restriction of the institution $SubPCFOL^=$ to those signatures with an empty set of partial function symbols and those sentences that do not involve partial projection symbols. Note that $SubCFOL^=$, like $SubPCFOL^=$, is still defined via a reduction to $PCFOL^=$, which involves signatures $\hat{\Sigma}$ containing partial projection symbols. However, these symbols are not used in the sentences, and they are redundant in the models (meaning that leaving them out leads to isomorphic model categories). Thus, it is justified to call $SubCFOL^=$ an institution of *total* algebras.

Predicates The institution $SubPCFOAlg^{=}$ is the restriction of $SubPCFOL^{=}$ to those signatures with an empty set of predicate symbols. Note that the signatures $\hat{\Sigma}$ and therefore the sentences in $SubPCFOAlg^{=}$ do involve membership predicate symbols. In the OBJ community, these are called "sort constraints". That is, $SubPCFOAlg^{=}$ includes subsorting with sort constraints.

Subsorting The institution $PCFOL^{=}$ has already been defined. It can be made into a subinstitution of $SubPCFOL^{=}$ by extending each signature with the trivial subsort relation (i.e., the subsort relation which is the identity relation on the set of sorts).

Sort Generation Constraints The institution $SubPFOL^{=}$ is the restriction of the institution $SubPCFOL^{=}$ to those sentences that are not sort generation constraints.

Equality The institution $SubPCFOL$ is the restriction of the institution $SubPCFOL^{=}$ to those sentences that involve neither strong nor existential equality.

A Number of Levels of Axiom Expressiveness

In the sequel, we introduce a number of subinstitutions of $SubPCFOL^{=}$ that correspond to different levels of expressiveness of the axioms. In contrast to the previous subsection, these are not orthogonal features, but rather we get a hierarchy of expressiveness.

First-order Logic This is given by $SubPCFOL^{=}$, which trivially is a subinstitution of itself.

Positive Conditional Logic Positive conditional logic more precisely means: universally quantified positive conditional logic. Usually this means that formulas are restricted to universally quantified implications that consist of a premise that is a conjunction of atoms, and a conclusion that is an atom:

$$\forall x_1 : s_1 \ldots \forall x_k : s_k \bullet \varphi_1 \wedge \ldots \wedge \varphi_m \Rightarrow \varphi$$

Positive conditional means that the φ_i must not implicitly contain negative parts. Usually, this condition is satisfied by atomic formulas. However, strong equations are implicit implications (*if* one side is defined, *then* so is the other, and they are equal), and the premise of an implication is a negative part of a formula. Hence, strong equations may not occur in the premises of positive conditional axioms (they are harmless in the conclusion, since the implicit premise can be thought of as an additional premise of the whole implication). The main motivation for this is that we want to use proof techniques such as conditional term rewriting and paramodulation [Pad88] and semantical constructions such as initial models. These work only if strong equations are not allowed in the premises (see [Cer93, AC95]).

Let $SubPCHorn^{=}$ be the restriction of $SubPCFOL^{=}$ to sentences of form

$$\forall x_1 : s_1 \ldots \forall x_k : s_k \bullet \varphi_1 \wedge \ldots \wedge \varphi_m \Rightarrow \varphi$$

where the φ_i and φ are atomic formulas such that none of the φ_i is a strong equation.

Generalized Positive Conditional Logic In the following, we generalize the above form of positive conditional formulas. Each formula of this more general kind is equivalent to a set of formulas of the standard conditional kind. Thus, there is an easy transformation from generalized positive conditional logic to plain positive conditional logic.

Within generalized positive conditional formulas, we also allow

- conjunctions of atoms in the conclusion (they can be removed by writing, for each conjunct, an implication with the original premise and the conjunct as conclusion), and
- equivalences instead of implications (an equivalence is equivalent to two implications).

Thus, let $SubPCGHorn^=$ be the restriction of $SubPCFOL^=$ to sentences of form

$$\forall x_1 : s_1 \ldots \forall x_k : s_k \bullet \varphi_1 \wedge \ldots \wedge \varphi_m \Rightarrow \psi_1 \wedge \ldots \wedge \psi_n$$

where the φ_i and ψ_j are atomic formulas such that none of the φ_i is a strong equation, or of form

$$\forall x_1 : s_1 \ldots \forall x_k : s_k \bullet \varphi_1 \wedge \ldots \wedge \varphi_m \Leftrightarrow \psi_1 \wedge \ldots \wedge \psi_n$$

where the φ_i and ψ_j are atomic formulas that are not strong equations.

Atomic Logic Let $SubPCAtom^=$ be the restriction of $SubPCFOL^=$ to sentences of form

$$\forall x_1 : s_1 \ldots \forall x_n : s_n \bullet \varphi$$

where φ is an atomic formula not being a strong equation.

This is the restriction of conditional logic to unconditional formulas. Strong equations are removed due to their conditional nature: in [Mos96a] it is proved that strong equations can simulate positive conditional formulas.

A Terminology for Naming CASL Subinstitutions

We now give a two-component name to the various subinstitutions that can be obtained by combining CASL's features. The first component is a vector of tokens. The presence (or absence) of a token denotes the presence (or absence) of a corresponding feature (cf. Sect. 3.1.9). The second component determines the level of expressiveness due to Sect. 3.1.9.

We assign the following tokens to the features:

- *Sub* stands for subsorting,
- *P* stands for partiality,
- *C* stands for sort generation constraints, and
- an equality symbol (=) stands for equality.

There is a naming problem with the predicate feature. Firstly, the letter P already stands for partiality. Secondly, *FOL* for first-order logic or *Horn* for Horn clause logic have become quite standard, but do not contain a token corresponding to predicates. Therefore, we deal with the predicate feature together with the levels of expressiveness from Sect. 3.1.9:

With predicates, we have the following endings:

- *FOL* stands for the unrestricted form of axioms (first-order logic),
- *GHorn* stands for the restriction to generalized positive conditional logic,
- *Horn* stands for the restriction to positive conditional logic,
- *Atom* stands for the restriction to atomic logic.

Without predicates, we have the following endings:

- *FOAlg* stands for the unrestricted form of axioms (first-order logic),
- *GCond* stands for the restriction to generalized positive conditional logic,
- *Cond* stands for the restriction to positive conditional logic,
- *Eq* stands for the restriction to atomic logic.

Any subset of the set of the four tokens *Sub*, *P*, *C* and =, followed by any of the eight above introduced endings now denotes the subinstitution obtained by intersecting

- the subinstitution of $SubPCFOL^{=}$ corresponding to the ending with
- the intersection of all the subinstitutions of $SubPCFOL^{=}$ associated to those letters *not* occurring in the set of tokens.

We finally adopt the convention that the equality sign = is always put at the end, as a superscript.

Some Interesting Subinstitutions of CASL

This section shall help to understand what the above naming scheme means in practice.

SubPCFOL$^{=}$ (read: subsorted partial constraint first-order logic with equality). This is the logic of CASL itself!

SubPFOL$^{=}$ (read: subsorted partial first-order logic with equality). CASL without sort generation constraints. This is described in [CHKBM97].

FOL$^{=}$ Standard many-sorted first-order logic with equality.

PFOL$^{=}$ Partial many-sorted first-order logic with equality.

FOAlg$^{=}$ First-order algebra (i.e., no predicates).

SubPHorn$^{=}$ This is the positive conditional fragment of CASL. It has two important properties:

1. Initial models and free extensions exist (see [Mos02]).
2. Using a suitable encoding of subsorting and partiality, one can use conditional term rewriting or paramodulation [Pad88] for theorem proving.

SubPCHorn$^{=}$ The positive conditional fragment plus sort generation constraints. Compared with $SubPHorn^{=}$, one has to add induction techniques to the theorem proving tools.

PCond$^{=}$ These are Burmeister's partial quasi-varieties [Bur86] modulo the fact the Burmeister does not have total function symbols. But total function symbols can be easily simulated by partial ones, using totality axioms, as in the partly total algebras of [BLR02]. A suitable restriction leads to Reichel's HEP-theories [Rei87]. Meseguer's Rewriting Logic [Mes92] can be embedded into $PCond^{=}$.

Horn$^{=}$ This is Eqlog [GM86, Pad88]. By further restricting this we get Membership Equational Logic $MEqtl$ [Mes98a], Equational Type Logic [MSS90] and Unified Algebras [Mos89]. Of course, Membership Equational Logic, Equational Type Logic and Unified Algebras are not just restrictions of $Horn^{=}$, but all have been invented in order to represent more complex logics within a subset of $Horn^{=}$.

Horn Logic Programming (Pure Prolog) [Llo87].

SubCond$^{=}$ Subsorted conditional logic. This is similar but not equal to OBJ [GW88]. See [Mos02] for a detailed comparison.

Cond$^{=}$ This is many-sorted conditional equational logic [TWW81] .

SubPAtom The atomic subset of CASL. Unconditional term rewriting becomes applicable.

SubPCAtom The atomic subset plus sort generation constraints.

Eq$^{=}$ This is the classical equational logic [GTW78].

CEq$^{=}$ Equational logic plus sort generation constraints.

In the literature, some of the above institutions are typically defined in a way allowing empty carrier sets, while CASL excludes empty carriers. This problem is discussed in [Mos02].

See Fig. 3.7 for a graph of CASL sublogics generated by the Heterogeneous Tool Set. Indeed, more important is the ability of HETS to detect the smallest sublogic of a given specification.

There are also some sublogics that do not fit in the scheme developed so far. For example, **Prop**, propositional logic, is the restriction of CASL to signatures that consist entirely of nullary predicates (these correspond to propositional variables).

3.1.10 Second-Order Logic

For the encoding of sort generation constraints, we will need the institution $SOL^=$ (second-order logic with equality), a *superinstitution* of $FOL^=$, which is strong enough to express sort generation constraints directly. Note that sort generation constraints cannot be expressed within first-order logic (since sort generation constraints can be used to specify the natural numbers up to isomorphism, this follows from Gödel's incompleteness theorem). Indeed, they cannot be expressed in HASCASL either, since HASCASL comes with a Henkin semantics (see Sect. 3.4).

The institution $SOL^=$ of second-order logic can be described as follows:

Signatures Signatures and signature morphisms are those of $FOL^=$.

Models Models, model homomorphisms and reducts are that of $FOL^=$.

Sentences Σ-sentences may contain variables that may be typed not only with sorts, but also with function types $w \longrightarrow s$ or predicate types $pred(w)$, where $w \in S^*$, $s \in S$. Quantification within sentences over variables of these higher types is also allowed. Variables of function or predicate types may be applied to arguments, like function and predicate symbols in $FOL^=$.

Satisfaction The satisfaction is defined much as in $FOL^=$, with the exception that valuations map variables of a function type to functions of that type, and variables of a predicate type to predicates of that type. □

3.2 Modal CASL

Modal logic was originally conceived as the logic of necessary and possible truths. Meanwhile, it is also used to express statements of knowledge, belief, provability, obligation and permissions, relations in time and space, and facts about behaviour of programs. From this diversity, it is clear that multi-modal logics will play a role. Moreover, there is a strong connection between Kripke models of multi-modal logics and labelled transition systems that are used for modeling reactive and concurrent systems. For the latter, often more expressive temporal logics like μ-calculus, ω-automata, CTL^*, CTL or LTL are needed. Here, we choose to extend modal logic with CTL^*, which is less expressive, but more readable than the μ-calculus, and still expressive enough for most applications.

MODALCASL extends CASL by modal logic, providing a multi-modal first-order logic with partiality and subsorting. Modalities can be either declared as fixed modalities, or a term-modal logic in the sense of [FM91, Tha00] can be used. For the latter, modalities are generated by terms, and some form of dynamic logic can be expressed as well. It is also possible to use both fixed modalities and term-modalities in parallel.

Specific modal logics can be obtained via restrictions to sublanguages.

3.2.1 Signatures

A MODALCASL signature consists of

- a CASL signature,

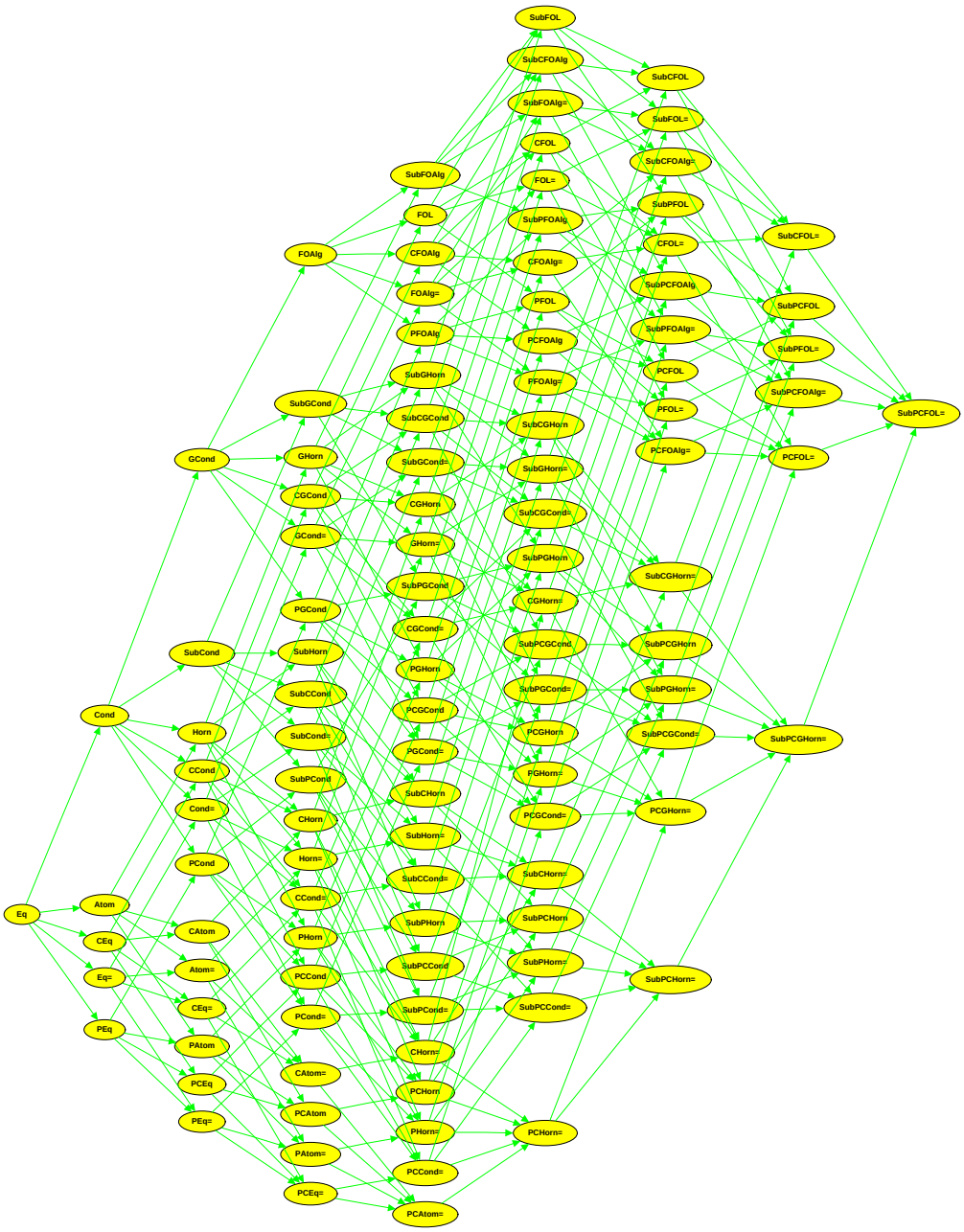


Figure 3.7: Graph of sublogics of CASL

- a predicate on the operation and predicate symbols of the CASL signature, marking some of them as *rigid* (the other ones are called flexible),
- a set of *modalities*, and
- a subset of the sort set of the CASL signature, called the *set of modality sorts*.

A MODALCASL signature morphism consists of

- a CASL signature morphism between the CASL signatures, and
- a mapping between the modality sets,

such that both rigidity of symbols as well as modality sorts are preserved.

3.2.2 Models

A MODALCASL model consists of

- a non-empty set of worlds W,
- for each world $w \in W$, a CASL model M_w,
- for each modality m, and for each carrier element m of each modality sort, a binary accessibility relation $\sim_m$ on W,

such that carrier sets and the interpretation of rigid operation and predicate symbols are the same for all M_w, and the associated accessibility relation is preserved under the embedding of a carrier element along a subsort injection.

The reduct of MODALCASL model against a signature morphism keeps the set of possible worlds. The accessibility relations are reduced against the renaming of modalities. The accessibility relations for the carrier elements of modality sorts are reduced along with the carrier elements (e.g. if a modality sort is not present in the source signature, they are just forgotten). The CASL models are reduced individually.

3.2.3 Sentences

MODALCASL sentences are closed state formulas built using the following grammar (S stand for state formulas, P for path formulas):

$$\text{S} ::= \text{At} \mid \mathit{false} \mid \text{S} \wedge \text{S} \mid \forall x : s \,.\, \text{S} \mid A\ \text{P} \mid [m]\ \text{S} \mid [m\text{-}]\ \text{S} \mid [m^*]\ \text{S} \mid [m\text{-}^*]\ \text{S}\ \text{P} ::= \text{S} \mid \mathit{false}$$
$$\mid \text{P} \wedge \text{P} \mid \forall x : s \,.\, \text{P} \mid X(m)\ \text{P} \mid X\text{-}(m)\ \text{P} \mid G\ \text{P} \mid G\text{-}\ \text{P} \mid \text{P}\ U\ \text{P} \mid \text{P}\ U\text{-}\ \text{P}$$

where m is a modality or a term of a modality sort (which can be omitted if m is *empty*) and At denotes a CASL atomic formula.

Informally,

- $[m]\varphi$ means that φ holds in all worlds that are 1-step-reachable through modality m from the present world,
- $[m\text{-}]\varphi$ means that φ holds in all worlds that are 1-reverse-step-reachable through modality m,
- $[m^*]\varphi$ means that φ holds in all worlds that are reachable by a finite number of steps with modality m,
- $[m\text{-}^*]\varphi$ means that φ holds in all worlds that are reachable by a finite number of reverse steps with modality m,
- $A\varphi$ means that φ holds on all paths starting in the current world,

- $X(m)\varphi$ holds if φ holds for the next time position of the path (and the transition to there is m),
- $X\text{-}(m)\varphi$ holds if φ holds for the previous time position of the path (and the transition to there is m),
- $G\,\varphi$ holds if φ holds for all future times on the path,
- $G\text{-}\,\varphi$ holds if φ held for all past times on the path,
- $\varphi\,U\,\psi$ holds if φ holds until ψ holds (or φ holds for all future times) on the path,
- $\varphi\,U\text{-}\,\psi$ holds if φ held since ψ held (or φ held at all past times) on the path.

We adopt the abbreviations of CASL (see Sect. 3.1.1), and furthermore, $\langle m\rangle\varphi$ abbreviates $\neg[m]\neg\varphi$, $\langle m*\rangle\varphi$ abbreviates $\neg[m*]\neg\varphi$, $E\varphi$ abbreviates $\neg A\neg\varphi$, $F\,\varphi$ abbreviates $\neg G\,\neg\varphi$, $\varphi\,W\,\psi$ abbreviates $\neg\psi\,U\,\varphi\wedge\psi$, and similarly for the past time operators (i.e. those marked with a "-"). m (together with the brackets) can be omitted if equal to *empty*.

Sentence translation along a signature morphism extends that of CASL in a straightforward manner.

3.2.4 Satisfaction

Satisfaction is based on the definition of satisfaction in CASL, while modalities are interpreted with Kripke semantics. Term evaluation is inherited from CASL, but now additionally parameterized by a world w (this is necessary due to the flexible operations). A state formula is interpreted relative to a model M, a valuation $\nu\colon X\longrightarrow M$ and a world w of the model:

- $\nu, w \Vdash_\Sigma p_{s_1\ldots s_n}(t_1,\ldots,t_n)$ iff $(\nu,w)^\#(t_i)$ is defined $(i=1,\ldots,n)$ and, moreover, $((\nu,w)^\#(t_1),\ \ldots\ ,(\nu,w)^\#(t_n))\in (p_{s_1\ldots s_n})_{M_w}$
- $\nu, w \Vdash_\Sigma t_1 \stackrel{e}{=} t_2$ iff $(\nu,w)^\#(t_1)$ and $(\nu,w)^\#(t_2)$ are both defined and equal,
- $\nu, w \Vdash_\Sigma t_1 = t_2$ iff $(\nu,w)^\#(t_1)$ and $(\nu,w)^\#(t_2)$ are either both defined and equal, or both undefined,
- $\nu, w \Vdash_\Sigma \mathit{def}\ t$ iff $(\nu,w)^\#(t)$ is defined,
- not $\nu, w \Vdash_\Sigma \mathit{false}$
- $\nu, w \Vdash_\Sigma (\varphi\wedge\psi)$ iff $\nu, w \Vdash_\Sigma \varphi$ and $\nu, w \Vdash_\Sigma \psi$
- $\nu, w \Vdash_\Sigma (\forall x : s\,.\,\varphi)$ iff for all valuations $\xi\colon X\cup\{x:s\}\longrightarrow M$ which extend ν on $X\setminus\{x:s\}$ (i.e., $\xi(x)=\nu(x)$ for $x\in X\setminus\{x:s\}$), we have $\xi, w \Vdash_\Sigma \varphi$.
- $\nu, w \Vdash_\Sigma A\varphi$ if for all paths π starting from w (i.e. $\pi_w(0)=w$), we have $\nu,\pi,0\Vdash_\Sigma\varphi$,
- $\nu, w \Vdash_\Sigma [m]\varphi$ if for all w' with $w\sim_{(\nu,w)^\#(m)} w'$, we have $\nu, w' \Vdash_\Sigma \varphi$,
- $\nu, w \Vdash_\Sigma [m\text{-}]\varphi$ if for all w' with $w'\sim_{(\nu,w)^\#(m)} w$, we have $\nu, w' \Vdash_\Sigma \varphi$,
- $\nu, w \Vdash_\Sigma [m^*]\varphi$ if for all w' with $w(\sim_{(\nu,w)^\#(m)})^* w'$, we have $\nu, w' \Vdash_\Sigma \varphi$,
- $\nu, w \Vdash_\Sigma [m\text{-}^*]\varphi$ if for all w' with $w'(\sim_{(\nu,w)^\#(m)})^* w$, we have $\nu, w' \Vdash_\Sigma \varphi$.

Here, $(\_)^*$ denotes reflexive transitive closure. Moreover, we set $(\nu,w)^\#(m)=m$ if m is just a modality (and not a term modality). A *path* $\pi=\langle\pi_w,\pi_m\rangle$ consists of a sequence π_w of worlds and a sequence π_m of modalities or modality sort carrier elements, both indexed by natural numbers, such that for all $t\in I\!N$, $\pi_w(t)\sim_{\pi_m(t)}\pi_w(t+1)$.

A path formula is interpreted relative to a model M, a valuation $\nu\colon X\longrightarrow M$ a path π and a natural number t marking a position ("time") on the path.

- $\nu, \pi, t \Vdash_\Sigma \varphi$ if $\nu, \pi_w(t) \Vdash_\Sigma \varphi$ (φ a state formula)
- not $\nu, \pi, t \Vdash_\Sigma \mathit{false}$
- $\nu, \pi, t \Vdash_\Sigma (\varphi \wedge \psi)$ iff $\nu, \pi, t \Vdash_\Sigma \varphi$ and $\nu, \pi, t \Vdash_\Sigma \psi$
- $\nu, \pi, t \Vdash_\Sigma (\forall x : s \,.\, \varphi)$ iff for all valuations $\xi\colon X \cup \{x : s\} \longrightarrow M$ which extend ν on $X \setminus \{x : s\}$ (i.e., $\xi(x) = \nu(x)$ for $x \in X \setminus \{x : s\}$), we have $\xi, \pi, t \Vdash_\Sigma \varphi$,
- $\nu, \pi, t \Vdash_\Sigma X(m)\,\varphi$ if $\pi_m(t) = (\nu, \pi_w(t))^\#(m)$ and $\nu, \pi, t+1 \Vdash_\Sigma \varphi$
- $\nu, \pi, t \Vdash_\Sigma X\text{-}(m)\,\varphi$ if $t = 0$ or $\pi_m(t-1) = (\nu, \pi_w(t))^\#(m)$ and $\nu, \pi, t-1 \Vdash_\Sigma \varphi$
- $\nu, \pi, t \Vdash_\Sigma G\,\varphi$ if for all $t' \geq t$, we have $\nu, \pi, t' \Vdash_\Sigma \varphi$
- $\nu, \pi, t \Vdash_\Sigma G\text{-}\,\varphi$ if for all $t' \leq t$, we have $\nu, \pi, t' \Vdash_\Sigma \varphi$
- $\nu, \pi, t \Vdash_\Sigma \varphi\, U\, \psi$ if either for all $t' \geq t$, we have $\nu, \pi, t' \Vdash_\Sigma \varphi$, or there is a $t' \geq t$ such that $\nu, \pi, t' \Vdash_\Sigma \psi$ and for all $t \leq u < t'$, we have $\nu, \pi, u \Vdash_\Sigma \varphi$,
- $\nu, \pi, t \Vdash_\Sigma \varphi\, U\text{-}\,\psi$ if either for all $t' \leq t$, we have $\nu, \pi, t' \Vdash_\Sigma \varphi$, or there is a $t' \leq t$ such that $\nu, \pi, t' \Vdash_\Sigma \psi$ and for all $t' < u \leq t$, we have $\nu, \pi, u \Vdash_\Sigma \varphi$.

A state formula φ is satisfied in a model M (written $M \models \varphi$) iff it is satisfied w.r.t. all variable valuations into M and all worlds $w \in W$.

The satisfaction condition is proved similarly to that of the CASL institution.

3.2.5 Sublanguages of MODALCASL

We single out a number of sublanguages of MODALCASL. All of them but the last one can also be qualified with a "Prop", which means the restriction to propositional logic (i.e. signatures are restricted to those with sorts and operation symbols, and only nullary predicate symbols are allowed).

Modal is first-order multi-modal logic. It excludes from full MODALCASL the path formulas and the modalities $[m\text{-}]$, $[m^*]$ and $[m\text{-}^*]$.

MonoModal is the restriction of Modal to a single modality *empty*.

CTL* removes all the past formulas and the modalities $[m]$ and $[m^*]$.

CTL is a restriction of CTL* where only path formulas involving temporal operators (X, G, U) are allowed.

LTL is a restriction of CTL* to those formulas containing exactly one A, and this occurs at the outermost position of the formula.

IndexedPropModal (indexed propositional modal logic) is a weak fragment of MonoModal that is still more expressive than PropMonoModal. Within one sentence, we require that for each argument position, the variable appearing in that position is the same for all predicate applications. This fragment has been chosen in such a way that for the purpose of performing proofs, we still can reduce it to propositional modal logic. Indeed, IndexedPropModal just formalizes the common practice to work with propositions in PropModal that are indexed by some (possibly infinite) set.

3.2.6 Amalgamation

ModalCasl fails to have the amalgamation property. This is because there is a part of the ModalCasl models that is unnamed in the signatures: namely, the set of worlds. More specifically, the initial (=empty) ModalCasl signature is not mapped to the terminal model category. Instead, each set of possible worlds leads to a different model of the empty signature.

One might try to repair this situation by dropping the set of worlds from models of signatures without modalities (which would mean that some reduct functors would drop the set of worlds as well). However, this would destroy the following pleasant property:

Theorem 3.21 ModalCasl has a cocomplete signature category and is semi-exact.

Proof: Cocompleteness of the signature category is proved similarly as for the Casl institution.

Let a pushout

$$\begin{array}{ccc} \Sigma & \xrightarrow{\sigma_1} & \Sigma_1 \\ \downarrow{\scriptstyle \sigma_2} & & \downarrow{\scriptstyle \theta_2} \\ \Sigma_2 & \xrightarrow{\theta_1} & \Sigma' \end{array}$$

a Σ_1-model M_1 and a Σ_2-model M_2 with $M_1|_{\sigma_1} = M_2|_{\sigma_2}$ be given. The latter implies that M_1 and M_2 have the same set W of possible worlds[5], which then must be also the set of possible worlds for the amalgamated model M'. For each world $w \in W$, M'_w is the amalgamation (as Casl-models) of $(M_1)_w$ and $(M_2)_w$. Finally, each modality in M' is interpreted (i.e. associated with a transition relation) in the way as its origin in Σ_1 is interpreted in M_1 (resp. its origins in Σ_2 is interpreted in M_2, where equality of the reducts $M_1|_{\sigma_1} = M_2|_{\sigma_2}$ ensures that this is well-defined in case it can be traced back to both Σ_1 and Σ_2). The interpretation of carrier elements for modality sorts as transition relation works similarly. □

3.3 CoCasl

In recent years, coalgebra has emerged as a convenient and suitably general way of specifying the reactive behaviour of systems [Rut00]. While algebraic specification deals with functional behaviour, typically using *inductive datatypes* generated by constructors, coalgebraic specification in concerned with reactive behaviour modeled by *coinductive process types* that are observable by selectors, much in the spirit of automata theory. An important role is played here by *final coalgebras*, which are complete sets of possibly infinite behaviours, such as streams or even the real numbers [PP99].

While Casl has been designed as a unifying standard for specification of inductive datatypes and functional requirements, for the much younger field of coalgebraic specification there is still a divergence of notions and notations. The design of CoCasl aims a fruitful synergy by extending Casl with coalgebraic constructs that dualize the algebraic constructs already present in Casl. Figure 3.8 contains a summary of dualizations of Casl concepts in CoCasl

At the level of basic specifications, the duality addresses the various forms of the **types** construct that serves to define inductive datatypes in Casl. In its elementary form, its dual is the **cotypes** construct, which serves to specify process types with observers (we shall reserve the word 'datatype' for the algebraic **types**). Recall that in Casl, a **type** declaration can be strengthened in two ways. In a **generated type**, junk is excluded, while a **free type** additionally forbids confusion. Dually, we introduce a **cogenerated cotypes** construct for fully abstract process types (Sect. 3.3.1), as well as a **cofree cotypes** construct, which additionally requires that all possible observable behaviours are realized in the process type; cf. Section 3.3.2. (Intercombinations such as **cofree types** etc. are not provided, and their emulation is expressly discouraged.) Moreover, we introduce a modal logic for axioms about state evolution in process types as syntactical sugar (Sect. 3.3.3).

[5]Here we need that the set of possible worlds is not forgotten when reducing to the empty signature.

Algebra	**Coalgebra**
type = (partial) algebra	**cotype** = coalgebra
constructor	observer (=selector)
generation	observability
generated type	**cogenerated cotype**
= no junk	= full abstractness
= induction principle	= coinduction principle
= pre-initiality	= pre-finality
no confusion	all possible behaviours
= inductive definition principle	= coinductive definition principle
= weak initiality	= weak finality
free type	**cofree cotype**
= absolutely initial datatype	= absolutely final process type
= no junk + no confusion	= full abstractness + all possible behaviours
free { ... }	**cofree { ... }**
= initial datatype (typically with equations)	= final process type (typically with modal axioms)

Figure 3.8: Summary of dualities between CASL and COCASL.

At the level of structured specifications, we dualize the structured **free** construct to a structured **cofree** construct which equips arbitrary specifications with a final semantics. See Appendix C for more information.

Let us now come to the level of COCASL basic specifications in more detail. CASL's **types** construct is complemented in COCASL by the **cotypes** construct. The syntax of this construct is nearly identical to the **type** construct; e.g., one may write

cotype $Process ::= cont(hd1 :?Elem;\ next :?Process)$
$\mid fork(hd2 :?Elem;\ left :?Process;\ right :?Process)$

thus determining constructors and selectors as for types.

However, for cotypes, the constructors are optional and the selectors (which we henceforth call observers) are mandatory. The latter requirement rules out CASL's sort alternatives making a given sort a subsort of the declared type, as in

type $Int ::= \mathbf{sort}\ Nat \mid -\_\_(Pos)$

Moreover, we also allow additional parameters for the observers. These have to come from the local environment (recall that the latter consists of all the declarations before the **cotype**):

spec MOORE =
sorts In, Out
cotype $State ::= (next : In \to State;\ observe : Out)$
end

The **cotype** definition in this case expands to

sort $State$
ops $next : In \times State \to State;$
$observe : State \to Out$

Observers with additional parameters do not have a corresponding constructor, since the constructor would need to have a higher-order type — e.g. in the above example $(In \to State) \to State$ — which is unavailable in CASL.

Last but not least, the **cotype** construct introduces a number of additional axioms concerning the domains of definition of the observers, besides the axioms relating constructors with their observers as for types:

- definedness of observers is independent of the additional parameters; the *domain* of an observer can thus be defined as a subset of the associated cotype,
- the domains of two observers in the same alternative are the same,
- the domains of two observers in different alternatives are disjoint, and
- the domains of all observers of a given sort are jointly exhaustive.

Thus, the alternatives in a cotype are to be understood as parts of a disjoint sum, so that cotypes, unlike types, correspond directly to coalgebras (see Proposition 3.23 below).

Definition 3.22 A cotype in CoCASL is given by the local environment sorts and the family of observers

$$CT = (S, (obs_{ijk}\colon T_i \longrightarrow T_{ijk})_{i=1\ldots n, j=1\ldots m_i, k=1\ldots r_{ij}}).$$

Here, S is a set of sorts (the local environment sorts, also called *observable sorts*), $T_1 \ldots T_n$ are the newly declared process types (or *non-observable sorts*) in the cotype (which possibly involve mutual recursion like in Figure 3.13), and obs_{ijk} is the k-th observer of the j-th alternative in the **cotype** definition of T_i. The result sort T_{ijk} of the observer may be either one of the T_i or one of the local environment sorts in S. Next, consider observers with additional parameters. In a **cotype** declaration, they are written $obs_{ijk} : s_1 \times \cdots \times s_m \to s$, where $s_1 \ldots s_m$ come from S and s either is one of the T_i or comes from S as well. In order to keep the format $obs_{ijk}\colon T_i \longrightarrow T_{ijk}$ for the type of the observer, the corresponding T_{ijk} is not simply a sort, but a function space

$$s_1 \times \cdots \times s_m \to s,$$

and the observer, normally having type $obs_{ijk} : s_1 \times \cdots \times s_m \times T_i \to s$, by currying can be equivalently considered to have the higher order type

$$obs_{ijk} : T_i \to (s_1 \times \cdots \times s_m \to s),$$

which is just $T_i \to T_{ijk}$. Although higher-order functions are not available in CoCASL, we prefer this notation for uniformity reasons. Still, the signature $Sig(CT)$ of a cotype CT is a first-order signature consisting of the local environment sorts S, the cotype sorts $T_1 \ldots T_n$, and the first-order profiles of the observers.

The *induced theory* of the cotype consists of the signature $Sig(CT)$ and the axioms generated by the cotype declaration as described above. The induced theory is also referred to as CT. An *S-palette* is an S-sorted family $C = (C_s)$ of sets of *colours*; a *C-colouring* is a a family h of maps $(h_s\colon A_s \longrightarrow C_s)_{s\in S}$. A CT-algebra A is called a *(CT, C)-algebra* if A interprets the non-observable sorts as prescribed by C, i.e. $A_s = C_s$ for all $s \in S$; a *homomorphism of (CT, C)-algebras* is a CT-algebra homomorphism that acts as the identity on the sorts in S.

Note that within cotypes, also constructors may be declared. However, we ignore them here, since they do not contribute to the coalgebra structure. However, they *do* play a role when homomorphisms are concerned, which is why we exclude them in the next proposition:

Proposition 3.23 To a given CoCASL **cotype** definition without constructors with induced theory CT and set S of observable sorts, one can associate a functor $F\colon \mathbf{Set}^n \longrightarrow \mathbf{Set}^n$ such that, for each S-palette C, the category of (CT, C)-algebras is isomorphic to the category of F-coalgebras. In particular, this implies that all homomorphisms between (CT, C)-algebras are closed.

```
spec STREAM1 [sort Elem] =
  cogenerated cotype
      Stream ::= cons(hd : Elem; tl : Stream)
end
```

Figure 3.9: Cogenerated specification of bit streams in CoCasl

.

PROOF: We begin with the parameterless case, without any local environment, i.e. we have a cotype

$$CT = (\emptyset, (obs_{ijk}: T_i \longrightarrow T_{ijk})_{i=1\ldots n, j=1\ldots m_i, k=1\ldots r_{ij}}).$$

By abuse of notation, we treat the T_i as set variables in the definition of the functor $F: \mathbf{Set}^n \longrightarrow \mathbf{Set}^n$:

$$F(T_1, \ldots, T_n) = (\coprod_j \prod_k T_{1,j,k}, \ldots, \coprod_j \prod_k T_{n,j,k}),$$

We now have to prove the stated isomorphism of categories. The axioms in Γ ensure that each T_i is the disjoint sum of sets X_{ij}, where X_{ij} is the domain of definition of the observers in the j-th alternative of the cotype declaration for T_i. Thus, we can regard CT-algebras as coalgebras

$$(T_1, \ldots, T_n) \xrightarrow{(g_1, \ldots, g_n)} F(T_1, \ldots, T_n)$$

on $\mathbf{Set}^n$ by taking g_i to be defined on X_{ij} by $g_i(x) = (obs_{ij1}(x), \ldots, obs_{ijr_{ij}}(x))$. It is easy to reverse this process: given an F-coalgebra A, the tuple $\langle obs, \ldots, obs_{ijr_{ij}} \rangle$ of observers in the j-th alternative for T_i is defined as the restriction of the i-th component of the structure map of A to the preimage of the j-th summand $\prod_k T_{ijk}$ of the i-th component of $F(T_1, \ldots, T_n)$. Altogether, this leads to a bijective correspondence between CT-algebras and F-coalgebras. Furthermore, this correspondence is functorial, i.e. a tuple $h = (h_1, \ldots, h_n): M \longrightarrow N$ of maps is a homomorphism of CT-algebras iff it is a homomorphism of the corresponding F-coalgebras. This equivalence is due to the fact that homomorphisms of partial algebras preserve definedness and hence respect the disjoint decompositions of the T_i, so that h_i can be decomposed into m_i maps between the disjoint summands of T_i. It is straightforward to generalize these arguments to the case that S is non-empty. □

3.3.1 Generation and cogeneration constraints

Dually to Casl's sort generation constraints, CoCasl introduces *cogeneratedness constraints* that amount to an implicit coinduction axiom and thus restrict the models of the type to fully abstract ones. This means that equality is the largest congruence w.r.t. the introduced sorts, operations and predicates (excluding the constructors). Put differently, everything that cannot be distinguished by its behaviour, as determined by the observers and the predicates, is identified (where observations can only be made on sorts in the local environment, i.e. outside the type declaration itself). In the example in Figure 3.9, the STREAM-models are (up to isomorphism) the subsets of E^ω that are closed under tl, where E is the interpretation of the sort *Elem*. (Note: since there is only one alternative, there is no difference between a type and a cotype here.)

A more complex example is the specification of CCS and CSP, see Fig. 1.6 and [MSRR]. States are generated by the CCS syntax, but they are identified if they are bisimilar w.r.t. the ternary transition relation. This can be expressed in CoCasl by stating that states are cogenerated w.r.t. the transition relation.

```
spec STREAM2 [sort Elem] =
  cofree cotype
      Stream ::= (hd : Elem; tl : Stream)
end
```

Figure 3.10: Cofree specification of bit streams in CoCASL.

Given a signature $\Sigma = (S, TF, PF, P, \leq)$, a *cogeneration constraint* over a signature is a subsignature fragment (i.e. a tuple of subsets of the respective signature components, which need not by itself form a complete signature) $\bar{\Sigma} = (\bar{S}, \bar{TF}, \bar{PF}, \bar{P})$ of Σ. In the above example, the cogeneration constraint is $(\{Elem\}, \{hd, tl\}, \emptyset, \emptyset)$. The constraint $\bar{\Sigma}$ is *satisfied* in a Σ-model M if each $(\bar{S}, (S, \bar{TF}, \bar{PF}, \bar{P}))$-bisimulation on M is the equality relation.

In duality to generated types in CASL, the construct **cogenerated cotype ...** abbreviates **cogenerated {cotype ...}**. No such abbreviation is provided for **cogenerated {type ...}**, the use of which is in fact expressly discouraged (as are **generated {cotypes ...}**).

Remark 3.24 Note that observers of cotypes always have exactly one non-observable argument. However, like the **generated { ... }** construct in CASL, the **cogenerated { ... }** construct allows the inclusion of arbitrary signature items in the cogeneratedness constraint, so that observers of arbitrary arity are also possible. In particular, full abstractness for binary observers in the sense of [Tew02] (i.e. observers with two non-observable argument sorts) is expressible.

Remark 3.25 At the level of model homomorphisms, the duality between generatedness and cogeneratedness constraints becomes formally a lot clearer: a generatedness constraint essentially amounts to a weakened form of initiality in the sense that a model M of the corresponding specification is *pre-initial* in the fibre over its reduct to the local environment (cf. Definition C.12 below) — i.e. there is at most one morphism from M into any other model over the same reduct. Dually, a model M that satisfies a cogeneratedness constraint is *pre-final* in its fibre in the sense that there exists at most one morphism from any other model over the same reduct into M. This may also roughly be expressed as follows: generated models do not have proper substructures, and cogenerated models do not have proper quotients.

3.3.2 Free types and cofree cotypes

Dually to CASL's **free types** construct, in CoCASL we provide a **cofree cotypes** construct that specifies the absolutely final coalgebra of infinite behaviour trees (see Example C.14 on why there is no **cofree types** construct). More concretely, this means that, in addition to cogeneratedness, there is also a principle stating that there are enough behaviours, namely all infinite trees [AM82] (with branching as specified by the observers). In contrast to its dual (no confusion among constructors), the latter principle cannot be expressed in first-order logic; however, a second-order specification is possible (see below). In the example in Figure 3.10, the STREAM2-models are isomorphic to E^ω, where E is the interpretation of the sort *Elem*. An example with an extra parameter for the observer is the specification of function types in Figure 3.11 (actually, this shows that higher-order types can be easily encoded in CoCASL). Similarly, Figure 3.12 specifies the final Moore automaton. Finally, in Figure 3.13 we use mutually recursive cofree cotypes to specify trees of infinite depth and branching, dualizing the *Ntree* example of Figure 3.2.

We are now ready to dualize the important algebraic concept of term algebra.

Definition 3.26 Given a cotype

$$CT = (S, (obs_{ijk} : T_i \longrightarrow T_{ijk})_{i=1...n, j=1...m_i, k=1...r_{i,j}})$$

```
spec FunctionType =
  sorts  A, B
  cofree cotype
        Fun[A, B] ::= (eval  :  A → B)
end
```

Figure 3.11: Cofree specification of function types.

```
spec FinalMoore1 =
  sorts  In, Out
  cofree cotype  State ::= (next : In → State;  observe : Out)
end
```

Figure 3.12: Cofree specification of the final Moore automaton.

```
spec InfTree [sort  Elem] =
  cofree cotypes
      InfTree    ::=  (label : Elem;  children : InfForest)
      InfForest  ::=  (first : InfTree;  rest : InfForest)
end
```

Figure 3.13: Cofree specification of trees of possibly infinite depth and branching.

and an S-palette C, the *behaviour algebra* $Beh_{CT}(C)$ is defined to be the following (CT, C)-algebra:

- the carriers for observable sorts (i.e. in S) are those determined by C;
- the carriers for a non-observable sort T_{i_0} consist of all infinite trees of the following form:
  - each inner node is labelled with a pair (T_i, j), where T_i is a non-observable sort and $j \in \{1, \ldots, m_i\}$ selects an alternative out of those for T_i;
  - the root is labelled with (T_{i_0}, j_0) for some j_0;
  - each leaf is labelled with an observable sort $s \in S$ and some colour from C_s;
  - each inner node with label (T_i, j) has one child for each of the observers obs_{ijk} ($k = 1 \ldots r_{ij}$) and each tuple of colours for the extra parameters of the observer. The child node is labelled with the result sort of the observer.
- an observer operation $obs_{i_0,j,k}$ is defined for a tree with root (T_{i_0}, j_0) if and only if $j = j_0$, and in this case, it just selects the child tree corresponding to the observer and the argument colours for the extra parameters of the observer.

Proposition 3.27 Given a cotype

$$CT = (S, (obs_{ijk} : T_i \longrightarrow T_{ijk})_{i=1\ldots n, j=1\ldots m_i, k=1\ldots r_{ij}})$$

and an S-palette C, the *behaviour algebra* $Beh_{CT}(C)$ is final in the category of (CT, C)-algebras (note that the latter correspond to coalgebras).

Proof: Using the characterization of Proposition 3.23, the result follows from the general construction of final coalgebras for polynomial functors over the category of $\{T_1, \ldots, T_n\}$-sorted sets (this generalizes the well-known result for **Set** [AK95, AM82]). Intuitively, the morphism from a given (CT, C)-algebra into $Beh_{CT}(C)$ constructs the behaviour of an element, which is the infinite tree given by all possible observations that can be made successively applying the observers until a value of observable sort (i.e. in S) is reached. □

Given a signature Σ, we formally add cofreeness constraints of form $cofree(CT)$, where

$$CT = (S, (obs_{ijk} : T_i \longrightarrow T_{ijk})_{i=1\ldots n, j=1\ldots m_i, k=1\ldots r_{ij}})$$

is a cotype with $Sig(CT) \subseteq \Sigma$, as Σ-sentences to the CoCasl logic. A cofreeness constraint $cofree(CT)$ holds in a Σ-algebra A if the reduct of A to $Sig(CT)$ is isomorphic to the behaviour algebra $Beh_{CT}(C)$ over the set of colours C with $C_s := A_s$ for $s \in S$.

Note that this implies the satisfaction of the cogeneratedness constraint

$$(S, \{obs_{ijk} | obs_{ijk} \text{ total}\}, \{obs_{ijk} | obs_{ijk} \text{ partial}\}, \emptyset),$$

i.e. each cofree cotype is also cogenerated. The converse does not hold, i.e. a cogenerated cotype need not be cofree. However, cogenerated *cotypes* still behave quite nicely (in contrast to arbitrary cogenerated *types*): the elements of carriers of the non-observable sorts (i.e. those outside S) are completely determined by their *behaviours*. Thus, the elements can be identified with their behaviours, and up to isomorphism, we have a submodel of the cofree model. Hence, cofreeness essentially adds the requirement that each possible behaviour is actually represented by an element.

Full abstractness of cofree cotypes implies that cofreeness is not destroyed in the presence of constructors. Normally, constructors are determined only up to bisimilarity and hence may destroy the homomorphism condition. However, in the cofree model, bisimilarity is just equality.

The main benefit of cofree cotypes (in comparison to cogenerated cotypes) is the principle

corecursive definitions in cofree cotypes are conservative.

This completes the definition of CoCasl constraint sentences. Note that in order to be able to translate the various constraints along signature morphisms in such a way that the satisfaction condition for institutions is fulfilled, one has to equip the constraints with an additional signature morphism, as in Sect. 3.1.

$$
\begin{array}{rll}
\varphi ::= & & t_1 = t_2 \\
& | & t_1 \stackrel{e}{=} t_2 \\
& | & def\ t \\
& | & [t]\,\varphi \\
& | & \langle t\rangle\,\varphi \\
& | & [t*]\,\varphi \\
& | & \langle t*\rangle\,\varphi \\
& | & [\{t_1,\ldots,t_n\}]\,\varphi \\
& | & \langle\{t_1,\ldots,t_n\}\rangle\,\varphi \\
& | & [\{t_1,\ldots,t_n\}*]\,\varphi \\
& | & \langle\{t_1,\ldots,t_n\}*\rangle\,\varphi
\end{array}
$$

Figure 3.14: Syntax of CoCasl's modal logic.

3.3.3 Modal logic

We now define a multi-sorted modal logic for use with process types, the basic idea being that observer operations give rise to modalities that describe the evolution of the system upon application of the observer. The underlying intuition is that the non-observable sorts of a process type form a multi-sorted state space, and that observers either directly produce observable values or effect an evolution of the state. Modal logic allows formulating statements about such systems without explicit reference to the states. The effect is that axioms formulated in modal logic indeed describe only the observable behaviour of a system, formally: satisfaction of modal formulae is bisimulation invariant (see e.g. [MSRR, Kur01a, Pat01]). Methodologically, this means that the state space is appropriately encapsulated; a technical advantage is that restriction by modal formulae preserves existence of final models.

In CoCasl, this takes the following shape. We define modal formulae for a given **cotype** declaration. All the sorts defined in the cotype are called *non-observable*, and the selectors are called *observers*. Sorts from the local environment are called *observable*. These notions can also be reformulated in terms of a signature of the modal CoCasl institution, see Sect. 3.3.4.

The full syntax of CoCasl's modal logic is given in Fig. 3.14 and explained successively in the sequel. Note that the syntax does not include propositional variables, since these would violate invariance under bisimulation. Atomic formulae in the modal logic involve *observer terms*. These are built from unary observers with *observable* result sort (which are treated as flexible constants, i.e. constants that depend on the respective state), observers with additional parameters (which then need to be applied to sufficiently many observer terms) and variables and function symbols from the local environment. The modal logic has (existential or strong) equations between as well as definedness assertions of observer terms as atomic sentences. Sentences may be combined using the usual propositional connectives, the quantification over variables of observable sorts, as well as the following modalities: An observer t (possibly applied to extra parameters) with *non-observable* result sort leads to modalities $[t]$, $\langle t\rangle$, $[t*]$, $\langle t*\rangle$ (all-next, some-next, always, eventually). Using this logic, we can write, in the example of Figure 3.15,

$$hd = 0 \ \wedge\ [tl]hd = 0 \ \Rightarrow\ [tl][tl]hd = 1$$

as syntactic sugar for

$$hd(s) = 0 \wedge hd(tl(s)) = 0 \Rightarrow hd(tl(tl(s))) = 1$$

More precisely, we define the meaning of a modal formula φ to be the meaning of the formula

$$\forall x : s[\![\varphi]\!]_{x:s}$$

Here, $[\![\_]\!]\_$ takes a modal formula (or an observer term) and a sorted term to an ordinary formula (or ordinary term). Intuitively, the sorted term, which is written as a subscript, carries the current state. This is defined as follows:

```
spec BitStream3 =
  free type Bit ::= 0 | 1
  cotype BitStream ::= (hd : Bit; tl : BitStream)
  ∀ s : BitStream
  • hd(s) = 0 ∧ hd(tl(s)) = 0 ⇒ hd(tl(tl(s))) = 1
end
```

Figure 3.15: Specification of a subset of bit streams in CoCasl.

- $[\![u]\!]_{t:s} \equiv u$, if u is an observer term consisting of variables and operation symbols from the local environment,
- $[\![f]\!]_{t:s} \equiv f(t)$ if $f\colon s \longrightarrow s'$ is a unary observer with observable result,
- $[\![f(t_1,\ldots,t_n)]\!]_{t:s} \equiv f([\![t_1]\!]_{t:s},\ldots,[\![t_n]\!]_{t:s},t)$, if $f\colon s_1 \times \cdots \times s_n \times s \longrightarrow s'$ is an observer with additional parameters and observable result, and t_i is an observer term of sort s_i $(i = 1,\ldots,n)$,
- $[\![u_1 = u_2]\!]_{t:s} \equiv [\![u_1]\!]_{t:s} = [\![u_2]\!]_{t:s}$,
- $[\![u_1 \stackrel{e}{=} u_2]\!]_{t:s} \equiv [\![u_1]\!]_{t:s} \stackrel{e}{=} [\![u_2]\!]_{t:s}$,
- $[\![def\ u]\!]_{t:s} \equiv def [\![u]\!]_{t:s}$,
- $[\![[f]\varphi]\!]_{t:s} \equiv def\ f(t) \Rightarrow [\![\varphi]\!]_{f(t):s'}$, if $f\colon s \longrightarrow s'$ is a unary observer with non-observable result,
- $[\![[f(t_1,\ldots,t_n)]\varphi]\!]_{t:s} \equiv def\ f([\![t_1]\!]_{t:s},\ldots,[\![t_n]\!]_{t:s},t) \Rightarrow [\![\varphi]\!]_{f([\![t_1]\!]_{t:s},\ldots,[\![t_n]\!]_{t:s},t):s'}$, if $f\colon s_1 \times \cdots \times s_n \times s \longrightarrow s'$ is an observer with additional parameters and non-observable result and t_i is an observer term of sort s_i $(i = 1,\ldots,n)$,
- $[\![[f]\varphi]\!]_{t:s} \equiv \forall x_1 : s_1,\ldots,x_n : s_n \,.\, def\ f(x_1,\ldots,x_n,t) \Rightarrow [\![\varphi]\!]_{f(x_1,\ldots,x_n,t):s'}$, if $f\colon s_1 \times \cdots \times s_n \times s \longrightarrow s'$ is an observer with additional parameters and non-observable result.

The translation is extended to the logical connectives and quantifiers by structural rules which just copy these.

Note that each modal formula has a *sort*, which is the sort occurring in the subscript argument of the translation function. In particular, a modal formula is well-formed and the translation function $[\![\_]\!]\_$ is defined only in case of correct sorting. One may switch to a different sort (i.e. a different state space) using the modalities, but only in a well-sorted way. If necessary (due to overloading), observers have to be provided with explicit types.

The other modalities now can be defined as derived notions, where the starred forms $[t*]$, $\langle t* \rangle$, being inspired by dynamic logic, need infinitary formulae. We here only treat the case of unary observers, the other cases being entirely analogous:

- $[\![\langle f \rangle \varphi]\!]_{t:s} \equiv \neg [\![[f] \neg \varphi]\!]_{t:s}$
- $[\![[f*]\varphi]\!]_{t:s} \equiv [\![\varphi \wedge [f]\varphi \wedge [f][f]\varphi \wedge [f][f][f]\varphi \wedge \ldots]\!]$
  (here, argument and result sort of f must coincide)
- $[\![\langle f* \rangle \varphi]\!]_{t:s} \equiv \neg [\![[f*] \neg \varphi]\!]_{t:s}$

The starred modalities have the limitation that only *one* specific observation can be repeated arbitrarily often. However, sometimes it is desirable to express that a group of observations can be repeated. We hence allow for grouping observers with braces: $[\{f_1,\ldots,f_n\}]$ and $\langle\{f_1,\ldots,f_n\}\rangle$

```
spec BitStream4 =
  free type Bit ::= 0 | 1
  cotype BitStream ::= (hd : Bit; tl : BitStream)
  • ⟨tl∗⟩hd = 1
end
```

Figure 3.16: Specification of a fairness property.

denote the conjunction and the disjunction, respectively, of the modal formulae obtained for the individual observers. Note that for the unstarred versions, this can also be expressed explicitly as a conjunction (disjunction), while this is not possible for the starred versions. This machinery allows us to express that a buffer eventually outputs all elements that are read in:

$$\forall a : Elem\,.\\ [next(input(a))]\,\langle\{next(input), next(output)\}*\rangle\,\langle next(output(a))\rangle\,true$$

The modal logic introduced above allows expressing safety or fairness properties. For example, the model of the specification BitStream4 of Figure 3.16 consists of bitstreams that will always eventually output a 1. Here, the 'always' stems from the fact that the modal formula is, on the outside, implicitly quantified over all states, i.e. over all elements of type $BitStream$.

Remark 3.28 The modal μ-calculus [Koz83], which provides a syntax for least and greatest fixed points of recursive modal predicate definitions, is expressible using free and cofree specifications: μ is expressible by free recursively defined predicates, while ν is expressible by cofree recursively defined predicates. We have refrained from including syntactical sugar for the μ-calculus in CoCasl, because this would involve higher order variables and hence appear to be against the grain of CoCasl, which is first-order in spirit (although higher-order types can be emulated).

Remark 3.29 In [MSRR], we also introduce modalities for structured observations. For simplicity and also because this is subject of ongoing research, we do not include these here.

3.3.4 The CoCasl Institution

We now come to the definition of the CoCasl institution. Actually, CoCasl does not form just a single institution, but a hierarchy of institutions with increasing expressiveness.

Definition 3.30 The *(plain)* CoCasl *institution* PlainCoCasl is identical to the Casl institution [CoF04], except that it has two additional types of sentences, namely, *cogeneratedness constraints* and *cofreeness constraints* as explained in Sections 3.3.1 and 3.3.2.

From this institution, which does not record enough information on cotype definitions in order to define the required notion of modal formula, we distinguish the modal CoCasl institution, defined as follows.

Definition 3.31 An *extended* CoCasl *signature* consists of a Casl signature (cf. Sect. 3.1) and the following additional data:

- a transitive relation *sees* and
- a partial equivalence relation *sibling* on the set of sorts.

A sort is called a *cotype* if it is in the domain of *sibling* (in signatures generated by CoCASL specifications, the cotypes in this sense will indeed be the sorts coming from **cotype** declarations).

Signature morphisms σ are required to preserve the *sibling* and *sees* relations.

The set of *sentences* associated to an extended CoCASL signature Σ consists of the sentences associated to the underlying CASL signature in the CoCASL institution and, additionally, the *modal formulae* over Σ. The syntax of modal formulae is defined as in Sect. 3.3.3, with flexible constants and modal operators determined as follows. We say that a modal operator has *type* $U \to S$ if it applies to modal formulae of type U, yielding modal formulae of type S. Each function symbol $f : R_1 \times \ldots \times R_n \times S \to W$, where S is a cotype that sees the R_i, gives rise to

- a parameterized flexible constant $f : R_1 \times \ldots \times R_n \to W$ if S *sees* W;
- simple modal operators $[f(r_1, \ldots, r_n)]$, $\langle f(r_1, \ldots, r_n)\rangle$, $[f(r_1, \ldots, r_n)*]$, and $\langle f(r_1, \ldots, r_n)*\rangle$ of type $W \to S$, parameterized over $r_i : R_i$, $i = 1, \ldots, n$, if W is a sibling of S.

Finally, modal operators can be combined e.g. in the form $[\{f_1(r_{11}, \ldots, r_{1n_1}), \ldots, f_l(r_{k1}, \ldots, r_{kn_k})\}]$, and parameters may be omitted as explained in Sect. 3.3.3.

Given a modal formula ϕ, the translation of ϕ along a signature morphism σ is defined by recursion over the formula structure. Here, modal operators and flexible constants associated to a function symbol f are translated into the corresponding entities for $\sigma(f)$, which exist by preservation of the *sees* and *sibling* relations.

The notions of *model* and *model reduction* for extended CoCASL signatures is the same as in the plain CoCASL institution.

The *satisfaction relation* for modal formulae is defined as described in Sect. 3.3.3.

Proposition and Definition 3.32 These data define an institution MODALCOCASL, the *modal* CoCASL *institution*.

PROOF: We have to establish the satisfaction condition for modal formulae. This is done by a straightforward induction on the formula structure. □

The additional data for extended CoCASL signatures show up in the semantics of CoCASL constructs as follows. The *sees* and *sibling* relations are determined purely by the **cotype** declarations. W.r.t. these relations, a **cotype** declaration of cotypes $S_1, \ldots, S_n$ has the following effects.

- The *sees* relation is extended by relations S_i *sees* U for all sorts U in the local environment such that S_i has a selector with result type U or a parameter (i.e. argument other than S_i) of type U, *except* when U is one of the S_j. The transitive closure of the resulting relation is the new *sees* relation.
- The cotypes $S_1, \ldots, S_n$ are declared to be *sibling*s. The partial equivalence generated by the resulting relation is the new *sibling* relation.

In particular, redeclaring a cotype may increase the number of sorts it *sees* as well as the number of its *sibling*s.

The intuition behind this is that the local environment is regarded as observable for purposes of observing a given cotype; i.e. the *sees* relation gives rise to a *local* notion of observability. In particular, it is possible to instantiate observable parameter sorts in a parameterized specification such as the specification LIST [**sort** *Elem*] of lists of entries of type *Elem* with, $\langle f(r_1, \ldots, r_n)\rangle$, $[f(r_1, \ldots, r_n)*]$, and $\langle f(r_1, \ldots, r_n)*\rangle$ a non-observable argument sort to obtain e.g. lists of streams.

Remark 3.33 The above definitions do not prevent the user from causing a certain amount of havoc by abusive renaming or redeclaration of symbols. E.g. it is possible to declare a cotype S that *sees* a sort T in its local environment, and then redeclare T as a cotype that *sees* S and hence itself. Then an observer $f : T \to T$ gives rise both to a flexible constant f and to a modal operator $[f]$, despite the proviso in the semantics of cotypes which excludes siblings from the *sees* relation. While this would certainly be regarded as a specification error — the sorts S and T would more

appropriately be defined within a single cotype declaration — we have preferred delegating this and further problems to a forthcoming set of methodological guidelines rather than overburden the definition of the signature category with further formal restrictions.

3.3.5 Amalgamation

Proposition 3.34 The category of ModalCoCasl signatures has pushouts.

Proof: Let $\sigma : \Sigma_1 \to \Sigma_2$ and $\tau : \Sigma_1 \to \Sigma_3$ be morphisms of extended CoCasl signatures, and let

$$\begin{array}{ccc} \Sigma_1 & \xrightarrow{\sigma} & \Sigma_2 \\ {\scriptstyle\tau}\downarrow & & \downarrow{\scriptstyle\bar{\tau}} \\ \Sigma_3 & \xrightarrow[\bar{\sigma}]{} & \Sigma_4 \end{array}$$

be the pushout of the underlying Casl signatures (which we denote by Σ_1 etc. as well). The resulting Casl signature Σ_4 is made into an extended CoCasl signature by taking the *sees* and *sibling* relations to be the smallest transitive relation and partial equivalence relation, respectively, that make $\bar{\sigma}$ and $\bar{\tau}$ morphisms of extended CoCasl signatures; the presentations in Σ_4 are defined as the images of the presentations in Σ_2 and in Σ_3 under $\bar{\sigma}$ and $\bar{\tau}$, respectively. This defines Σ_4 as a pushout of extended CoCasl signatures. □

Since both PlainCoCasl and ModalCasl models are just Casl models, and the sees and sibling relations do not interact with models, we have

Proposition 3.35 PlainCoCasl and ModalCoCasl admit amalgamation under the same restrictions as Casl (see Sect. 3.1.7).

3.4 HasCasl

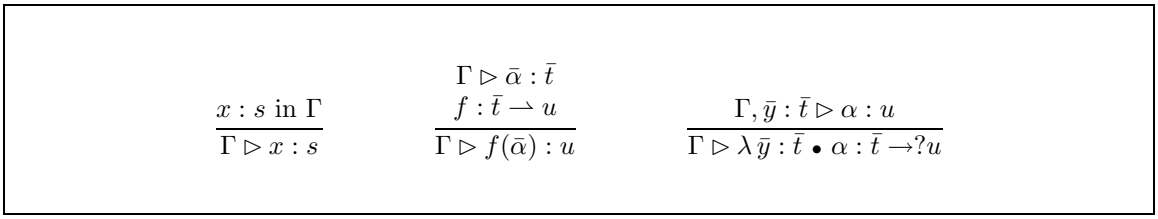
The development of programs in modern functional languages such as Haskell calls for a wide-spectrum specification formalism that supports the type system of such languages, in particular higher order types, type constructors, and parametric polymorphism, and that contains a functional language as an executable subset in order to facilitate rapid prototyping. HasCasl, a higher order extension of Casl, is geared towards precisely this purpose. Its semantics is tuned to allow program development by specification refinement, while at the same time staying close to the set-theoretic semantics of first order Casl. The number of primitive concepts in the logic has been kept as small as possible (actually, HasCasl is based on the partial λ-calculus). Various extensions to the logic, in particular general recursion and specification of side-effects encapsulated via monads, can be formulated within the language itself.

3.4.1 The partial λ-calculus

The natural generalization of the simply typed λ-calculus to the setting of partial functions is the partial λ-calculus as introduced in [Mog86, Mog88, Ros86]. The basic idea is that function types are replaced by partial function types, and λ-abstractions denote partial functions instead of total ones.

A ***simple signature*** consists of a set of ***sorts*** and a set of partial ***operators*** with given ***profiles*** (or arities) written $f : \bar{s} \rightharpoonup t$, where t is a ***type*** and $\bar{s}$ is a ***multi-type***, i.e. a (possibly empty) list of types. A type is either a sort or a ***partial function type*** (product types will be introduced in Sect. 3.4.2)

$$\bar{s} \to ?t,$$



$$\frac{x : s \text{ in } \Gamma}{\Gamma \rhd x : s} \qquad \frac{\Gamma \rhd \bar{\alpha} : \bar{t} \quad f : \bar{t} \rightharpoonup u}{\Gamma \rhd f(\bar{\alpha}) : u} \qquad \frac{\Gamma, \bar{y} : \bar{t} \rhd \alpha : u}{\Gamma \rhd \lambda\, \bar{y} : \bar{t} \bullet \alpha : \bar{t} \to ? u}$$

Figure 3.17: Typing rules for the partial λ-calculus

with $\bar{s}$ and t as above (one cannot resort to currying for multi-argument partial functions; e.g., $s \to ?(t \to ?u)$ is not isomorphic to $st \to ?u$ [Mog86]). Following [Mog86], we assume for each multi-type $\bar{s}$ and each type t an application operators with profile $(\bar{s} \to ?t)\bar{s} \rightharpoonup t$ in the signature, so that application does not require extra typing or deduction rules. This operator is denoted as usual by juxtaposition, while application of other operators in the signature is written with brackets. For $\bar{t} = (t_1, \ldots, t_m)$, $\bar{s} \to ?\bar{t}$ denotes the multi-type $(\bar{s} \to ?t_1, \ldots, \bar{s} \to ?t_m)$, not to be confused with the (non-existent) 'type' $\bar{s} \to ?t_1 \times \ldots \times t_m$. A ***morphism*** between two simple signatures is a pair of maps between the corresponding sets of sorts and operators, respectively, that is compatible with operator profiles.

A signature gives rise to a notion of typed terms in context according to the typing rules given in Figure 3.17, where a context Γ is a list $(x_1 : s_1, \ldots, x_n : s_n)$, shortly $(\bar{x} : \bar{s})$, of type assignments for distinct variables. More precisely, we speak simultaneously about terms and ***multi-terms***, i.e. lists of terms also denoted shortly in the form $\bar{\alpha}$ instead of $(\alpha_i, \ldots, \alpha_n)$. The judgement $\Gamma \rhd \alpha : t$ reads '(multi-)term α has (multi-)type t in context Γ'. The empty multi-term () doubles as a term of 'type' (), where the latter is also denoted as *Unit*.

A ***partial λ-theory*** $\mathcal{T}$ is a signature Σ together with a set $\mathcal{A}$ of ***axioms*** that take the form of existentially conditioned equations: an (existential) ***equation*** $\bar{\alpha}_1 \stackrel{e}{=} \bar{\alpha}_2$ is read '$\bar{\alpha}_1$ and α_2 are defined and equal'. Equations $\bar{\alpha} \stackrel{e}{=} \bar{\alpha}$ are abbreviated as $def\,\bar{\alpha}$ and called ***definedness judgements***. An ***existentially conditioned equation (ECE)*** is a sentence of the form $\Gamma \rhd def\,\bar{\alpha} \Rightarrow \phi$, where $\bar{\alpha}$ is a multi-term and ϕ is an equation in context Γ, to be read 'ϕ holds on the domain of $\bar{\alpha}$'. By equations between multi-terms, we can express conjunction of equations (e.g. $def(\alpha, \beta) \equiv def\,\alpha \wedge def\,\beta$); true will denote $def()$.

Remark 3.36 In the simple signature associated to a HasCasl signature according to the translation given in Sect. 3.4.5, all operators except the application operators will be ***total constants*** $f : t$, i.e. operators $f : () \rightharpoonup t$ together with an axiom $def\,f()$.

In Figure 3.18, we present a set of proof rules for existential equality in a partial λ-theory. The rules are parameterized over a fixed context Γ. We write $\Gamma \rhd def\,\bar{\alpha} \vdash \phi$ if an equation ϕ can be deduced from $def\,\bar{\alpha}$ in context Γ by means of these rules; in this case, $\Gamma \rhd def\,\bar{\alpha} \Rightarrow \phi$ is a ***theorem***. The rules are essentially a version of the calculus presented in [Mog86], adapted for existential (rather than strong) equations. Of course, there is no reflexive law, since $\alpha \stackrel{e}{=} \alpha$ is false if α is undefined. For conciseness, subderivations are denoted in the form $\Delta \rhd def\,\bar{\alpha} \vdash \phi$, where the context Δ and the assumption $def\,\bar{\alpha}$ are to be understood as *extending* the ambient context and assumptions. E.g. the first premise of rule (sub) reads 'in the context enlarged by $\bar{y} : \bar{t}$ and under the additional assumption $def\,\bar{\alpha}$, ϕ is derivable'. ***Strong equations*** $\Delta \rhd \alpha \stackrel{s}{=} \beta$, or just $\alpha \stackrel{s}{=} \beta$, are abbreviations for '$\Delta \rhd def\,\alpha \vdash def\,\beta$ and $\Delta \rhd def\,\beta \vdash \alpha \stackrel{e}{=} \beta$'; in particular, rule ($\beta$) is really two rules. Rule (ξ) implies that all λ-terms are defined.

The higher order rules (ξ) and (β) show a slight preference for strong equations. Note, however, that the usual form of the η-equation, $\lambda\, \bar{y} : \bar{t} \bullet \alpha(\bar{y}) = \alpha$, is an ECE, not a strong equation.

A ***translation*** between partial λ-theories $\mathcal{T}_1$ and $\mathcal{T}_2$ with signatures Σ_1 and Σ_2, respectively, is a signature morphism $\sigma : \Sigma_1 \to \Sigma_2$ which transforms axioms into theorems — i.e. for every axiom $\Gamma \rhd def\,\bar{\alpha} \Rightarrow \phi$ in $\mathcal{T}_1$, $\sigma\Gamma \rhd def\,\sigma\bar{\alpha} \vdash \sigma\phi$ in $\mathcal{T}_2$.

$$\textbf{(var)}\ \frac{x : s \text{ in } \Gamma}{def\, x} \qquad \textbf{(st)}\ \frac{def\, f(\bar{\alpha})}{def\, \bar{\alpha}} \qquad \textbf{(unit)}\ \frac{x : Unit \text{ in } \Gamma}{x \stackrel{e}{=} ()}$$

$$\textbf{(sym)}\ \frac{\bar{\alpha} \stackrel{e}{=} \bar{\beta}}{\bar{\beta} \stackrel{e}{=} \bar{\alpha}} \qquad \textbf{(tr)}\ \frac{\bar{\alpha} \stackrel{e}{=} \bar{\beta} \quad \bar{\beta} \stackrel{e}{=} \bar{\gamma}}{\bar{\alpha} \stackrel{e}{=} \bar{\gamma}} \qquad \textbf{(cong)}\ \frac{\bar{\alpha} \stackrel{e}{=} \bar{\beta} \quad def\, f(\bar{\alpha})}{f(\bar{\alpha}) \stackrel{e}{=} f(\bar{\beta})}$$

$$\textbf{(ax)}\ \frac{(\bar{y} : \bar{t} \rhd def\, \bar{\alpha} \Rightarrow \phi) \in \mathcal{A} \quad \bar{y} : \bar{t} \text{ in } \Gamma \quad def\, \bar{\alpha}}{\phi} \qquad \textbf{(sub)}\ \frac{\bar{y} : \bar{t} \rhd def\, \bar{\alpha} \vdash \phi \quad def\, (\bar{\alpha}[\bar{y}/\bar{\beta}]) \quad def\, (\bar{\beta})}{\phi[\bar{y}/\bar{\beta}]}$$

$$(\eta)\ \frac{x : \bar{t} \rightarrow ?u \text{ in } \Gamma}{(\lambda\, \bar{y} : \bar{t} \bullet x\, \bar{y}) \stackrel{e}{=} x} \qquad (\beta)\ \frac{\bar{y} : \bar{t} \text{ in } \Gamma}{(\lambda\, \bar{y} : \bar{t} \bullet \alpha)\, \bar{y} \stackrel{s}{=} \alpha}$$

$$(\xi)\ \frac{\bar{y} : \bar{t} \rhd \alpha \stackrel{s}{=} \beta}{\lambda\, \bar{y} : \bar{t} \bullet \alpha \stackrel{e}{=} \lambda\, \bar{y} : \bar{t} \bullet \beta}$$

Figure 3.18: Deduction rules for existential equality in context Γ

Remark 3.37 So-called ***conditioned terms*** $\bar{\alpha} res \bar{\beta}$, which denote the restriction of a multi-term $\bar{\alpha}$ to the domain of a multi-term $\bar{\beta}$ [Bur93, Mog88], can in our setting be coded using projection operators: we can put $\bar{\alpha} res \bar{\beta} = (\lambda \bar{x}, \bar{y} \bullet \bar{x})\,(\bar{\alpha}, \bar{\beta})$. Conditioned terms, in turn, provide a coding for λ-abstraction of multi-terms.

Remark 3.38 A notion of ***predicates*** is provided in the shape of terms $\Gamma \rhd \alpha : \mathit{Unit}$, for which we write α in place of $\mathit{def}\,\alpha$. The sentence $\mathit{def}\,\beta$ can be coded as the predicate $(\lambda x \bullet *)(\beta)$.

The expressive power of ECEs is greatly increased in the presence of an equality predicate (see also [CO89, Mog86]):

Definition 3.39 A partial λ-theory has ***internal equality*** if there exists, for each type s, a binary predicate (cf. Remark 3.38) eq_s such that

$$x, y : s \rhd eq_s(x, y) \Rightarrow x \stackrel{e}{=} y \quad \text{and} \quad x : s \rhd \mathsf{true} \Rightarrow eq_s(x, x)$$

Such a predicate allows coding conditional equations as ECEs. In combination with λ-abstraction, it gives rise to a full-fledged intuitionistic logic [CO89, LS86, SM02]; this is explained in more detail in Sect. 3.4.8.

The notion of model we choose for the partial λ-calculus and thus, in effect, for HasCasl, is that of ***intensional Henkin model***. Briefly, this means that not only may the sets interpreting partial function types fail to contain all set-theoretic partial functions, but they may also contain several elements describing the same set-theoretic function. Henkin models may be described as syntactic λ-algebras modeled on the corresponding notion defined for the total λ-calculus in [BTM85]:

Definition 3.40 A ***syntactic λ-algebra*** for a partial λ-theory $\mathcal{T}$ is a family of sets $[\![s]\!]$, indexed over all *types* of $\mathcal{T}$, together with partial interpretation functions

$$[\![\Gamma.\,\alpha]\!] : [\![\Gamma]\!] \rightharpoonup [\![t]\!]$$

for each term $\Gamma \rhd \alpha : t$ in $\mathcal{T}$, where $[\![\Gamma]\!]$ denotes the extension of the interpretation to contexts via the cartesian product. This interpretation is subject to the following conditions:

(i) $[\![\Gamma.\,x_i]\!]$, where $\Gamma = (\bar{x} : \bar{s})$, is the i-th projection;

(ii) $[\![\bar{y} : \bar{t}.\,\gamma]\!] \circ [\![\Gamma.\,\beta]\!] = [\![\Gamma.\,\gamma[\bar{y}/\beta]]\!]$, where $\Gamma \rhd \beta : \bar{t}$ is a multi-term, with the interpretation extended to multi-terms in the obvious way;

(iii) whenever $\Gamma \rhd \phi \vdash \alpha \stackrel{e}{=} \beta$ in $\mathcal{T}$ and $[\![\Gamma.\,\phi]\!](x)$ holds (i.e. is defined), then $[\![\Gamma.\,\alpha]\!](x) = [\![\Gamma.\,\beta]\!](x)$ are defined.

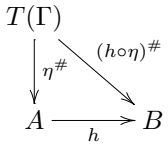
A ***model morphism*** between two syntactic λ-algebras is a family of functions h_s, where s ranges over all types, that satisfies the usual homomorphism condition for partial algebras w.r.t. all terms. A syntactic λ-algebra *satisfies* an ECE $\Gamma \rhd \mathit{def}\,\bar{\alpha} \Rightarrow \beta \stackrel{e}{=} \gamma$ if

$$\begin{array}{ll} [\![().\,\lambda\,\Gamma \bullet \beta res \bar{\alpha}]\!] & = [\![().\,\lambda\,\Gamma \bullet \gamma res \bar{\alpha}]\!] \quad \text{and} \\ [\![().\,\lambda\,\Gamma \bullet \mathit{def}\,(\beta, \bar{\alpha})]\!] & = [\![().\,\lambda\,\Gamma \bullet \mathit{def}\,\bar{\alpha}]\!]. \end{array}$$

It is shown in [Sch03] that such models are essentially equivalent to categorical models involving partial cartesian closed categories.

Remark 3.41 let $T_s(\Gamma)$ denote the set of terms of type s in context Γ. Then the family $T(\Gamma) = (T_s(\Gamma))$ can be equipped with total interpretation functions for all operators of the signature, but does not, of course, form a syntactic λ-algebra, since it fails to satisfy any equations, in particular β and η. However, Definition 3.40 says that $T(\Gamma)$ has a property resembling the universal property of classical term algebras: one has a family of partial evaluation functions $\eta^{\#} : T(\Gamma) \to A$ for each valuation η in A of the variables in Γ (in the above notation, $\eta^{\#}(\alpha) = A[\![\Gamma.\,\alpha]\!](\eta(\bar{x}))$), and that these

evaluation functions are compatible with homomorphisms in the sense that for each homomorphism h, the equation $h \circ \eta^\# = (h \circ \eta)^\#$ holds on the domain of $\eta^\#$, i.e. one has a lax commutative diagram



Remark 3.42 Having to interpret all λ-terms can be avoided by using combinators instead of λ-abstraction. In fact, it is implicit in [Mog88] that syntactic λ-algebras are equivalent to the combinatorically defined λ_p-*algebras* considered there; however, it is unclear whether λ_p-algebras can be finitely axiomatized.

A translation $\sigma : \mathcal{T}_1 \to \mathcal{T}_2$ of partial λ-theories gives rise to a ***reduct functor*** from the model category of $\mathcal{T}_2$ to that of $\mathcal{T}_1$: given a model M of $\mathcal{T}_2$, $M|_\sigma$ interprets each type and each term in $\mathcal{T}_1$ by the interpretation of its translation along σ in M.

3.4.2 Product types

In a first extension step, we add product types to the partial λ-calculus, i.e. essentially promote multi-types to types. In a ***signature with product types*** Σ, the set T of ***types*** is generated from the set S of sorts by the grammar

$$T ::= S \mid T \times \ldots \times T \mid T \to ?T,$$

with types of the form $\bar{s} \to ?t$ coded as $s_1 \times \ldots \times s_n \to ?t$; operators, however, have profiles consisting as before of a multi-type describing their arguments and a result type. Product types $s_1 \times \ldots \times s_n$ are equipped with tuple formation and projection operators $(\_, \ldots, \_) : \bar{s} \rightharpoonup s_1 \times \ldots \times s_n$ and $pr_i : s_1 \times \ldots \times s_n \rightharpoonup s_i$. Terms for Σ are defined as before, but using these new operators. Morphisms of such signatures are defined as for simple signatures; of course, compatibility with operator profiles now refers to an extension of the translation that takes product types into account. A ***partial λ-theory with product types*** is a signature with product types equipped with a set of ECEs, where ECEs can now be restricted to have a definedness condition for a term (rather than a multi-term) as their premise.

The semantics of a partial λ-theory $\mathcal{T}$ with product types is given by a translation into a partial λ-theory $\mathcal{T}'$: the sorts of $\mathcal{T}'$ are the sorts and the non-trivial product types of $\mathcal{T}$. This gives rise to an obvious translation of types in $\mathcal{T}$ to types in $\mathcal{T}'$. The operators of $\mathcal{T}'$ are, then, the operators of $\mathcal{T}$ with accordingly translated profiles. This, in turn, induces to a translation of terms; the axioms of $\mathcal{T}'$ are the correspondingly translated axioms of $\mathcal{T}$, extended by axioms stating that tuple formation and projections are mutual inverses, i.e.

$$\bar{x} : \bar{s} \rhd \mathsf{true} \Rightarrow pr_i(x_1, \ldots, x_n) \stackrel{e}{=} x_i \quad \text{and}$$
$$x : s_1 \times \ldots \times s_n \rhd \mathsf{true} \Rightarrow (pr_1(x), \ldots, pr_n(x)) \stackrel{e}{=} x.$$

The models of $\mathcal{T}$ are defined to be the models of $\mathcal{T}'$, and an ECE in $\mathcal{T}$ *holds* in such a model M iff its translation to an ECE in $\mathcal{T}'$ holds in M. Translations of partial λ-theories give rise to translations of the corresponding simple signatures and hence to ***reduct functors***.

3.4.3 Signatures

We now proceed to define actual HasCasl signatures, which will then be translated into simple signatures as defined in the previous section.

As it is standard in higher-order logic, operations are just constants of an appropriate (possibly higher-order) type. Moreover, the type of constants may be *polymorphic*, containing type variables

that may be instantiated later on. *Type constructors* map types to types. More generally, also higher-order type constructors are allowed, mapping type constructors to type constructors (where types are regarded as nullary type constructors). Declaration and application of type constructors is subject to correct *kinding*. Kinds can be regarded as sets of type constructors. Higher-order type constructors have higher-order kinds.

As in Haskell, a *type constructor class* is a user-declared subkind of a given kind, such that all members of the constructor class come with a bunch of operations (also called methods). Type constructors may be overloaded with several kinds built from different classes, but only if all these kinds are of the same shape, formalized as *raw kind*.

Finally, *type synonyms* just are abbreviations of more complex types by names.

A gentle introduction into polymorphic types, type constructors, kinds, and type classes is given in [HPF99].

A HasCasl ***signature*** $\Sigma = (C, \leq_C, T, A, O, \leq)$ consists of:

- a set C of ***type constructor classes*** (or just ***classes***) with assigned ***raw kinds***;
- a ***subclass*** relation $\leq_C$ between classes and ***kinds***
- a set T of ***type constructors*** consisting of their name and a set of kinds called ***profiles***;
- a set A of ***type synonyms***, where a type synonym associates a name to a ***pseudotype*** (i.e. a λ-term at the level of types; see below), its ***expansion***; and
- a set O of ***constant*** symbols with assigned ***type schemes*** (i.e types, possibly with a universal quantification over type variables at the outermost level; see below).
- a ***subtype*** relation between type constructors and pseudotypes.

The sets K and RK of *kinds* and *raw kinds*, respectively, are defined by the grammar

$$\begin{array}{ll} K & ::= C \mid \{V \bullet V \leq P\} \mid \mathit{Type} \mid K \cap K \mid K \to K \mid K^+ \to K \mid K^- \to K \\ RK & ::= \mathit{Type} \mid RK \to RK \mid RK^+ \to RK \mid RK^- \to RK, \end{array}$$

where *Type* is the kind of all types, P is the set of all pseudotypes (see below), and V is a set of type variables. The ***downset kind*** $\{a \bullet a \leq t\}$ denotes the kind of all subtypes of a given closed pseudotype t. The $+/-$ superscripts indicate covariant or contravariant dependency on the type arguments, respectively, for purposes of subtyping. A class Cl is associated to its raw kind Kd by writing $Cl : Kd$. The *raw kind* of a kind Kd is obtained by replacing each class occurring in Kd with its raw kind and each downset kind with the raw kind of the corresponding pseudotype as defined below. The formation of the ***intersection kind*** $Kd_1 \cap Kd_2$ is allowed only when Kd_1 and Kd_2 have the same raw kind, which is then also the raw kind of $Kd_1 \cap Kd_2$. We require that, whenever $Cl \leq_C Kd$, then the raw kinds of Cl and Kd are in the subkind relation. The subclass relation is extended to an order relation $\leq_K$ on kinds by the rules shown in Figure 3.19; note that co- and contravariant constructor kinds are subkinds of the corresponding constructor kind without variance information. The rule for intersection kinds works in both directions. By induction over the derivation length, it is shown that $Kd_1 \leq_K Kd_2$ implies that the same relation holds for the associated raw kinds, i.e. that the latter are identical up to possible removal of variance annotations. Note that a class or kind is not necessarily a subkind of its raw kind (e.g., given a class *Ord* of ordered types, $\mathit{Ord} \to \mathit{Ord}$ has raw kind $\mathit{Type} \to \mathit{Type}$, but is not a subkind of that kind.); however, for a class Cl of raw kind *Type*, it is required that $Cl \leq_C \mathit{Type}$.

By writing $t : Kd$, we express that a type t is associated to a kind Kd. We require that all the kinds assigned to a type constructor are of the same raw kind, which is then regarded as the ***raw kind*** of the type constructor (kinds *derivable* for the type constructor may have a greater raw kind). There are built-in type constructors

$$\begin{array}{ll} \_ \times \_ : & \mathit{Type}^+ \to \mathit{Type}^+ \to \mathit{Type}, \\ \_ \to ? \_,\ \_ \to \_ : & \mathit{Type}^- \to \mathit{Type}^+ \to \mathit{Type}, \text{ and} \\ \mathit{Unit} : & \mathit{Type} \end{array}$$

$$\frac{Cl \leq_C Kd}{Cl \leq_K Kd} \qquad \frac{Kd_1 \leq_K Kd_2 \quad Kd_3 \leq_K Kd_4}{Kd_2^{(+/-/.)} \to Kd_3 \leq_K Kd_1^{(+/-/.)} \to Kd_4}.$$

$$\frac{Kd \leq_K Kd_i, i = 1,2}{Kd \leq_K Kd_1 \cap Kd_2} \qquad \frac{t_1 \leq t_2}{\{a \bullet a \leq t_1\} \leq_K \{a \bullet a \leq t_2\}}$$

$$\frac{}{Kd_1^{(+/-)} \to Kd_2 \leq_K Kd_1 \to Kd_2}$$

$$\frac{}{Kd \leq_K Kd} \qquad \frac{Kd_1 \leq_K Kd_2 \quad Kd_2 \leq_K Kd_3}{Kd_1 \leq_K Kd_3}.$$

Figure 3.19: Subkinding rules

for products, partial and total function spaces, and the singleton type, respectively (with, in fact, an n-ary product type constructor $\_ \times \ldots \times \_$, covariant in all arguments, for each n).

A signature induces a set P of ***pseudotypes***, where a pseudotype, formed in a ***type context*** Θ of ***type variables***, is either a type variable, a type constructor, an application, or an abstraction. The type context consists of distinct type variables with assigned ***extended kinds***, denoted $(a_1 : Kd_1, \ldots, a_n : Kd_n)$ or, briefly, $(\bar{a} : \overline{Kd})$. Here, an extended kind is a kind, possibly annotated with a variance $(+/-)$ (called its ***outer variance***), as used in argument kinds of constructor kinds $Kd_1 \to Kd_2$.

More precisely, pseudotypes are formed and kinded according to the rules shown in Figure 3.20. A judgement of the form $\Theta \rhd t : Kd$ is to be read 't is a type constructor of kind Kd in context Θ which depends on the variables in Θ with the indicated variance'. The contexts Θ^{-1} and Θ^0 denote Θ with all outer variances reversed or removed, respectively. In the kinding rule for type abstraction, the variance of the abstracted variable in the premise must, of course, be identical to the variance of the argument kind in the conclusion. A pseudotype of kind *Type* is called a ***type***. The (unique) ***raw kind*** of a pseudotype can be calculated by the essentially the same set of rules, with the following modifications:

- type constructors are introduced with their raw kind instead of with one of their profiles
- type contexts contain only variables of raw kinds
- exact fits are required where the kinding rules have subkinding constraints, i.e. $\leq_K$ is replaced by $=$ throughout.

Note that the raw kind of a type constructor or pseudotype t need not be a derivable kind for t! The corresponding raw kinding judgements are written $\Theta \rhd_{\mathsf{raw}} t : Kd$.

It is easy to show that the kinds derivable for a pseudotype are upwards closed w.r.t the subkind relation (which is why we can require exact fits in the application rules). The raw kinds of kinds derivable for a pseudotype t are bounded below by the raw kind of t. Moreover, kinding is invariant under substitution and hence under β-equality (but not under η-equality, which is therefore not imposed on type constructors).

The subtype relation $\leq$ between type constructors and pseudotypes is extended to two preorders $\leq$ and $\leq_*$ on pseudotypes. The intuition behind this distinction is that certain subtypes will be mapped injectively into a supertype (recall that this is assumed for all subtypes in first order CASL),

$$\frac{t : Kd_1 \text{ in } \Sigma \quad Kd_1 \leq_K Kd_2}{\Theta \rhd t : Kd_2} \qquad \frac{a : Kd_1^{(+/.)} \text{ in } \Theta \quad Kd_1 \leq_K Kd_2}{\Theta \rhd a : Kd_2} \qquad \frac{\Theta \rhd t : Kd_1 \quad \Theta \rhd t : Kd_2}{\Theta \rhd t : Kd_1 \cap Kd_2}$$

$$\frac{\Theta \rhd t : Kd_1 \quad \Theta \rhd s : Kd_1 \to Kd_2}{\Theta^0 \rhd s\ t : Kd_2} \qquad \frac{\Theta \rhd t : Kd_1 \quad \Theta \rhd s : Kd_1^+ \to Kd_2}{\Theta \rhd s\ t : Kd_2}$$

$$\frac{\Theta^{-1} \rhd t : Kd_1 \quad \Theta \rhd s : Kd_1^- \to Kd_2}{\Theta \rhd s\ t : Kd_2} \qquad \frac{\Theta, a : Kd_1^{(+/-/.)} \rhd t : Kd_2 \quad Kd_3 \leq_K Kd_1}{\Theta \rhd \lambda\, a : Kd_1 \bullet t : Kd_3^{(+/-/.)} \to Kd_2}$$

Figure 3.20: Kinding rules for type constructors

while others may have non-injective coercion functions (e.g., function restriction). The former type of subtype relation is denoted by $\leq$, the latter by $\leq_*$. In Figure 3.21, this difference shows up in the rule for application of contravariant type constructors, which applies only to the relation $\leq_*$.

The subtyping relation implicitly contains

$$\_ \to \_ \leq \_ \to ? \_,$$

i.e. total functions can be regarded as partial when required. From this extended relation, the preorders on the set of pseudotypes are defined by the rules in Figure 3.21. Like kinding judgements, subtyping judgements are parameterized by a type context; however, for subtyping, the outer variances of type variables are irrelevant.

Type synonyms are intended as shorthands for pseudotypes; they are *not* meant as a means of constructing recursive types. More formally, expansion of type synonyms is required to be non-recursive, i.e. the relation 'the expansion of a contains b' on synonyms must be well-founded. A ***named pseudotype*** (as opposed to an *anonymous* pseudotype) is a pseudotype that can be constructed from type constructors, type synonyms, and its type context using only application (*not* λ-abstraction). Of course, any pseudotype can be made into a named pseudotype by just introducing suitable synonyms.

HasCasl features **ML-*polymorphism***, i.e. constants of types that contain type variables, implicitly or explicitly universally quantified *on the outermost level*. Thus, the types are complemented by ***type schemes***: A type scheme consists of a type context Θ and a *named* type t in that context (the variables of the type scheme will stem either from an explicit quantification or from a global or local variable declaration), together with a ***coherence flag*** stating whether or not instances are required to be coherent w.r.t. subtyping (typically, recursively defined polymorphic functions will be coherent, while predicates and functions used only for specification purposes may fail to be so); see also Sect. 3.4.4. Such a type scheme is written $\forall \Theta \bullet t$, with the coherence flag left implicit. Types will be regarded as type schemes with empty type context.

The ***constant symbols*** are given, like symbols in first order Casl, by their ***names*** with associated ***profiles***, the difference being that a profile is now represented by a single type scheme. A constant symbol with name f and profile t is written $f : t$. An operator is called ***monomorphic*** if t is a type, otherwise ***polymorphic***. The set O of constants contains the following ***distinguished constants***:

- the unique inhabitant of the unit type, $() : \mathit{Unit}$;

$$\frac{s \leq t \text{ in } \Sigma}{\Theta \rhd s \leq t} \qquad \frac{\Theta \rhd s \leq t}{\Theta \rhd s \leq_* t} \qquad \frac{\Theta \rhd_{\mathsf{raw}} t : Kd_1^+ \to Kd_2 \quad \Theta \rhd s_1 \leq s_2}{\Theta \rhd t\ s_1 \leq t\ s_2}$$

$$\frac{\Theta \rhd_{\mathsf{raw}} t : Kd_1^+ \to Kd_2 \quad \Theta \rhd s_1 \leq_* s_2}{\Theta \rhd t\ s_1 \leq_* t\ s_2} \qquad \frac{\Theta \rhd_{\mathsf{raw}} t : Kd_1^- \to Kd_2 \quad \Theta \rhd s_2 \leq_* s_1}{\Theta \rhd t\ s_1 \leq_* t\ s_2}$$

$$\frac{\Theta \rhd t_1 \leq t_2}{\Theta \rhd t_1\ s \leq t_2\ s} \qquad \frac{\Theta \rhd t_1 \leq_* t_2}{\Theta \rhd t_1\ s \leq_* t_2\ s}$$

$$\frac{\Theta, a : Kd_1 \rhd t \leq s \quad Kd_1 \leq_K Kd_2}{\lambda\, a : Kd_2 \bullet t \leq \lambda\, a : Kd_1 \bullet s} \qquad \frac{\Theta, a : Kd_1 \rhd t \leq_* s \quad Kd_1 \leq_K Kd_2}{\lambda\, a : Kd_2 \bullet t \leq_* \lambda\, a : Kd_1 \bullet s}$$

Figure 3.21: Subtyping rules for pseudotypes

- for each partial or total function type $s \to t$ or $s \to t$ an implicit ***application operator*** of profile $((s \to t) \times s) \to t$ or $((s \to t) \times s) \to t$, respectively;
- for each pair $s \leq t$ of types, a ***downcast*** operator $\_\ as\ s : t \to ?s$

Note that constant symbols may be ***overloaded***, i.e. different profiles can be associated to the same name. To ensure that there is no ambiguity in sentences at this level, constant symbols f are always ***qualified*** by their profile t when used, written f_t. (The language considered in [SMM] allows the omission of such qualifications when these are unambiguously determined by the context.) In fact, we require signatures to be *embedding-closed* (see also [SMT+01b]), i.e. the profiles associated to a given name must be upwards closed under $\leq_*$.[6] (Of course, embedding-closure is provided *implicitly*, so that the user is not actually required to specify all these profiles). This also makes sense of the profile of the upcast operator: $\_ : s$ implicitly has profiles $u \to t$ for all u, t with $u \leq s \leq t$.

A ***signature morphism***

$$\sigma : (C_1, \leq_C, T_1, A_1, O_1, \leq) \to (C_2, \leq_C, T_2, A_2, O_2, \leq)$$

consists of mappings from C_1 to C_2, from T_1 to $T_2 + A_2$, from A_1 to A_2, and from O_1 to O_2. These maps are required to preserve

- raw kinds of classes and type constructors
- the subclass relation in the sense that $Cl \leq_C Kd$ implies $Cl \leq_K Kd$.
- kinding judgements for type constructors in the sense that assigned kinds are mapped to derivable kinding judgements,
- expansions of type synonyms,
- profiles of constant symbols,

[6] This requirement makes it superfluous to define overloading relations as in CASL.

- all distinguished constants, and
- the subtyping relation $\leq$, again in the sense that subtyping judgements must be derivable in the target signature.

Moreover, distinguished constants must also be reflected (this in order to avoid ambiguities in the notation of signature morphisms). (This means that we could have omitted them for purposes of describing the signature category; they are included in the set of constants mainly in order to simplify the presentation of the typing and deduction rules.)

Remark 3.43 Note that the above definition explicitly allows type constructors to be mapped to type synonyms; this allows instantiating type constructors with pseudotypes, albeit at the cost of having to define an synonym first. A consequence is that the signature category fails to be cocomplete (while its non-full subcategory consisting of the signature morphisms that map type constructors to type constructors *is* cocomplete, being essentially the category of models of a Horn theory). However, the pushouts required for instantiating parameterized specifications do exist, which is all that is needed for HasCasl structured specifications.

As explained in Sect. 2.15, polymorphism can be achieved via derived signature morphisms. We hence generalize signature morphisms appropriately: A HasCasl *derived signature morphism* maps

- operator constants to terms;
- type constructors to λ-expressions which denote composite type constructors possibly containing subtype formation; and
- classes to subsets of the syntactic type universe.

A signature variable in this institution is an injective *plain* HasCasl signature morphism (which maps types to types, operators to operators etc. as usual) which is bijective on all syntactic entities except types. (This illustrates the necessity of the restricted cocompleteness requirement for institutions with signature variables: pushouts of derived signature morphisms in general fail to exist, while pushouts of derived signature morphisms along signature variables do exist; this phenomenon is typical of derived signature morphisms in general.) Then, polymorphic formulae and their satisfaction as defined above coincide with the corresponding notions in HasCasl as explained in Sect. 3.4. E.g., if $\sigma : \Sigma_1 \hookrightarrow \Sigma_2$ extends Σ_1 by a single new type constant a, then the polymorphic formula $\forall\sigma.\,\phi$ is equivalent to the polymorphic HasCasl sentence $\forall a : \mathit{Type}.\,\phi$: the left inverses τ of σ correspond to the possible instantiations of the type variable a in Σ_1. Note that the interpretation of instances of polymorphic operators involving a is forced by the interpretation of a, since, as emphasized above, signature morphisms map polymorphic operators as single entities.

Thanks to the richness of HasCasl specifications, model-expansive extensions of theories (cf. Sect. 2.15.5) are indeed the expected ones under this definition; this includes

- equational definitions
- well-founded recursive definitions of functions into types that admit a unique description operator [Sch]
- general recursive definitions over cpo's
- inductive datatype definitions, provided that the base theory already contains the natural numbers (this is a categorical result inherited from topos theory [MP00])
- class declarations.

If we move to the category of theories and then understand equality of such signature morphisms to hold only up to provable equivalence of terms, then even all pushouts of derived signature morphisms exist. This will be needed in Sect. 3.4.10 below. A more precise background is given in [Sch].

3.4.4 Models

The model semantics of HasCasl is split into two levels. The first level yields an institution only for the fragment of HasCasl without polymorphism, while in the general case one obtains an *rps preinstitution* [SS93], i.e. the satisfaction condition holds only in the direction leading from the extended to the reduced model. This is remedied at the second level by means of a general mechanism for transforming rps preinstitutions into institutions, which we define but do not discuss in detail here; a fuller presentation is given in [SM04].

At the ***first level***, the models of a signature Σ are defined by a translation of Σ into a partial λ-theory with products $\mathsf{Th}(\Sigma)$ to be defined in Sect. 3.4.5 — i.e. the models of Σ are defined to be the syntactic λ-algebras for $\mathsf{Th}(\Sigma)$, correspondingly for model morphisms. In the interest of compatibility with first-order Casl, the carrier sets of all types are additionally required to be non-empty. Note that every Casl model of a signature Σ can be extended to a HasCasl model of Σ regarded as a HasCasl signature. However, this extension will in general not be unique; in particular, there is always a free extension, where function types are in a sense minimal, and a standard extension with function types interpreted maximally, i.e. by full function sets.

At the ***second level***, models of a signature Σ are pairs (N, σ), where σ is a ***derived signature morphism*** (see Sect. 3.4.3) $\Sigma \to \Sigma_2$ into a further signature Σ_2, and N is a model of Σ_2 at the first level. Here, derived signature morphisms generalize signature morphisms in that they may map type constructors to pseudotypes, including types formed by subtype comprehension, and constant symbols to terms. ***Subtype comprehension***, in turn, refers to the formation of types of the form $\{x : t \mid \phi\}$, where t is a type and ϕ is a formula. The reduct of (N, σ) along a signature morphism τ into Σ is $(N, \sigma \circ \tau)$. This construction, together with the definition of satisfaction given in Sect. 3.4.7, ensures that the satisfaction condition holds.

3.4.5 Translation of HasCasl signatures into partial λ-theories

An ***instance*** of a kind is essentially a closed named pseudotype of that kind, taken modulo β-equality; in addition to the usual type forming operators, an instance may however include the use of subtype formation by definedness assertions for terms. The sorts of $\mathsf{Th}(\Sigma)$ are the ***loose types*** of Σ, where a loose type is an application of a type constructor other than the built-in type constructors $\times$, $\to$, $\to$, and *Unit*, to an instance of its argument kind. This gives rise to a recursively defined translation of kind instances in Σ to types in $\mathsf{Th}(\Sigma)$ (and conversely), since in β-normal types, λ-abstractions can only occur nested inside loose types. We leave this translation implicit in the notation.

The operators of $\mathsf{Th}(\Sigma)$ are defined to be

- for each operator f with profile $\forall \bar{a} : \overline{Kd} \bullet t$ a family of total constants (cf. Remark 3.36) $f_{\bar{s}} : t[\bar{s}/\bar{a}]$, indexed over all instances $\bar{s} : \overline{Kd}$;
- for each pair (s, t) of types in $\mathsf{Th}(\Sigma)$ such that $s \leq_* t$ holds for the corresponding types in Σ, an ***embedding operator***
$$em_{s,t} : s \rightharpoonup t.$$

Finally, $\mathsf{Th}(\Sigma)$ has the following axioms (all expressible as ECEs):

- ***coherence of subtyping*** essentially as in Casl;
- ***overloading axioms*** stating for each pair (s, t) of types in $\mathsf{Th}(\Sigma)$ and each constant $c : s$ that
$$em_{s,t}(c : s) = c : t$$
(where the profile $c : t$ is in Σ by embedding closure).
- injectivity of subtype embeddings $em_{s,t}$ for $s \leq t$ (*not*, more generally, for $s \leq_* t$), expressed by their mutual inverse property with the corresponding downcast operators (also as in Casl);

- ***coherence*** of correspondingly flagged polymorphic operators w.r.t. subtyping: if $f : \forall \bar{a} : \overline{Kd} \bullet t$ is a coherent polymorphic operator, and $\bar{s}$ and $\bar{u}$ are instances of $\overline{Kd}$ such that $t[\bar{s}/\bar{a}] \leq_* t[\bar{u}/\bar{a}]$, then
$$f_{\bar{u}} = em_{t[\bar{s}/\bar{a}],t[\bar{u}/\bar{a}]}(f_{\bar{s}}).$$

A signature morphism $\sigma : \Sigma_1 \to \Sigma_2$ induces a morphism $\mathsf{Th}(\sigma) : \mathsf{Th}(\Sigma_1) \to \mathsf{Th}(\Sigma_2)$ of simple signatures with products. The ***reduct functor*** for σ is defined to be that of $\mathsf{Th}(\sigma)$.

3.4.6 Sentences

Sentences for a signature Σ are just atomic formulae, understood to be universally quantified:

- fully-qualified terms of sort *Unit*, regarded as predicates qua implicit definedness assertions
- definedness assertions $def\_$ for fully-qualified terms
- existential and strong equations $\_ \stackrel{e}{=} \_$ and $\_ = \_$, respectively, between fully-qualified terms of the same sort.

Here, a *fully-qualified term* (or, when no confusion is likely, just a ***term***) is a term in $\mathsf{Th}(\Sigma)$ (with the context determined by enclosing quantifications). A fully-qualified term in type context Θ (arising from enclosing universal quantifications over type variables) is a fully-qualified term in $\Sigma+\Theta$, where $\Sigma+\Theta$ is obtained by extending Σ with the variables in Θ, regarded as type constructors of the appropriate kinds (with raw kinds determined by their unique kinds).

As syntactical sugar over these sentences, one has the following additional features:

- for each pair (s, t) of types in $\mathsf{Th}(\Sigma)$, an elementhood operator $\_ \in s : t \to Unit$ abbreviating $\lambda\, x : t \bullet def\, x\ as\ s$;
- a total λ-abstraction $\lambda\, \bar{x} : \bar{s} \bullet !\, \alpha$ which abbreviates a downcast of the partial λ-abstraction to the type of total functions;
- syntactical support for emulation of ***non-strict functions*** by the ***procedural lifting method***: let $?t$ abbreviate the type $Unit \to ?t$. We admit terms formed using two additional typing rules: a function that expects an argument of type t (possibly as part of a product type) may be applied to a term α of type $?t$, which is then implicitly replaced by $\alpha()$; conversely, a function that expects an argument of type $?t$ accepts arguments β of type t, which are implicitly replaced by $\lambda\, x : Unit \bullet \beta$ (where x is a fresh variable).
- ***let-terms***: for a term $\alpha : t$ in type context $\Theta = (\bar{a} : \overline{Kd})$, a variable x, and a term β in ***operator context*** $x : \forall\Theta \bullet t$, one has a term
$$let\ \forall\Theta \bullet x = \alpha\ in\ \beta;$$
here, a term in operator context $x : \forall\Theta \bullet t$ is a term in the signature obtained from Σ by adding a constant $x : \forall\Theta \bullet t$. Such a let-term abbreviates β with all occurrences $x_{\bar{s}}$ of x substituted by $\alpha[\bar{s}/\bar{a}]$. Repeated bindings $let\ \forall\Theta_1 \bullet x_1 = \alpha_1\ in\ let\ \forall\Theta_2 \bullet x_2 = \alpha_2\ in\ \ldots$ are abbreviated as $let\ \forall\Theta_1 \bullet x_1 = \alpha_1;\ \forall\Theta_2 \bullet x_2 = \alpha_2\ in\ \ldots$.
- ***recursive datatypes*** are syntactical sugar for the usual no-junk-no-confusion axioms; details are laid out in [SMM].

3.4.7 Satisfaction

At the first level of the model semantics (cf. Sect. 3.4.4), the ***satisfaction*** of a sentence in a model M is determined as usual by the holding of its atomic formulae w.r.t. assignments of (defined) values to all the variables that occur in them, the values assigned to variables of sort s being in s^M. The value of a term w.r.t. a variable assignment may be undefined, due to the application of a partial function during the evaluation of the term. Note, however, that the satisfaction of sentences is 2-valued (as is the holding of open formulae with respect to variable assignments). The satisfaction of a universal quantification over type variables is defined as satisfaction of all instances of that formula (this is possible because quantifications over type variables are allowed only at the outermost level).

A term of type *Unit* holds as an atomic formula if it is defined in M. A definedness assertion concerning a term holds iff the value of the term is defined (thus it corresponds to the application of an operator $a \rightharpoonup \mathit{Unit}$ to the term). An existential equation holds iff the values of both terms are defined and identical, whereas a strong equation holds also when the values of both terms are undefined.

Since the type context has been 'substituted away', every term α occurring in the expanded formulae is a term in $\mathsf{Th}(\Sigma)$. The value of α is determined as follows: the given variable assignment for the context Γ is an element x of $[\![\Gamma]\!]$ (cf. Definition 3.40); the value of α is defined to be $[\![\Gamma.\,\alpha]\!](x)$.

At the second level of the semantics, a model (N, σ) of Σ satisfies a sentence ϕ iff N satisfies the translated sentence $\sigma\phi$ at the first level.

3.4.8 The Internal Logic

The basic logic of HasCasl is quite limited: there are no logical connectives and only outer universal quantification. One can, however, emulate conjunction, the constant true proposition, and universal quantification, the latter via the elementhood predicate for total function types as subtypes of partial function types. However, we can strengthen the logic by means of an ***internal equality***. Let *Pred a* abbreviate the type $a \to ?\mathit{Unit}$, and call the inhabitants of (*Pred a*) ***predicates***. A predicate

$$\mathit{eq} : \forall a \bullet \mathit{Pred}\ (a \times a)$$

in a partial λ-theory (with products) is called an internal equality (see also [Mog86]) if $\mathit{eq}(x, y)$ is equivalent to $x \stackrel{e}{=} y$ in the deduction system of Figure 3.18 (due to intensionality, this is a stronger property than equivalence of the two formulae for each pair (x, y) of elements of a in a model).

In fact, internal equality can be specified in HasCasl. The introduction of internal equality is highly non-conservative, since it makes the logic available within λ-abstracted predicates substantially richer: one can define all quantifiers and logical operators of intuitionistic higher order logic similarly as in [LS86]. The specification of internal equality and the new connectives is given in Figure 3.22. The specification uses the type of truth values $\mathit{Logical} = \mathit{Pred\ Unit}$. Moreover, it liberally applies the support for non-strict functions to pass back and forth between predicate applications, i.e. partial terms of type *Unit*, and terms of type *Logical*.

In order to improve readability, the equality symbol $\stackrel{e}{=}$ can, after all, be used within λ-terms, but is, then, implicitly replaced by *eq*. It may come as a surprise that the last formula shown in Figure 3.22 as a consequence of the definitions expresses a form of extensionality; however, it is well-known that all categorical models are 'internally extensional' [MS89].

The internal logic is intuitionistic: there may be more than two truth values, and $\mathit{neg}(\mathit{neg}\ A)$ is in general different from A. The obvious deduction rules can be proved as lemmas; e.g., it is not hard to show that the rule

$$\frac{\phi\ \mathit{impl}\ \psi; \qquad \phi}{\psi}$$

is derivable from the rules in Figure 3.18 and the definitions in Figure 3.22.

The internal logic is used in specifications by implicitly importing its specification; the usual logical connectives and quantifiers are construed as operations of this specification.

spec INTERNALLOGIC =
var $a : Type$
funs $tt : Logical = \lambda\, x : Unit \bullet ()$
$all : Pred(Pred\ a) = \lambda\, p : Pred\ a \bullet p \in (a \to Unit)$
$\_\_ \,\&\, \_\_ : Pred(Logical \times Logical) = \lambda\, x, y : Logical \bullet def(x(), y())$
then
fun $eq : Pred(a \times a)$
- $all(\lambda\, x : a \bullet eq(x, x))$
- $\lambda\, x, y : a \bullet fst(x, eq(x, y)) = \lambda\, x, y : a \bullet fst(y, eq(x, y))$

then %def
funs $\_\_impl\_\_, \_\_or\_\_ : Pred(Logical \times Logical)$
$ff : Logical$
$neg : Pred\ Logical$
$ex : Pred(Pred\ a)$
- $\_impl\_ = \lambda\, x, y : Logical \bullet eq[Logical](x, x\ \&\ y)$
- $\_or\_ = \lambda\, x, y : Logical \bullet all(\lambda\, r : Logical \bullet$
$((x\ impl\ r)\ \&\ (y\ impl\ r))\ impl\ r)$
- $ff = \lambda\, y : Unit \bullet all(\lambda\, x : Logical \bullet x)$
- $neg = \lambda\, x : Logical \bullet x\ impl\ ff$
- $ex[a] = \lambda\, p : Pred\ a \bullet all(\lambda\, r : Logical \bullet$
$all(\lambda\, x : a \bullet p(x)\ impl\ r))\ impl\ r$

then %implies
var $a, b : Type$
- $all(\lambda\, f, g : a \to ?\ b \bullet all(\lambda\, x : a \bullet eq[?b](f(x), g(x)))\ impl\ eq(f, g))$

Figure 3.22: Specification of the internal logic

3.4.9 Proof Calculus

The proof calculus for the partial λ-calculus is given in Fig. 3.18. By the translation of HasCasl to the partial λ-calculus given in Sect. 3.4.5, it applies to HasCasl as well.

3.4.10 Amalgamation in HasCasl

The HasCasl signature category fails to be cocomplete; this is due to the possibility to map type constructors to type aliases; two such mappings may be contradictory and hence in general cannot be unified in a pushout signature. However, it not hard to show:

Proposition 3.44 The subcategory of HasCasl signatures and signature morphisms not involving type aliases is cocomplete.

HasCasl fails to be semi-exact:

Example 3.45 Let Σ_1 consist of a nullary type constructor s, and Σ_2 consist of a nullary type constructor t.

$$\begin{array}{ccc} \emptyset & \longrightarrow & \Sigma_2 \\ \downarrow & & \downarrow \\ \Sigma_1 & \longrightarrow & \Sigma' \end{array}$$

Then Σ' exhibits the type $s \to t$; however, due to Henkin semantics, its interpretation is not uniquely determined by the interpretations of s and t.

Example 3.46 Let Σ_1 consist of a nullary type constructor s, and Σ_2 consist of a unary type constructor t.

$$\begin{array}{ccc} \emptyset & \longrightarrow & \Sigma_2 \\ \downarrow & & \downarrow \\ \Sigma_1 & \longrightarrow & \Sigma' \end{array}$$

Then Σ' exhibits the type $t(s)$; however, its interpretation is not constrained at all.

Luckily, we still have the following:

Proposition 3.47 First-level HasCasl admits finite weak amalgamation.

Proof: Clearly, the initial (=empty) signature has a model. Since finite colimits can be obtained via pushouts and initial objects, it remains to show semi-exactness.

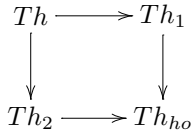
Let a pushout

$$\begin{array}{ccc} \Sigma & \longrightarrow & \Sigma_1 \\ \downarrow & & \downarrow \\ \Sigma_2 & \longrightarrow & \langle \Sigma', \Psi \rangle \end{array}$$

of HasCasl derived signature morphisms be given, and translate this diagram into the category of partial λ-theories:

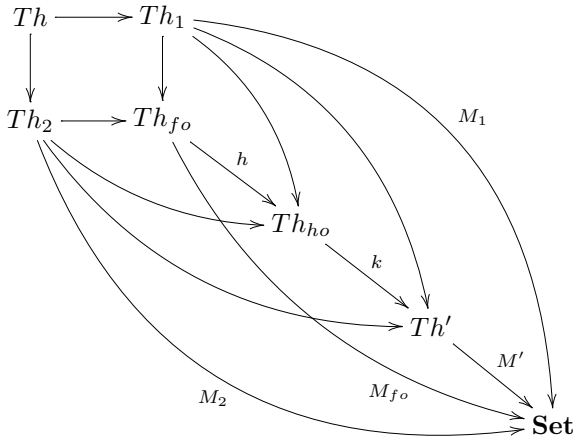
$$\begin{array}{ccc} Th & \longrightarrow & Th_1 \\ \downarrow & & \downarrow \\ Th_2 & \longrightarrow & Th' \end{array}$$

This diagram need not be a pushout, but by the universal property of the pushout



there is a morphism $k: Th_{ho} \longrightarrow Th'$ of partial λ-theories.

By the results of [Sch], HasCasl Henkin models (here: M_1 and M_2) are equivalently described as first-order functors from classifying categories of theories into **Set**. Since Hom-functors preserve limits, we have an amalgamated model M_{fo} for the pushout Th_{fo} of *first-order* finitary partial theories. Now by the pushout property, the is a morphism h of the pushout of first-order finitary partial theories into the pushout Th_{ho} of higher-order partial λ-theories. Since the latter also is a first-order finitary partial theory, and these admit free extensions along theory morphisms, M_{fo} has a free extension M' along $k \circ h$, and this is a suitable amalgamation of M_1 and M_2.



□

Unfortunately, this result does not carry over to second-level (i.e. polymorphic) HasCasl; here, even weak semi-exactness fails, because two extended models need to carry identical ordinary models in order to be amalgamable. Thus, the satisfactory institutional treatment of polymorphism under the requirement of weak semi-exactness remains an open research problem.

3.4.11 HasCasl Language Constructs

A HasCasl specification is essentially a convenient way to determine the signature and axioms of a partial λ-theory. The only actual additional language feature is that HasCasl has shallow type class polymorphism, which however is semantically coded out by collections of instances (w.r.t. the generic framework of Casl structured and architectural specifications, this has the effect that one obtains a so-called rps pre-institution rather than an institution; however, this is not relevant for the results presented here).

HasCasl signatures are written as follows. Basic types are introduced by means of the keyword **type**. Types may be parameterized by type arguments; e.g., we may write

var a : **Type**
type *List a*

and obtain a unary type constructor *List*. There are built-in type constructors (with fixed interpretations) $\_ * \_$ for product types, $\_ - > ? \_$ and $\_ - > \_$ for partial and total function types, respectively, *Pred* for predicate types, and a unit type *Unit*. A *type* is, then, anything that can be formed from the basic types and the existing type constructors.

Next, an *operator* is a constant f of some type t. The type t may contain type variables, making f an ML-style polymorphic operator. An operator is declared by

op $f : t$

From the given operators, we may form higher order terms in the usual way: a term is either a variable, an application, a tuple, or a (multi-argument) λ-abstraction. Such terms may then be used in *axioms* which may be formulated in the internal logic described in the preceding sections. Axioms may be explicitly or implicitly universally quantified over type variables at the outermost level.

Classes are declared in the form

class C

and are to be understood as subsets of the syntactical universe of all types. Types as well as type variables can be restricted to belong to an assigned class, e.g. by writing

type $t : C$

In particular, axioms and operators may be polymorphic over classes. Classes may be subclasses of each other, and they may have generic instances. By attaching polymorphic operators and axioms to a class, one achieves a similar effect as with Haskell's type classes.

More generally, one also has *constructor classes*, i.e. classes of polymorphic types such as *List*. They are interpreted as predicates on the syntactical universe of abstracted type expressions (also called *pseudotypes*), e.g.

$$\lambda\, a : Type \bullet a \to ?\ List\ a$$

Constructor classes may have subclasses; types, operators, and axioms may be polymorphic over constructor classes. A typical example of a constructor class is the class of monads.

The semantics of a HasCasl specification is then given by a translation into a partial λ-theory, where polymorphism of types, operators, and axioms is coded out by means of collections of instances; for definiteness, the chosen notion of model is that of intensional Henkin model — however, we will more often think of models as living in suitable pccc's.

By means of the internal logic, one can specify a class of complete partial orders and fixed point recursion in much the same style as in HOLCF [Reg95]. On top of this, syntactical sugar is provided that allows recursive function definitions in the style used in functional programming, indicated by the keyword **program**.

Moreover, HasCasl provides syntactic sugar for *monads* much in the same way Haskell does. Monads have been recognized by Moggi as an elegant device for dealing with stateful computation in functional programming languages. In addition to the Haskell "do"-notation, HasCasl provides a Hoare calculus for partial and total correctness of monadic programs and a dynamic logic. All this has been done in an entirely monad-independent way, on top of the HasCasl institution.

In summary, HasCasl is a language that allows both property-oriented specification and functional programming; executable HasCasl specifications may easily be translated into Haskell programs. As to the generality of the results obtained, one should keep in mind that HasCasl is just syntactical sugar for a standard intuitionistic higher order logic of partial functions (the internal logic of pccc's with equality).

3.4.12 Functional Programs: Haskell

The development of an institution for Haskell is subject of current research. Basically, Haskell signatures will be similar to HasCasl signatures, with additional constructs for declaring algebraic datatypes. The semantics of non-polymorphic Haskell will be given by a translation into the meta-language FPC, based on previous work by David Aspinall [Asp97]. However, we will probably refrain from adding a second level type system for specification purposes to Haskell, as Aspinall has done in his framework. Instead, we will try to institutionalize the *P-logic* [Kie03], a logic that has been built on top of Haskell and is the main logic of the *Programatica* project [Hal03], and that already has been (via a cooperation with Programatica) integrated into the Heterogeneous Tool Set.

3.5 CSP-CASL

CSP-CASL [Rog] is a combination of the process algebra CSP with CASL. The alphabet of CSP actions is obtained by the carrier sets of CASL models. The interest in formalizing CSP-CASL as an institution is that the semantics of CASL structured and architectural specifications carries over to CSP-CASL. It has been doubted if the latter have a meaning at all for CSP-CASL. However, it seems to be interesting that arbitrary CSP-CASL specifications can be used as formal parameters of specifications. On the CSP side, this leads to the possibility to use formal process names that are instantiated later on with actual processes. Moreover, while CSP has a program-like fixed one-model semantics for a given specification, CSP-CASL offers the possibility to mix this with loose semantics.

For simplicity, we equip CSP with an operational semantics based on labelled transition systems. Note that the other semantics used for CSP, like trace semantics, failure/divergence semantics and stable failures semantics can be derived from the operational one. Indeed, the main reason to have these different semantics are different notions of refinement. But then, one can equally well use just one semantics (labelled transition systems), plus a set of notions of refinement between LTS to capture these various semantics.

The definition of an institution for CSP-CASL is still ongoing work. Here, we only can provide a snapshot. The formal definition is based on an institution comorphism $(\Phi, \alpha, \beta)\colon PFOL^{=} \longrightarrow FOL^{=}$ that codes out partiality by adding a bottom value to carriers, very similar to our comorphism (4a) in Sect. 4.1.1 below. The details are described in [Rog].

Signatures CSP-CASL-Signatures (Σ, N) consist of a CASL signature Σ enjoying the local top sort property (i.e. any two sorts with a common lower bound have a common upper bound), together with a set N of process names.

Signature morphisms CSP-CASL-Signature morphisms $(\sigma, f)\colon (\Sigma_1, N_1) \longrightarrow (\Sigma_2, N_2)$ consist of a data-logic signature morphism $\sigma\colon \Sigma_1 \longrightarrow \Sigma_2$ in the sense of [Rog], i.e. σ is CASL signature morphisms that reflects and does not extend the subsort relation, together with a function $f\colon N_1 \longrightarrow N_2$.

Models A (Σ, N)-model (M, T) consists of a CASL Σ-model M and a function T from N into the set of labelled transition systems over the alphabet $A(\beta(M)) \uplus \{\tau, \dot{\tau}\}$. Recall from [Rog] that $\beta(M)$[7] extends the carriers by an additional element $\bot$ and that $A(M) = (\biguplus_{s \in S} M_s)_{/\sim}$, where $\sim$ is an equivalence identifying non-$\bot$ elements that are equal when injected into common supersorts, as well as $\bot$-elements of sorts having a common supersort.

There are several notions of model for CSP-CASL. We here choose the most informative one, based on labeled transition systems (LTS) [Ros97]. Thus, models are CASL-models, possibly equipped with an LTS being labeled in the disjoint union of all carriers. On the CASL-part, reducts are as in CASL. If a model is not equipped with an LTS, neither is its reduct. If a model is equipped with an LTS, and the LTS is labeled only with labels from carriers of the CASL-reduct, it is quite straightforward to construe the LTS as an LTS for the CASL-reduct (injectivity of signature morphisms on sorts ensures that carrier sets are not doubled). Otherwise, the LTS is deleted.

Sentences are either CASL sentences, or CSP process terms [Hoa85, Ros97] involving CASL-terms in place of alphabet letters. Sentence translation is straightforward.

Satisfaction for CASL sentences is as in CASL. A CSP process term P is evaluated using the CASL-part a model M, leading to an LTS L with labels in the disjoint union of all carriers. Now M satisfies P iff M is equipped with L. Details can be found in [Rog].

Model homomorphisms A (Σ, N)-model homomorphism $h\colon (M_1, T_1) \longrightarrow (M_2, T_2)$ is a Σ-homomorphism $h\colon M_1 \longrightarrow M_2$ in CASL such that $h(T_1(n)) = T_2(n)$ for each $n \in N$ (where h extends to labelled transition systems in a component wise manner).

[7] β is also called ext in [Rog].

Model reducts The way of defining model reducts is crucial for obtaining the satisfaction condition. The central problem is what to do with labels that are lost, because the carrier set they stem from is forgotten while taking the reduct. The idea is to replace any alphabet letter that is not available in the reduct with an invisible action $\dot{\tau}$ *that does not occur in the* CSP *semantics itself*. Formally, given a signature morphism $(\sigma, f)\colon (\Sigma_1, N_1) \longrightarrow (\Sigma_2, N_2)$, a (Σ_2, N_2)-model (M_2, T_2) is reduced along (σ, f) to the (Σ_1, N_1)-model $(M_1, T_1) = (M_2, T_2)|_{(\sigma,f)}$ given by $M_1 = M_2|_\sigma$ and $T_1 = \hat{\sigma} \circ T_2 \circ f$. Here, $\hat{\sigma}$ maps an LTS for M_2 into an LTS for M_1 by extending the corresponding map between the alphabets, which by abuse of notation we also denote by $\hat{\sigma}$. $\hat{\sigma}\colon A(\beta(M_2)) \uplus \dot{\tau} \longrightarrow A(\beta(M_1)) \uplus \dot{\tau}$ is defined by

$$\hat{\sigma}(\dot{\tau}) = \dot{\tau}$$

$$\hat{\sigma}([(s,a)]) = \begin{cases} [(s',a)], & \text{if } \sigma(s') = s \\ \dot{\tau}, & \text{if } s \text{ is not in the image of } \sigma \end{cases}$$

Well-definedness, i.e. independence on the choice of s, a and s', can be shown as follows. Let $(s_1, a_1) \sim (s_2, a_2)$, $\sigma(s_1') = s_1$ and $\sigma(s_2') = s_2$. We need to show $(s_1', a_1) \sim (s_2', a_2)$. If $(s_1, a_1) \sim (s_2, a_2)$ because $a_1 = a_2 = \bot$ and s_1 and s_2 have a common supersort, then by non-extension of σ, s_1' and s_2' have a common supersort as well, and we are done. Otherwise, $a_1 \neq \bot \neq a_2$, s_1 and s_2 have a common supersort, and a_1 and a_2 are equalized by injections into common supersorts of s_1 and s_2. By non-extension of σ, this carries over to s_1' and s_2'.

Homomorphism reducts are inherited from CASL.

Sentences A (Σ, N)-sentence is either a CASL Σ-sentence or a system of equations $(n_i = P_i)_{i=1,\ldots,n}$, where for each $i = 1, \ldots, n$, $n_i \in N$ and P_i is a CSP-process term over Σ involving process names from N, where Σ-terms are used as communications, Σ-sorts denote sets of communications, relational renaming is described by a binary Σ-predicate and Σ-formulae occur in the conditional (cf. Section 2.3 of [Rog]).

Sentence translation Translation of CASL sentences is as in CASL. An equation system $(n_i = P_i)_{i=1,\ldots,n}$ is translated via (σ, f) to $(f(n_i) = (\sigma, f)(P_i))_{i=1,\ldots,n}$, where $(\sigma, f)(P)$ is obtained by renaming the components of P according to σ and process names according to f.

Satisfaction A model (M, T) satisfies a CASL sentence φ iff $M \models \varphi$ in the CASL institution. It satisfies a sentence $(n_i = P_i)_{i=1,\ldots,n}$ iff $T(n) = [\![P]\!]_{\emptyset:\emptyset \longrightarrow \beta(M),T}]\!]_{CSP}$, and with $[\![P]\!]_{\emptyset:\emptyset \longrightarrow \beta(M),T}]\!]_{CSP}$ being the LTS semantics of CSP, while process names from N interpreted via T (hence the additional dependence on T). This is similar to the trace semantics $[\![P]\!]_{\emptyset:\emptyset \longrightarrow \beta(M)}]\!]_{CSP}$ in [Rog], but needs to be adapted to the LTS semantics [Ros97].

The satisfaction condition for CASL sentences follows from that of the CASL institution. The satisfaction condition for sentences of form $n = P$, based on [Rog] is seen as follows: Given a signature morphism $(\sigma, f)\colon (\Sigma_1, N_1) \longrightarrow (\Sigma_2, N_2)$ and a (Σ_2, N_2)-model (M_2, T_2),

$$\begin{array}{ll} & (M_2, T_2) \models (\sigma, f)(n = P) \\ \text{iff} & (M_2, T_2) \models f(n) = (\sigma, f)(P) \\ \text{iff} & T_2(f(n)) = [\![(\sigma, f)(P)]\!]_{\emptyset:\emptyset \longrightarrow \beta(M_2),T_2}]\!]_{CSP} \\ \text{iff} & \hat{\sigma}(T_2(f(n))) = \hat{\sigma}([\![(\sigma, f)(P)]\!]_{\emptyset:\emptyset \longrightarrow \beta(M_2),T_2}]\!]_{CSP}) \quad \text{(Lemma 3.49)} \\ \text{iff} & \hat{\sigma}(T_2(f(n))) = [\![P]\!]_{\emptyset:\emptyset \longrightarrow \beta(M_2|_\sigma),\hat{\sigma} \circ T_2 \circ f}]\!]_{CSP} \quad \text{(Lemma 3.48)} \\ \text{iff} & (M_2|_\sigma, \hat{\sigma} \circ T_2 \circ f) \models n = P \\ \text{iff} & (M_2, T_2)|_{(\sigma,f)} \models n = P. \end{array}$$

Again, this needs to be adapted to the LTS semantics.

Lemma 3.48 $\hat{\sigma}([\![(\sigma, f)(P)]\!]_{\emptyset:\emptyset \longrightarrow \beta(M_2),T_2}]\!]_{CSP}) = [\![P]\!]_{\emptyset:\emptyset \longrightarrow \beta(M_2|_\sigma),\hat{\sigma} \circ T_2 \circ f}]\!]_{CSP}$

PROOF: Straightforward induction over the structure of P. □

Lemma 3.49 If $\hat{\sigma}(\mathcal{T}) = \hat{\sigma}([\![(\sigma, f)(P)]\!]_{\emptyset:\emptyset \longrightarrow \beta(M_2),T_2}]\!]_{CSP})$,
then $\mathcal{T} = [\![(\sigma, f)(P)]\!]_{\emptyset:\emptyset \longrightarrow \beta(M_2),T_2}]\!]_{CSP}$,

PROOF: Since $\hat{\sigma}$ is extended pointwise to trace sets, it suffices to consider its action on alphabet letters. Since $[\![(\sigma, f)(P)]\!]_{\emptyset:\emptyset \longrightarrow \beta(M_2)}]\!]_{CSP}$ only involves sorts in the image of σ, we never enter the case that $\hat{\sigma}$ yields $\dot{\tau}$. Hence, it suffices to show that $\hat{\sigma}([(a_1, s_1)]) = \hat{\sigma}([(a_2, s_2)]) \neq \dot{\tau}$ implies $[(a_1, s_1)] = [(a_2, s_2)]$. Assume $\hat{\sigma}([(a_1, s_1)]) = \hat{\sigma}([(a_2, s_2)]) \neq \dot{\tau}$, i.e. $[(a_1, s_1')] = [(a_2, s_2')]$ for some s_1', s_2' with $\sigma(s_1') = s_1$ and $\sigma(s_2') = s_2$. The rest of the argument is analogous to that showing well-definedness of $\hat{\sigma}$ in the definition of reducts above (note that here we do not even need non-extension of σ, but only the trivial fact that σ preserves common supersorts). □

3.6 Bibliographical Notes

3.6.1 CASL

"CASL, the Common Algebraic Specification Language, has been designed by an open collaborative initiative called the *The Common Framework Initiative*, CoFI.[8] The rationale behind this initiative was that the lack of a common framework for algebraic specification and development of software was a major hindrance for the dissemination and application of algebraic specification techniques. In particular, there was a proliferation of specification languages – some differing in only quite minor ways from each other. The major languages include ACT ONE/ACT TWO [CEW93], ASF [BHK89], ASL [Wir86], CLEAR [BG80], EXTENDED ML [KST97], LARCH [GH93], OBJ3 [GWM+92], PLUSS [BGM89], and SPECTRUM [BFG+93]. This abundance of languages was an obstacle for the adoption of algebraic methods for use in industrial contexts, making it difficult to exploit standard examples, case studies and training material." [BM04]

The outcome of this effort has been the design of the language CASL, and documented in a survey paper [ABK+], the CASL user manual [BM04] and reference manual [CoF04].

The CASL institution has been designed by the CoFI language task group during various meetings, based on numerous study notes and discussions. The subsorted institution has been reported in [CHKBM97, Mos02]. Cocompleteness of the CASL signature category has been proved by the author in [Mos98]. Failure of amalgamation for the CASL institution has been discovered by Maura Cerioli and was first reported in [SMT01a].

3.6.2 MODALCASL

MODALCASL has been designed by the author, following the literature about multi-modal and temporal logics [CH95, Sch04, Pop94, BRV01, AGM92]. The notation $[*]$ is due to [BRV01] (they call it "master modality"). The way of combining multi-modal logic with first-order logic and CTL* seems to be new and is (even in its CTL-fragment) more expressive than first-order CTL as studied e.g. in [BDG+98]. Note that we do not assume a special program syntax for the generation of Kripke structures as e.g. in first-order dynamic logic [Har79] — which does not preclude the *specification* of such programs. However, we also want to open to generation of Kripke structures by completely different institutions, like CCS or CSP (see also Sect. 3.5).

We have chosen constant domains in MODALCASL. We do not allow for cumulative domains (the latter are used e.g. in [Tha00]): firstly, we could not identify constant domains as a sublanguage (e.g. the Barcan formulae etc. are too weak to characterize this), and secondly, constant domains are more expressive (cumulative domains can be simulated using an existence predicate).

It would be nice to include non-normal modal logics as well, which requires replacing Kripke semantics by e.g. neighbourhood semantics, but at present it is not entirely clear how this is best integrated with the path formulas of CTL^*.

[8] CoFI is pronounced like 'coffee'.

CASL-LT [RAC00] is a CASL extension that is also based on CTL^*. However, its syntax deviates much from the usual CTL^* syntax, and the Kripke transition relation is made explicit through a ternary predicate. We leave transition relation implicit as standard in modal logic, but can make it explicit, if needed, by translation to CASL (see Sect. 4.2). Moreover, since we work with natural numbers as paths indices (in contrast the semantics of CASL-LT, where paths are cut down), we can include past temporal operators quite naturally. Although in principle, past operators are not necessary, their omission may lead to an exponential blow up for the size of formulas [Sch04]. The CASL-LT formulas $\mathit{first\_state}(x, \varphi)$ and $\mathit{first\_label}(x, \varphi)$ involving a binding structure can be simulated in MODALCASL.

Craig interpolation for various logics, including modal logics, is discussed in the forthcoming book [GM]. This book should also make clear how much simpler it is to rely on amalgamation instead of interpolation properties, as we do in Chaps. 5 and 6.

3.6.3 COCASL

The relatively young field of coalgebra has successfully dualized[9] many notions and results of classical universal algebra. This has been applied to the specification of reactive and object oriented systems. See [JR97, Rut00, Jac02, Rei95, Kur01a] for an introduction and further references. COCASL is a combination of algebraic and coalgebraic specification. The presentation here follows largely [MSRR], which provides more details and further examples.

A modal logic for coalgebra has been developed in the seminal paper [Mos99]. However, it is not immediately suitable for use in a specification language due to the presence of infinitary conjunction and the complex nature of its modal operator. The syntax chosen here is largely in the spirit of [Jac00, Kur01b] in that modalities are indexed by observer terms. The syntax of [Jac01], inherited from [Rößi00], differs in that it uses instead modal operators built along the structure of the signature functor, plus a single modality for the coalgebra structure. For the functors covered in [Jac01], this choice does not affect expressivity at the level of state formulae. (The syntax of [Jac01] allows formulating modal statements also at the level of the functor ingredients such as products and sums; however, the main interest is still in state formulae.) The syntax of the modal operators in CCSL [RTJ01], in turn, deviates from the others in that state variables are kept explicit; moreover, CCSL has an explicit bisimilarity relation (which can be emulated in COCASL using cogeneratedness constraints). Among the pre-existing modal logics for coalgebra, [Jac01] is unique (along with COCASL, of course) in admitting several non-observable sorts. Iterative modalities as in COCASL are otherwise found only in CCSL.

COCASL is more expressive than other algebra-coalgebra combinations in the literature: [Cîr02] uses a simpler logic, CCSL [RTJ01] has fewer datatypes available, while hidden algebra such as in BOBJ [Roş00] and reachable-observable algebra such as in COL [BHb] do not support cofree types. If, for example, streams are not specified as the final (=cofree) model, then there are stream models which do not contain all corecursively definable functions (like the flipping of streams), so that corecursive definitions fail to be conservative.

By contrast, cofree cotypes in COCASL support a style of specification separating the basic process type (with its data sorts, observers and other operations) from further, derived operations defined on top of this in a conservative way. Note that this is not a purely theoretical question: programming languages such Charity [CF92] and Haskell [PJ03] support infinite data structures that correspond to the infinite trees in the behaviour algebras, and one should be able to specify that as many infinite trees as needed for all programs over some data structure expressible in these languages are present in the models of a specification. The Haskell semantics for lazy data structures (at least for the non-left-$\rightarrow$-recursive case) indeed comprises *all* infinite trees, i.e. is captured exactly by a behaviour algebra.

The institution of *Constructor-based Observational Logic (COL)* [BHb] combines reachability induced by constructors with observational equality induced by observers. COCASL does not directly support observational equality or bisimilarity, but full abstractness ('bisimilarity is equality')

[9] in the category-theoretic sense

can be expressed via cogeneration constraints, as shown in the process algebra examples. In COL, observability is a global notion and required to be preserved and reflected by signature morphisms. CoCasl's local notion of observability provides an extra degree of flexibility — in particular, it allows instantiating observable sorts with non-observable ones. Unlike COL, CoCasl does not simultaneously support a glass-box and a black-box view on a specification. However, we plan to develop a notion of behavioural refinement between CoCasl specifications. Then, the black-box/glass-box view of [BHb] could be expressed in CoCasl as a refinement of a black-box specification into a glass-box one, thus also providing a clear separation of concerns.

The *Coalgebraic Class Specification Language* CCSL [RTJ01], developed in close cooperation with the LOOP project [vdBJ01], is based on the observation of [Rei95] that coalgebras can give a semantics to classes of object-oriented languages. CCSL provides a notation for parameterized class specifications based on final coalgebras. Its semantic is based on a higher-order equational logic and it provides theorem proving support by compilers that translate CCSL into the higher-order logic of PVS and Isabelle. In its current version, CCSL does not support data type specifications with partial constructors, axioms or equations, i.e. it only supports free types without axioms in the sense of Casl.

3.6.4 HasCasl

The language HasCasl has been introduced in [SM02, SMM] as a higher order extension of Casl, based on the partial λ-calculus [Mog86, Mog88, Ros86]. For more details on both syntax and semantics, see [SM02, SMM].

Support for reasoning about side-effects encapsulated via monads has been introduced in [SM, SM03] by providing a monad-independent Hoare calculus and a monad-independent dynamic logic; this work is extended in [SM04] to cover partial and total correctness of abruptly terminating programs. For all these results, the logic of HasCasl is used as a background formalism.

3.6.5 CspCASL

Csp [Hoa85, Ros97] is a process algebra that has been studied now for 25 years. Still, a formal integration with datatypes has been provided only recently: The foundations of Csp-Casl have been laid out in [Rog]. Very promising is the recent encoding of Csp in Isabelle/HOL [IM05], which improves an existing encoding and will be the basis for proof support for Csp-Casl.

Future work will also consider combinations of Casl and CCS [All01].

Chapter 4

An Initial Logic Graph

In this chapter, we define a variety of translations (formalized as institution morphisms, comorphisms, etc.) between the institutions defined in the previous chapter, thus obtaining a logic graph. The logic graph will also consists of modifications between the translations, expressing that some (composites of) translations are the same. It is expected that this initial logic graph will be extended with more and more logics, translations and modifications as time goes by. We think that the best strategy is to integrate the needed logics into one big graph, instead of having several ones (and therefore also several resulting heterogeneous languages).

4.1 Comorphisms Among Subinstitutions of CASL

In order to relate subinstitutions of CASL, we relate their underlying institutions. We therefore use the notion of institution comorphisms (also called simple maps of institutions) [Mes89b, Tar96] introduced in Sect. 2.

4.1.1 The First-Order Level

The diagram in Fig. 4.1 shows that the first-order subinstitutions of CASL have all the same expressiveness, except from FOL, which is a bit weaker ($FOL^=$ can be represented in FOL only with a comorphism admitting model expansion, but not as a subinstitution). Some of the arrows are labeled with numbers in brackets; this refers to later subsections where the corresponding comorphisms are described in detail. Obvious subinstitution comorphisms are not labeled.

Another diagram, shown in Fig. 4.2, can be obtained by adding a "C" to each institution in Fig. 4.1.

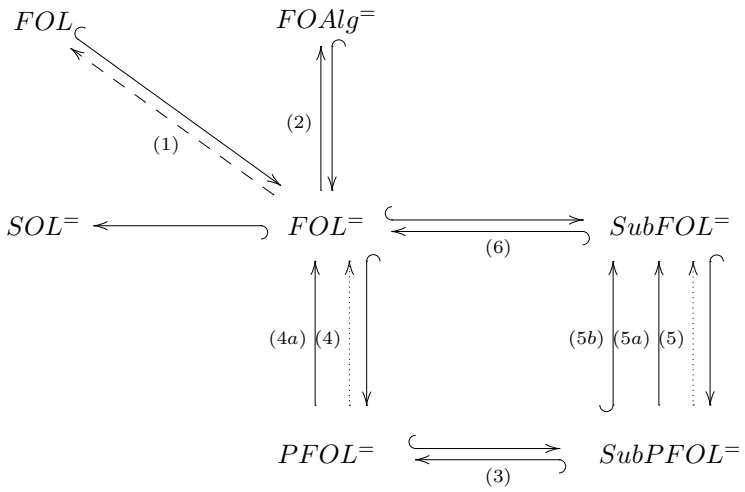
(1)**: Mapping $FOL^=$ to FOL**

The idea here is simply to replace equality by a congruence relation.

Signatures A $FOL^=$-signature Σ is mapped to the FOL−presentation $\Phi(\Sigma)$ that extends Σ by adding predicate symbols $\equiv$: $s \times s$ (overloaded for each sort s) that are axiomatized to be a congruence (also w.r.t. the predicates).

Models A model M is translated by factoring its carriers w.r.t. $\equiv$. The functions and predicates can act on the equivalence classes because $\equiv$ is a congruence. A homomorphism $h\colon M \longrightarrow M'$ is translated to $\bar{h}$ with $\bar{h}([a]) = [h(a)]$ (where $[a]$ is the congruence class of a). This is well-defined since $a \equiv_M a'$ implies $h(a) \equiv_{M'} h(a')$ (predicates are preserved by homomorphisms).

Sentences Sentence translation is done by replacing $=$ by $\equiv$.



Index for arrow types:

1. Admits model expansion
2. Strongly persistently liberal
3. Model-bijective
4. Subinstitution

Figure 4.1: The first-order level

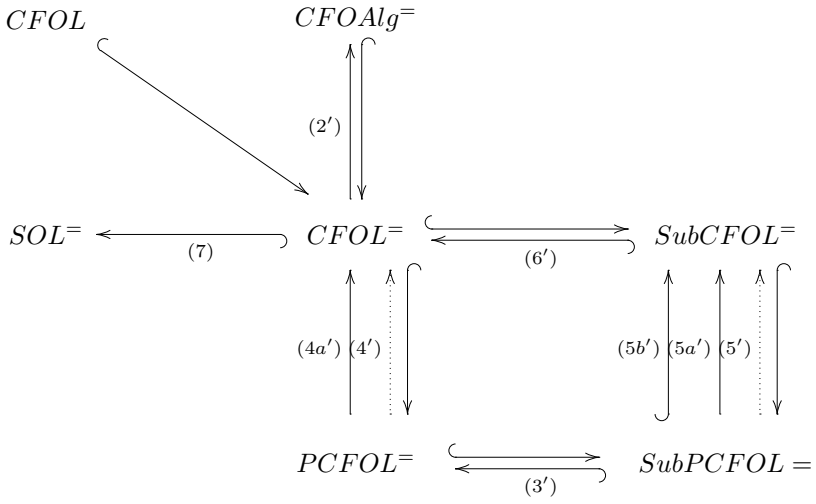


Figure 4.2: The first-order level with sort generation constraints (index: see Fig. 4.1).

Satisfaction To prove the satisfaction condition, define a mapping on valuations as follows: Given a $\Phi(\Sigma)$-model M' and a valuation $\nu: X \longrightarrow M'$, define $\beta(\nu): X \longrightarrow \beta_\Sigma(M')$ by

$$\beta(\nu)_s(x) := [\nu(x)] \text{ for } x \in X_s.$$

We then have

$$\beta(\nu)^\#(t) = [\nu^\#(t)],$$

which can be easily proved by induction over t, using the congruence axioms. By induction over $\varphi \in \mathbf{Sen}(\Sigma)$, we can now show that

$$\nu \Vdash \alpha_\Sigma(\varphi) \text{ iff } \beta(\nu) \Vdash \varphi.$$

Concerning e.g. strong equations, we have $\nu \Vdash \alpha_\Sigma(t_1 = t_2)$ iff $\nu \Vdash t_1 \equiv t_2$ iff $\nu^\#(t_1) \equiv_{M'} \nu^\#(t_2)$ iff $[\nu^\#(t_1)] = [\nu^\#(t_2)]$ iff $\beta(\nu)^\#(t_1) = \beta(\nu)^\#(t_2)$ iff $\beta(\nu) \Vdash t_1 = t_2$.

Concerning predicate applications, $\nu \Vdash \alpha_\Sigma(p(t_1, \ldots, t_n))$ iff $\nu \Vdash p(t_1, \ldots, t_n)$ iff $(\nu^\#(t_1), \ldots, \nu^\#(t_n)) \in P_{M'}$ iff (by the congruence axiom for p) $([\nu^\#(t_1)], \ldots, [\nu^\#(t_n)]) \in P_{\beta_\Sigma(M')}$ iff $((\beta(\nu))^\#(t_1), \ldots, (\beta(\nu))^\#(t_n)) \in P_{\beta_\Sigma(M')}$ iff $\beta(\nu) \Vdash p(t_1, \ldots, t_n)$.

Concerning quantification, $\nu \Vdash \forall x : s \bullet \alpha_\Sigma(\psi)$ iff for all $\xi: X \cup \{x : s\} \longrightarrow M'$ extending ν on $X \setminus \{x : s\}$, $\xi \Vdash \alpha_\Sigma(\psi)$ iff (by induction hypothesis) for all $\xi: X \cup \{x : s\} \longrightarrow M'$ extending ν on $X \setminus \{x : s\}$, $\beta(\xi) \Vdash \varphi$ iff for all $\rho: X \cup \{x : s\} \longrightarrow \beta_\Sigma(M')$ extending $\beta(\nu)$ on $X \setminus \{x : s\}$, $\rho \Vdash \varphi$ iff $\beta(\nu) \Vdash \forall x : s \bullet \varphi$.

The other cases are treated similarly. The satisfaction condition now follows by noting that β is surjective on valuations. □

The comorphism can be seen to admit model expansion as follows: Given a Σ-model M in $FOL^=$, turn it into a $\Phi(\Sigma)$-model by letting $\equiv$ be interpreted as the identity relation. This model is a pre-image of M under the model translation. However, the comorphism is not persistently liberal: for example, let Σ consist of just one sort, let M be the Σ-model consisting of two points, and let M' be the $\Phi(\Sigma)$-model consisting of two equivalent points. Then there is just one homomorphism from M to $\beta_\Sigma(M')$. If the comorphism were persistently liberal, there would have to be just one homomorphism from $\gamma_\Sigma(M)$ to M' as well. But this is impossible, since then $\gamma_\Sigma(M)$ would have to be empty (even if empty carriers were allowed, this still would contradict the requirement $\beta_\Sigma(\gamma_\Sigma(M)) = M$).

Note that the comorphism cannot be generalized to the level of sort generation constraints in a straightforward way, because these are not preserved along taking quotients: Let Nat be the signature consisting of a sort Nat and two total function symbols $0 : Nat$ and $suc: Nat \longrightarrow Nat$. Let M' be the $\Phi(Nat)$-model consisting of two copies of the natural numbers, which are identified by the congruence relation. Since the constant 0 yields the zero only of one copy of the naturals, M is not term-generated. However $\beta_\Sigma(M)$ consist of just one copy of the naturals and therefore is term-generated.

(2) and (2′): Mapping $(C)FOL^=$ to $(C)FOAlg^=$

Here, the idea is to replace predicates by Boolean-valued functions. Note that the Booleans can be axiomatized monomorphically in $FOAlg^=$.

Signatures A signature is mapped by adding the presentation Bool and replacing each predicate symbol $p : w$ by a total function symbol $f_p: w \longrightarrow Bool$

spec Bool =
  sort *Bool*
  ops *True*, *False* : *Bool*
  forall b : *Bool*
  • $b = True \vee b = False$

- $\neg\ True\ =\ False$

end

Strictly speaking, BOOL has to be renamed in order to become disjoint with the signature.

Models A model is translated by replacing each *Bool*-valued function by the corresponding predicate.

Sentences Sentence translation is done by replacing atomic formulas

$$p_w(t_1, \ldots, t_n)$$

by

$$(f_p)_{w,Bool}(t_1, \ldots, t_n) = True_{Bool}.$$

Sort generation constraints are left unchanged.

Satisfaction The satisfaction condition is proved in a straightforward way. □

It is also straightforward to show that this gives a model-bijective institution comorphism. Note, however, that we do *not* get a subinstitution comorphism. This is because homomorphisms in $FOL^=$ need to preserve just truth (but not falsehood) of predicates, while homomorphisms in the comorphism in $FOAlg^=$ need to preserve both truth *and* falsehood of (represented) predicates.

Indeed, we conjecture that there cannot be a subinstitution comorphism from $FOL^=$ to $FOAlg^=$ since $FOL^=$ is strictly more expressive than $FOAlg^=$ w.r.t. a construct that actually exploits the homomorphisms: with the **free** construct, it is possible to express the transitive closure of an arbitrary (parameter) relation, while we conjecture that this is not possible in $FOAlg^=$.

(3) and (3′): Mapping $SubP(C)FOL^=$ to $P(C)FOL^=$

The translation of $SubP(C)FOL\ =$ to $P(C)FOL\ =$ is trivial, since $SubPCFOL^=$ is defined in terms of $PCFOL^=$:

Signatures A signature Σ is mapped to the presentation $(\hat{\Sigma}, \hat{J}(\Sigma))$.

Models Model translation is the identity.

Sentences Sentence translation is the identity.

Satisfaction The satisfaction condition follows immediately. □

This trivially gives a subinstitution comorphism.

(4) and (4a): Mapping $PFOL^=$ to $FOL^=$

A translation of $PFOL^=$ into $FOL^=$ is described in [CMR99]. We refine this translation here. The main idea is to use a definedness predicate to divide each carrier into "defined" and "undefined" elements. The "defined" elements represent ordinary values, while the "undefined" elements all represent the undefined. Partial functions thus can be totalized: they possibly yield an "undefined" element. We specify that there is at least one "undefined" element $\bot$; however, it may be not the only one.

Signatures A $PFOL^=$-signature $\Sigma = (S, TF, PF, P)$ is translated to a $FOL^=$-presentation having the signature

$$Sig(\Phi(\Sigma)) = (S, TF \uplus PF \uplus \{\bot : s \mid s \in S\}, P \uplus \{D : s \mid s \in S\})$$

and the set of axioms $Ax(\Phi(\Sigma))$:

$\exists x : s \bullet D_s(x)$	$s \in S$	(1)
$\neg D_s(\bot_s)$	$s \in S$	(2)
$D_s(f(x_1, \ldots, x_n)) \Leftrightarrow \bigwedge_{i=1..n} D_{s_i}(x_i)$	$f : s_1, \ldots, s_n \to s \in TF$	(3)
$D_s(g(x_1, \ldots, x_n)) \Rightarrow \bigwedge_{i=1..n} D_{s_i}(x_i)$	$g : s_1 \ldots s_n \to ? \; s \in PF$	(4)
$p(x_1, \ldots, x_n) \Rightarrow \bigwedge_{i=1..n} D_{s_i}(x_i)$	$p\colon s_1 \ldots s_n \in P$	(5)

D plays the role of a definedness predicate: the elements inside D are called "defined", those outside D are called "undefined". The axioms in the signature translation state that there is at least one "defined" element (1), that $\bot$ is an "undefined" element (2), total functions are indeed total (3) and all functions ((3), (4)) and predicates (5) are strict.

A signature morphism σ is translated to a presentation morphism $\Phi(\sigma)$ which acts as σ on those parts of $\Phi(\Sigma)$ being included from Σ, while it maps the added structure for a signature component to the added structure for a mapped component.

Models A $\Phi(\Sigma)$-structure M (in $FOL^{=}$) is translated to the partial Σ-structure $\beta_\Sigma(M) = M'$ (in $PFOL^{=}$) with

- $M'_s = (D_s)_M$ for $s \in S$,
- $f_{M'}$ is f_M restricted to M'_w for $f : w \to s \in TF$ (this is a total function by (3)),
- $g_{M'}(a_1, \ldots, a_n) = \begin{cases} g_M(a_1, \ldots, a_n), & \text{if } g_M(a_1, \ldots, a_n) \in M'_s \\ \text{undefined}, & \text{otherwise} \end{cases}$ for $g : w \to ? \; s \in PF$,
- $p_{M'} = p_M \cap M'_w$ for $p\colon w \in P$.

Homomorphisms are translated by restricting them to the carriers of M'. This is well-defined since predicates are preserved by homomorphisms.

Sentences The sentence translation keeps the structure of the sentences and maps strong and existential equality to appropriate circumscriptions using the definedness predicate D. Definedness is mapped to D, and quantifiers are relativized to the set of all defined elements.

Formally, a Σ-sentence φ (in $PFOL^{=}$) is translated to the $\Phi(\Sigma)$-sentence $\alpha_\Sigma(\varphi)$:

$\alpha_\Sigma(def(t)) = D_s(t)$	$\alpha_\Sigma(t_1 = t_2) = ((D_s(t_1) \vee D_s(t_2)) \Rightarrow t_1 = t_2)$
$\alpha_\Sigma(\varphi \wedge \psi) = \alpha_\Sigma(\varphi) \wedge \alpha_\Sigma(\psi)$	$\alpha_\Sigma(t_1 \stackrel{e}{=} t_2) = t_1 = t_2 \wedge D_s(t_1)$
$\alpha_\Sigma(p(t_1, \ldots, t_n)) = p(t_1, \ldots, t_n)$	$\alpha_\Sigma(F) = F$
$\alpha_\Sigma(\varphi \Rightarrow \psi) = \alpha_\Sigma(\varphi) \Rightarrow \alpha_\Sigma(\psi)$	$\alpha_\Sigma(\forall x : s \bullet \varphi) = \forall x : s \bullet D_s(x) \Rightarrow \alpha_\Sigma(\varphi)$

Satisfaction To prove the satisfaction condition, we need to talk about partial variable valuations. This is necessary since a valuation in a $\Phi(\Sigma)$-model M' may assign an "undefined" element to a variable. In the Σ-model $M = \beta_\Sigma(M')$, this should correspond to a truly undefined variable. Thus, we consider *partial* variable valuations $\nu\colon X \to ? \; M$. The evaluation $\nu^\#$ of terms and the satisfaction $\nu \Vdash$ of formulas is defined as before (see Sect. 3.1.1), with the following two exceptions:

- $\nu^\#_s(x) = \begin{cases} \nu(x), & \text{if this is defined} \\ \text{undefined}, & \text{otherwise} \end{cases}$ for all $x \in X_s$ and all $s \in S$;
- $\nu \Vdash_\Sigma (\forall x : s \bullet \varphi)$ iff for all valuations $\xi\colon X \cup \{x : s\} \longrightarrow M$ which extend ν on $X \setminus \{x : s\}$ *and are defined on* $x : s$, we have $\xi \Vdash_\Sigma \varphi$.

Note that the new definitions of $\nu^\#$ and $\Vdash$ coincide with the old ones for total valuations. To achieve this, it is crucial to require the extended valuation ξ to be defined on x in the clause for satisfaction of quantified formulas above.

We now have the following lemma:

Lemma 4.1 For each Σ-formula φ and each (total) valuation $\nu: X \longrightarrow M'$ into a $\Phi(\Sigma)$-model M', define the partial valuation $\rho: X \longrightarrow \beta_\Sigma(M')$ to be

$$\rho(x) = \begin{cases} \nu(x), & \text{if } \nu(x) \in (D_s)_{M'} \\ \text{undefined}, & \text{otherwise} \end{cases} \quad (x \in X_s)$$

Then we have

(a) $\rho^\#(t)$ is defined iff $\nu^\#(t) \in (D_s)_{M'}$ for all Σ-terms t of sort s.

(b) In the case that one of the sides of (1) holds, we have

$$\rho^\#(t) = \nu^\#(t).$$

(c) $\rho \Vdash^{PFOL^=}_\Sigma \varphi$ iff $\nu \Vdash^{FOL^=}_{Sig(\Phi(\Sigma))} \alpha_\Sigma(\varphi)$.

PROOF: (a) and (b): By induction over the structure of t. For variables, we need just use the definition of ρ. For applications of total or partial function symbols, use axioms (3) or (4) in $\Phi(\Sigma)$.

(c): By induction over the structure of φ. Considering existence equations, we have

$\rho \Vdash^{PFOL^=}_\Sigma t_1 \stackrel{e}{=} t_2$
iff $\rho^\#(t_1)$ and $\rho^\#(t_2)$ are defined and equal
iff $\nu^\#(t_1) \in (D_s)_M$ and $\nu^\#(t_2) \in (D_s)_M$ and $\nu^\#(t_1) = \nu^\#(t_2)$
iff $\nu \Vdash^{FOL}_{Sig(\Phi(\Sigma))} t_1 = t_2 \wedge D_s(t_1)$
iff $\nu \Vdash^{FOL}_{Sig(\Phi(\Sigma))} \alpha_\Sigma(t_1 \stackrel{e}{=} t_2)$.

Considering universally quantified formulas, we have

$\rho \Vdash^{PFOL}_\Sigma \forall x : s \bullet \varphi$
iff for all $\xi: X \cup \{x : s\} \longrightarrow \beta_\Sigma(M')$ extending ρ on $X \setminus \{x : s\}$ and being defined on $\{x : s\}$, $\xi \Vdash^{PFOL}_\Sigma \varphi$
iff for all $\tau: X \cup \{x : s\} \longrightarrow M'$ extending ν on $X \setminus \{x : s\}$ for which $x \in (D_s)_{M'}$, $\tau \Vdash^{FOL}_{Sig(\Phi(\Sigma))} \alpha_\Sigma(\varphi)$
iff $\nu \Vdash^{FOL}_{Sig(\Phi(\Sigma))} \forall x : s \bullet D_s(x) \Rightarrow \alpha_\Sigma(\varphi)$
iff $\nu \Vdash^{FOL}_{Sig(\Phi(\Sigma))} \alpha_\Sigma(\forall x : s \bullet \varphi)$.

The other cases are treated similarly. □

The satisfaction condition can now be shown as follows. Let φ be a Σ-sentence and $\nu: \emptyset \longrightarrow M'$ and $\rho: \emptyset \longrightarrow \beta_\Sigma(M')$ be the unique empty valuations. Then

$\beta_\Sigma(M') \models^{PFOL^=}_\Sigma \varphi$
iff $\rho \Vdash^{PFOL^=}_\Sigma \varphi$
iff $\nu \Vdash^{FOL^=}_{Sig(\Phi(\Sigma))} \alpha_\Sigma(\varphi)$
iff $M' \models^{FOL^=}_{Sig(\Phi(\Sigma))} \alpha_\Sigma(\varphi)$. □

The model translation can be easily shown to be surjective: For a partial Σ-structure M, form its *one-point completion* by just adding one element, $*$, to all carriers, which is the interpretation of $\bot$. The functions map $*$ to itself and behave as in M otherwise, where undefinedness of partial functions is mapped to $*$. Predicates are false on $*$. The predicate D is true everywhere except on $*$. Thus, our institution comorphism admits model expansion. Moreover, with the argument of Example 2.23, the comorphism also has the weak $\mathcal{D}$-amalgamation property, where $\mathcal{D}$ is the class of all injective signature morphisms.

Proposition 4.2 The above constructed institution comorphism is strongly persistently liberal.

PROOF: Define γ as follows. Put $(\gamma_\Sigma(M))_s := M_s \uplus \bar{M}_s$, where

- $\bar{M} \subseteq T_{Sig(\Phi(\Sigma))}(|M|)$ is the least sorted set satisfying
  - $\bot \in \bar{M}_s$
  - $f(a_1, \ldots, a_n) \in \bar{M}_s$ for $f \in TF_{w,s} \cup PF_{w,s}$, $w = s_1 \ldots s_n$, $(a_1 \ldots a_n) \in (M_w \uplus \bar{M}_w) \setminus dom f_M$
- $(D_s)_{\gamma_\Sigma(M)} := M_s$,
- $(\bot_s)_{\gamma_\Sigma(M)} := \bot$,
- $f_{\gamma_\Sigma(M)}(a_1, \ldots, a_n) := \begin{cases} f_M(a_1, \ldots, a_n), & \text{if } (a_1, \ldots, a_n) \in dom f_M \\ f(a_1, \ldots a_n), & \text{otherwise} \end{cases}$ for $f: w \longrightarrow s \in TF \cup PF$,
- $p_{\gamma_\Sigma(M)} = p_M$ for $p : w \in P$.

Now (1) of $\Phi(\Sigma)$ above is satisfied since the carriers of M are non-empty. (2) holds by definition of $(D_{\gamma_\Sigma(M)})_s$, and (3) and (4) hold by definition of $f_{\gamma_\Sigma(M)}$. (5) holds by definition of $p_{\gamma_\Sigma(M)}$.

Clearly, we have $\beta_\Sigma(\gamma_\Sigma(M)) = M$. To show the universal property of

$$M \xrightarrow{id} \beta_\Sigma(\gamma_\Sigma(M)) \ ,$$

let $h: M \longrightarrow \beta_\Sigma(N)$ be a homomorphism. Since $\beta_\Sigma(N)_s \subseteq N_s$, we can define $h^\# : \gamma_\Sigma(M) \longrightarrow N$ by

$$h^\#_s(a) = \begin{cases} h(a), & \text{if } a \in M_s \\ \nu^\#(a), & \text{if } a \in \bar{M}_s \end{cases}$$

where $\nu: |M| \longrightarrow N$ is defined by $\nu(a) = h(a)$. The first case is determined by the requirement that $\beta_\Sigma(h^\#) = h$, while the second case is determined by the requirement that $h^\#$ is a homomorphism. Thus, $h^\#$ is the unique homomorphism from $\gamma_\Sigma(M)$ to N that is mapped to h under β_Σ. □

Concerning model-bijectivity, by adding to $\Phi(\Sigma)$ the sentences $\neg\ x = \bot_s \Rightarrow\ D_s(x)$ (for each $s \in S$), $\bot$ becomes the *unique* "undefined" element. With this, we get an institution comorphism $(4a)$ that is model-bijective. Moreover, the translation of strong equations can be simplified by translating them just to ordinary equations. However, the comorphism then is no longer persistently liberal.

$(4')$, $(4a')$ and $(4b')$: Mapping $PCFOL^=$ to $CFOL^=$

The comorphism $(4a')$ extends the model-bijective institution comorphism $(4a)$ described above. Due to the need to translate sort generation constraints, we must be able to generate also the "undefined" elements in the $CFOL^=$-models. In order to be able to correctly model generatedness w.r.t. to partial functions, we have to ensure generatedness of the "undefined" elements. This is ensured by the axioms $\neg\ x = \bot_s \Rightarrow\ D_s(x)$ (for each $s \in S$). Since then there is just one "undefined" element, namely $\bot$, the "undefined" elements are now term generated. However, this method leads to a loss of the persistent liberality of the comorphism.

Signatures Signatures are translated as in $(4a)$.

Models Models are translated as in $(4a)$.

Sentences First-order sentences are translated as in $(4a)$. A Σ-sort generation constraint $(\overset{\bullet}{S}, \overset{\bullet}{F}, \theta: \bar{\Sigma} \longrightarrow \Sigma)$ is translated to $(\overset{\bullet}{S}, \overset{\bullet}{F} \cup \{\bot_s : s | s \in S\}, \theta')$, where $\theta': \Phi(\bar{\Sigma}) \longrightarrow \Phi(\Sigma)$ is the extension of θ to $\Phi(\bar{\Sigma})$ that is defined by mapping the added symbols in $\Phi(\bar{\Sigma})$ to the corresponding added symbols in $\Phi(\Sigma)$.

Satisfaction The satisfaction condition for formulas is proved as in (4a). Concerning sort generation constraints, note that model translation just removes the interpretation of $\bot$ from the carriers. Since $\bot$ cannot occur in $\overset{\bullet}{F}$ and moreover all functions are strict, generatedness w.r.t. $\overset{\bullet}{F}$ in the model $\beta_\Sigma(M')$ (where interpretations of $\bot_s$ have been removed) is the same as generatedness w.r.t. $\overset{\bullet}{F} \cup \{\bot_s : s | s \in S\}$ in the model M' (where interpretations of $\bot_s$ are not removed). □

If one wants to have a strongly persistently liberal comorphism (call it $(4')$, since it extends (4)) also translating sort generation constraints, one can restrict sort generation constraints in $PCFOL^=$ to those including total function symbols only: in this case, the introduction of a unique undefined element is not necessary.

The same restriction of sort generation constraints in $PCFOL^=$ also has to be done if we want to extend $(4b)$ to $(4b')$: sort generation constraints can just be left as they are in this case.

(5) and (5a): Mapping $SubPFOL^=$ to $SubFOL^=$

These are the same comorphisms as (4) and $(4a)$, except that the subsorting relation (and the corresponding injection and projection functions and membership predicates in the models and sentences) have to be mapped as well; we can take the identity mapping in all cases.

Also, by modifying (5) (similar to the passage from (4) to $(4a)$), we get a comorphism $(5a)$ that is model-bijective.

$(5')$ and $(5a')$: Mapping $SubPCFOL^=$ to $SubCFOL^=$

Again, the translation $(5a')$ is very similar to that of the previous section. As in $(4a')$, due to the need to translate sort generation constraints, we must be able to generate also the "undefined" elements in the $SubCFOL^=$-models. In order to be able to do this, we again restrict the models to those with exactly one "undefined" element. This leads to a loss of persistent liberality of the comorphism.

As in $(4')$, we can restore strongly persistent liberality by a restriction of sort generation constraints in $SubPCFOL^=$ to those not involving partial function symbols: in this case, the extra axioms for the $\bot$ functions are not needed. We thus get a comorphism $(5')$.

(6) and (6$'$): Mapping $Sub(C)FOL^=$ to $(C)FOL^=$

Here we can use a restriction of the translation of $Sub(C)PFOL^=$ to $P(C)FOL^=$ described in Sect. 4.1.1 above, which leaves out the partial projection symbols and those axioms from $\hat{J}(\Sigma)$ involving these. There is one problem connected with this: the membership predicate was originally axiomatized using partial projections and definedness:

$$\forall x : s' \bullet \in^s_{s'} (x) \Leftrightarrow def\ \mathtt{pr}_{(s',s)}(x)$$

for $s \leq s'$. Now we cannot just leave out these axioms, since then the membership predicates could be interpreted in an unintended way. Therefore, these axioms have to be replaced by

$$\forall x : s' \bullet \in^s_{s'} (x) \Leftrightarrow \exists y \bullet \mathtt{inj}_{(s,s')}(y) = x$$

(7): Mapping $CFOL^=$ to $SOL^=$

$CFOL^=$ adds sort generation constraints to $FOL^=$. These cannot be expressed within $FOL^=$, but can be translated to induction schemes, which can be expressed in second-order logic with equality ($SOL^=$) by second-order quantification over predicates.

Signatures Signatures and signature morphisms are translated identically.

Models Models, model homomorphisms and reducts are translated identically.

Sentences First-order sentences are translated identically. For a sort generation constraint

$$(\overset{\bullet}{S}, \overset{\bullet}{F}, \theta\colon \bar{\Sigma} \longrightarrow \Sigma)$$

we assume without loss of generality that all the result sorts of function symbols in $\overset{\bullet}{F}$ occur in $\overset{\bullet}{S}$ (see the corresponding remark in Sect. 3.1.1 where sort generation constraints have been introduced). Let

$$\overset{\bullet}{S} = \{s_1;\ \ldots;\ s_n\}, \quad \overset{\bullet}{F} = \{f_1\colon s_1^1 \ldots s_{m_1}^1 \longrightarrow s^1;\ \ldots;\ f_1\colon s_1^k \ldots s_{m_k}^k \longrightarrow s^k\}$$

The sort generation constraint is now translated to the $SOL^{=}$-sentence

$$\forall P_{s_1} : pred(\theta(s_1)) \ldots \forall P_{s_n} : pred(\theta(s_n)) \\ \bullet\ (\varphi_1 \wedge \cdots \wedge \varphi_k) \Rightarrow \textstyle\bigwedge_{j=1,\ldots,n} \forall x : \theta(s_j) \bullet P_{s_j}(x)$$

where

$$\varphi_j = \forall x_1 : \theta(s_1^j), \ldots, x_{m_j} : \theta(s_{m_j}^j) \\ \bullet \left(\textstyle\bigwedge_{i=1,\ldots,m_j;\ s_i^j \in \overset{\bullet}{S}} P_{s_i^j}(x_i) \right) \Rightarrow P_{s^j}\ \big(\theta(f_j)(x_1, \ldots, x_{m_j})\big)$$

Satisfaction To prove the satisfaction condition, let a Σ-model M and a Σ-sort generation constraint $(\overset{\bullet}{S}, \overset{\bullet}{F}, \theta\colon \bar{\Sigma} \longrightarrow \Sigma)$ be given. Call an $\bar{S}$-sorted set $\bar{\mathcal{P}} \subseteq |M|_\theta|$ $(\overset{\bullet}{S}, \overset{\bullet}{F}, \theta)$-*closed* iff it is closed under the application of functions $\theta(f)_M$ with $f \in \overset{\bullet}{F}$ and the filling in of arbitrary values of $M_{\theta(s)}$ for $s \notin \overset{\bullet}{S}$.

Lemma 4.3 Let X be the variable system $\{P_{s_1} : pred(\theta(s_1)), \ldots, P_{s_n} : pred(\theta(s_n))\}$. Given an assignment $\xi\colon X \longrightarrow M$, consider the $\bar{S}$-sorted set $\bar{\mathcal{P}}(\xi)$ consisting of $\xi(P_s)$ for $s \in \overset{\bullet}{S}$ and of $M_{\theta(s)}$ for $s \notin \overset{\bullet}{S}$.
Then we have

$$\xi \Vdash \varphi_1 \wedge \cdots \wedge \varphi_k \text{ iff } \bar{\mathcal{P}}(\xi) \text{ is } (\overset{\bullet}{S}, \overset{\bullet}{F}, \theta)\text{-closed.}$$

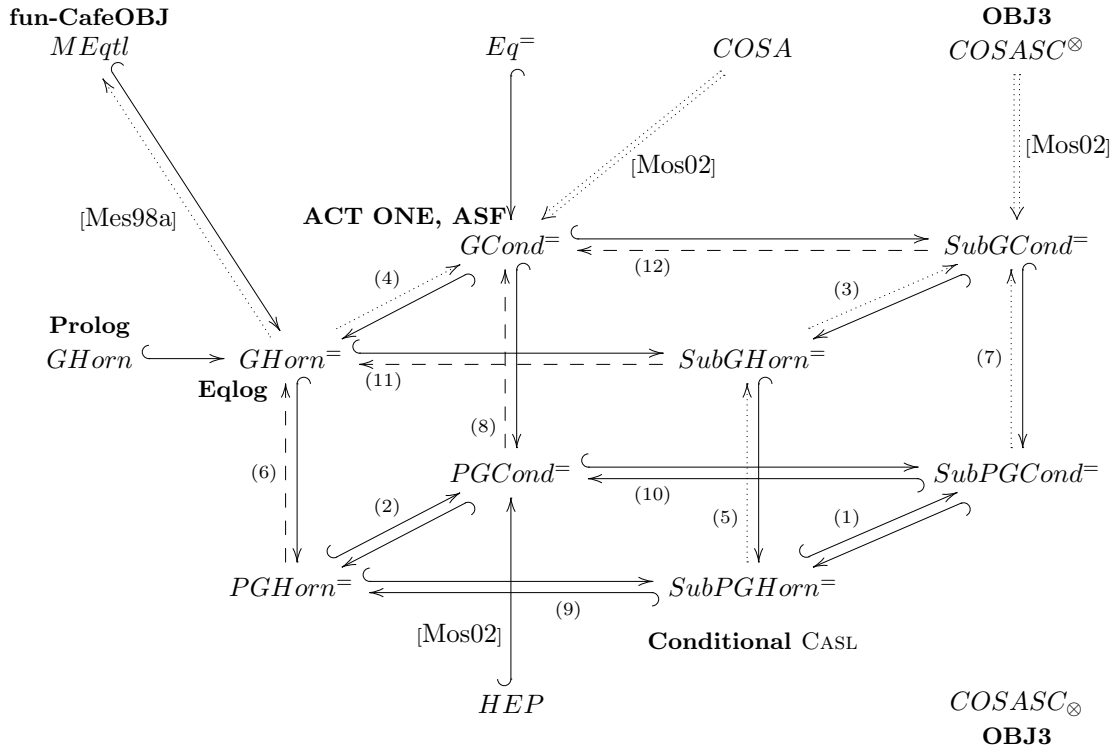
PROOF: This directly follows from the form of the φ_j. Note that the filling in of arbitrary values of $M_{\theta(s)}$ for $s \notin \overset{\bullet}{S}$ is captured by the condition $s_i^j \in \overset{\bullet}{S}$ in the conjunction in the premise of φ_j: for $s_i^j \notin \overset{\bullet}{S}$, nothing is required. □

Now M satisfies the sort generation constraint $(\overset{\bullet}{S}, \overset{\bullet}{F}, \theta\colon \bar{\Sigma} \longrightarrow \Sigma)$

iff the smallest $(\overset{\bullet}{S}, \overset{\bullet}{F}, \theta)$-closed set is $|M|_\theta|$
iff for all $\xi\colon X \longrightarrow M$, $\bar{\mathcal{P}}(\xi)$ is $(\overset{\bullet}{S}, \overset{\bullet}{F}, \theta)$-closed implies $\bar{\mathcal{P}}(\xi) = |M|_\theta|$
iff for all $\xi\colon X \longrightarrow M$,
$\xi \Vdash \varphi_1 \wedge \cdots \wedge \varphi_k$ implies $\xi \Vdash \bigwedge_{j=1,\ldots,n} \forall x : \theta(s_j) \bullet P_{s_j}(x)$
iff M satisfies the translation of the sort generation constraint. □

4.1.2 The Positive Conditional Level

At the positive conditional level (see Fig. 4.3), we have used generalized conditional logic ($GCond^{=}$, $GHorn^{=}$ etc.). The reason for this is that we want to be able to use equivalences (and not just implications) in the translations of sentences, which would not be possible within ordinary conditional logic ($Cond^{=}$, $Horn^{=}$ etc.). However, note that there is a *conjunctive* subinstitution comorphism



Index for arrow types:

1. Admits model expansion
2. Strongly persistently liberal
3. Model-bijective
4. Subinstitution

Figure 4.3: The positive conditional level

from any of the generalized conditional logics to the corresponding ordinary conditional logic. Note also that in most cases, the diagram remains the same if the G is deleted everywhere (only a few arrows become conjunctive comorphisms).

$MEqtl$ denotes Meseguer's Membership Equational Logic, see [Mes98a], where also the arrows to and from $MEqtl$ are described. Also, Equational Type Logic [MSS90] and Unified Algebras [Mos89] and similar frameworks could be added at the same corner of the diagram (however, the arrow from $GHorn^{=}$ will not be persistently liberal in all these cases).

At the positive conditional level, we have the following levels of expressiveness:

1. The institutions $SubPGHorn^{=}$, $PGHorn^{=}$, $SubPGCond^{=}$ and $PGCond^{=}$ (all involving partiality) are all subinstitutions of each other. They form the strongest level of expressiveness. By using more complicated comorphisms, one also can count $SubGHorn^{=}$ and $SubGCond^{=}$ to belong to this level, see [Mos96a]. We have not included these comorphisms here, since they are based on a weaker notion of embedding, which is too weak to be practically useful for borrowing.

2. The next level of expressiveness is $GHorn^{=}$. Here, we lose the ability of specifying conditional generation of data using partial functions within **free** specifications, as shown in [Mos95].

3. Then comes $GCond^{=}$. As shown in [Mos95], due to lack of predicates, we cannot specify conditional generation of relations using **free**, as in the transitive closure example (see Example C.2).

4. The lowest level is $Eq^{=}$, where we have no conditional axioms.

All these levels are separated from each other in [Mos02].

We now come the the description of the comorphisms.

(1): Mapping $SubPGHorn^{=}$ to $SubPGCond^{=}$

Here, we cannot directly use the idea of comorphism (2) from the first-order level of replacing predicates by Boolean functions, since the set of Booleans cannot be axiomatized monomorphically in positive conditional logic.[1] However, what we can do is to axiomatize a *single-valued* set $Bool1$ of truth-values and map predicates to *partial* functions into $Bool1$.

Signatures A signature is translated by adding disjointly to it the presentation

spec BOOL1 =
  sort *Bool*
  op *True1* : *Bool1*
  • $\forall x, y : Bool1 \bullet x = y$
end

and replacing each predicate symbol $p : w$ by a partial function symbol $p : w \longrightarrow ?Bool1$.

Models A model M is translated by replacing each partial function $(p_{w,Bool1})_M$ by its domain, considered as a predicate. Note that model homomorphisms can be translated identically, since they behave in the same way for domains of partial functions and for predicates.

Sentences A sentence is translated by replacing each occurrence of a predicate symbol application $p_w(t_1, \ldots, t_n)$ by the equation $p_{w,Bool1}(t_1, \ldots, t_n) = True1$.

Satisfaction The satisfaction condition is straightforward to show. □

It is easy to show this comorphism is a subinstitution comorphism.

[1] Actually, since positive conditional logic is closed under products by the well-known Mal'cev theorem, any theory having a model with a two-valued Boolean carrier set has also models where the Boolean carrier set exceeds any given cardinality.

(2): Mapping $PGHorn^=$ to $PGCond^=$

Here, we can use the same comorphism as in (1), just restricted to signatures with trivial subsort relation.

(3): Mapping $SubGHorn^=$ to $SubGCond^=$

Here, we can use a comorphism similar to that in (1). The only difference is that predicate symbols $p : w$ are translated to *total* function symbols $p : w \longrightarrow Bool1$. (Note that we cannot represent predicates by subsorts, since predicates may be empty, while subsorts may not.) This leads to a loss of the subinstitution property, but we still have a strongly persistently liberal comorphism, which can be seen as follows: Let $\Sigma = (S, TF, P, \leq)$ be a signature in $SubGHorn^=$. Given a Σ-model M, $\gamma_\Sigma(M)$ extends M by interpreting $Bool1$ as

$$\{true\} \uplus \{p_w(a_1, \ldots, a_n) \mid p : w \in P, a_1, \ldots, a_n \in M_w \setminus p_M\}$$

$True1$ is interpreted with $true$, and

$$(p_w)_{\gamma_\Sigma(M)}(a_1, \ldots, a_n) = \begin{cases} true, & \text{if } (a_1, \ldots, a_n) \in p_M \\ p_w(a_1, \ldots, a_n), & \text{otherwise} \end{cases}$$

Clearly, $\beta_\Sigma(\gamma_\Sigma(M)) = M$. Now let $h\colon M \longrightarrow \beta_\Sigma(N)$ be a Σ-homomorphism. Then h can be uniquely extended to a $\Phi(\Sigma)$-homomorphism $h^\#\colon \gamma(M) \longrightarrow N$ with $\beta_\Sigma(h^\#) = h$ by putting

$$(h^\#)_{Bool1}(true) = True1_N$$
$$(h^\#)_{Bool1}(p_w(a_1, \ldots, a_n)) = (p_w)_N(h(a_1), \ldots, h(a_n))$$

(4): Mapping $GHorn^=$ to $GCond^=$

Here, we can use the same comorphism as in (3), just restricted to signatures with trivial subsort relation. Again, the comorphism is strongly persistently liberal.

(5): Mapping $SubPGHorn^=$ to $SubGHorn^=$

This strongly persistently liberal comorphism works like the comorphism (5) from the first-order level, except that the constants $\bot$ and their axiomatizations are omitted: then, all axioms in $\Phi(\Sigma)$ are in Horn form. The omission of $\bot$ leads to a loss of the weak (injective)-amalgamation property.

(6): Mapping $PGHorn^=$ to $GHorn^=$

This comorphism works similar to the comorphism (4) from the first-order level. As in (5), we have to remove the constants $\bot$ and their axiomatizations which are not in Horn form (thereby losing the weak (injective)-amalgamation property). Moreover, the axioms $\exists x : s \bullet D_s(x)$ are not in Horn form either. These axioms have to be replaced by the introduction of new constants $c_s : s$ (for each $s \in S$). The new constant has essentially the same effect as the existential axiom, except that homomorphisms have to preserve it, which leads to a loss of persistent liberality (but the comorphism still admits model expansion).

(7): Mapping $SubPGCond^=$ to $SubGCond^=$

This works like the (strongly persistently liberal) comorphism (5) above.

(8): Mapping $PGCond^{=}$ to $GCond^{=}$

Here, we cannot restrict (4) from the first-order level, since we need to eliminate the definedness predicates. Therefore, we use the composition

$$PGCond^{=} \hookrightarrow PGHorn^{=} \overset{(6)}{-\!\!\succ} GHorn^{=} \overset{(4)}{\cdots\!\succ} GCond^{=}$$

which is a comorphism admitting model expansion.

(9): Mapping $SubPGHorn^{=}$ to $PGHorn^{=}$

This subinstitution comorphism works exactly as the comorphism (3) from the first-order level: all axioms in $\Phi(\Sigma)$ are already in Horn form.

(10): Mapping $SubPGCond^{=}$ to $PGCond^{=}$

This comorphism works similar to (9). The difference it that the membership predicates have to be deleted, Now when looking at the axioms in $\hat{J}$, one can see that the membership predicates are just the domains of the partial projections. Thus, all occurrences of

$$t \in s$$

in sentences have to be translated to

$$def\ \mathtt{pr}_{(s',s)}(t).$$

(11): Mapping $SubGHorn^{=}$ to $GHorn^{=}$

When setting up this comorphism, we encounter the difficulty to axiomatize the membership predicates. In $\hat{J}$, they are axiomatized as the domains of the partial projections. Now we do not have partial functions at hand. In the comorphism (6) at the first-order level this problem is solved by the axiomatization

$$\forall x : s' \bullet \in^{s}_{s'}(x) \Leftrightarrow \exists y \bullet \mathtt{inj}_{(s,s')}(y) = x$$

But this axiom is not in Horn form. The best that we can do at the moment instead of this is just to use the composition

$$SubGHorn^{=} \hookrightarrow SubPGHorn^{=} \overset{(9)}{\hookrightarrow} PGHorn^{=} \overset{(6)}{-\!\!\succ} GHorn^{=}$$

which is a comorphism admitting model expansion.

(12): Mapping $SubGCond^{=}$ to $GCond^{=}$

For the same reasons as for (11) above, we use the composition

$$SubGCond^{=} \hookrightarrow SubPGCond^{=} \overset{(10)}{\hookrightarrow} PGCond^{=} \overset{(8)}{-\!\!\succ} GCond^{=}$$

which again is a comorphism admitting model expansion.

4.2 Relating modal logics and CASL

It is not obvious how to obtain a forgetful institution morphism from MODALCASL to CASL: on the signature level, it is easy to forget modalities and information about flexibility and rigidity of symbols. However a given MODALCASL model generally consists of many CASL models M_w, and the only way to obtain a CASL model seems to take their product. While this works for the Horn fragment SubPHorn$^{=}$, it does not work for the whole of CASL: e.g. a sentence expressing that there are exactly two carrier elements is not preserved under products.

4.2.1 The standard translation

CASL is obviously a subinstitution of MODALCASL: each CASL signature Σ is turned into a MODALCASL signature $\Phi(\Sigma)$ by equipping it with the empty set of modalities and modality sorts and making all operation and predicate symbols rigid. This easily extends to signature morphisms. A $\Phi(\Sigma)$-model is then turned into a Σ-model by just forgetting the set of worlds. A Σ-sentence naturally can be considered to be a $\Phi(\Sigma)$-sentence.

The following comorphism from the Modal sublanguage of MODALCASL to CASL itself is usually called the "standard translation":

Signatures A modal signature Σ is mapped to a CASL signature $\Phi(\Sigma)$ by adding a sort W (for the set of worlds), a binary relation symbol $R_m : W \times W$ for each modality m and a ternary relation symbol $R_s : s \times W \times W$ for each modality sort s. While rigid symbols are just kept, flexible operation and predicate symbols get the sort W as additional (first) argument. This is easily extended to signature morphisms.

Models A $\Phi(\Sigma)$-model M is turned into a Σ-Kripke model by taking W as set of worlds, R_m and R_s as accessibility relations, and forming M_w by fixing the first argument of flexible operations and predicates in M to w.

This is easily seen to be model-expansive.

Sentences A Σ-sentence is inductively translated to a $\Phi(\Sigma)$-sentence as follows:

$$
\begin{array}{l}
\alpha_\Sigma(w, x : s) = x : s \text{ if } x \text{ is a variable} \\
\alpha_\Sigma(w, f(t_1, \ldots, t_n)) = f(\alpha_\Sigma(t_1), \ldots, \alpha_\Sigma(t_n)) \text{ if } f \text{ is a rigid operation symbol} \\
\alpha_\Sigma(w, f(t_1, \ldots, t_n)) = f(w, \alpha_\Sigma(t_1), \ldots, \alpha_\Sigma(t_n)) \text{ if } f \text{ is a flexible operation symbol} \\
\alpha_\Sigma(w, p(t_1, \ldots, t_n)) = p(\alpha_\Sigma(t_1), \ldots, \alpha_\Sigma(t_n)) \text{ if } p \text{ is a rigid predicate symbol} \\
\alpha_\Sigma(w, p(t_1, \ldots, t_n)) = p(w, \alpha_\Sigma(t_1), \ldots, \alpha_\Sigma(t_n)) \text{ if } p \text{ is a flexible predicate symbol} \\
\alpha_\Sigma(w, def\ t) = def\ \alpha_\Sigma(w, t) \\
\alpha_\Sigma(w, t_1 = t_2) = \alpha_\Sigma(w, t_1) = \alpha_\Sigma(w, t_2) \\
\alpha_\Sigma(w, t_1 \stackrel{e}{=} t_2) = \alpha_\Sigma(w, t_1) \stackrel{e}{=} \alpha_\Sigma(w, t_2) \\
\alpha_\Sigma(w, false) = false \\
\alpha_\Sigma(w, \varphi \wedge \psi) = \alpha_\Sigma(w, \varphi) \wedge \alpha_\Sigma(w, \psi) \\
\alpha_\Sigma(w, \forall x : s \,.\, \varphi) = \forall x : s \,.\, \alpha_\Sigma(w, \varphi) \\
\alpha_\Sigma(w, [m]\,\varphi) = \forall w' : W \,.\, R_m(w, w') \Rightarrow \alpha_\Sigma(w', \varphi) \text{if } m \text{ a modality} \\
\alpha_\Sigma(w, [m : s]\,\varphi) = \forall w' : W \,.\, R_s(\alpha_\Sigma(w, m), w, w') \Rightarrow \alpha_\Sigma(w', \varphi) \text{if } s \text{ a modality sort}
\end{array}
$$

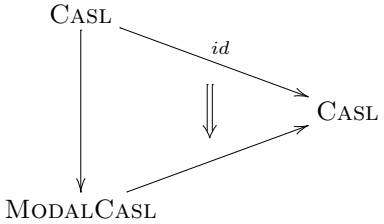
Finally, for a sentence φ, we set $\alpha_\Sigma(\varphi) = \forall w : W \,.\, \alpha_\Sigma(w, \varphi)$.

Satisfaction By a straightforward induction, we can prove

$$\nu, w \Vdash \varphi \text{ iff } \nu[x \mapsto w] \Vdash \alpha(x, \varphi)$$

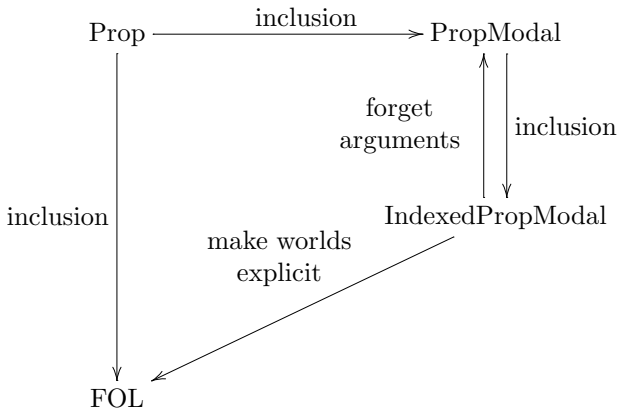
From this, the satisfaction condition follows.

When translating a signature from CASL to MODALCASL and back, we do not get the original signature, but rather its extension by an additional sort W. However, the original signature can be embedded into the extended one, and this gives rise to a comorphism modification from the identity comorphism on CASL to the composite:



4.2.2 Indexed Propositional Modal Logic

For indexed propositional modal logic, we can use the standard translation, but also a reduction to propositional modal logic. Consider the following graph of institutions resp. logics and comorphisms:



The institution comorphisms shown in the above graph are all trivial inclusions, except the comorphisms from IndexedPropModal into FOL and PropModal. We now describe the latter ones.

The comorphism from IndexedPropModal into FOL is just the standard translation of the previous section.

The comorphism from IndexedPropModal to PropModal maps an IndexedPropModal-signature to a PropModal-signature by just forgetting the sorts and the arguments of the predicate symbols, ending up with a set of propositional constants. Similarly, sentences are mapped by forgetting the argument variables for the predicate symbols. Finally, a PropModal-model can be extended to an IndexedPropModal-model by interpreting all carrier sets a singletons; the predicates then actually degenerate to propositional constants, and these are obtained from the PropModal-model. Again, the satisfaction condition is straightforward.

4.3 Relating CASL and CoCASL

Here, we have the following comorphisms:

$$\text{CASL} \longrightarrow \text{PLAINCOCASL} \rightleftarrows \text{MODALCOCASL} \longrightarrow SOL^{=}$$

The first two arrows from left to right are straightforward inclusions. The last one, an encoding of MODALCOCASL into SOL, uses comorphism $(4a) \circ (3)$, composed with the embedding into $SOL^{=}$, from in Sect. 4.1.1. For the **cogenerated cotypes**, the corresponding second-order coinduction principle is generated. The infinite trees needed for **cofree cotypes** can be specified in $SOL^{=}$ as well; actually, Isabelle/HOL [Pau94] already comes with such trees.

Finally, the observation that modal formulae can be regarded as syntactical sugar now becomes the formal statement that the modal CoCASL institution can be encoded in the plain CoCASL institution. Φ takes a MODALCOCASL signature to its underlying CASL (i.e. plain CoCASL) signature, model reduction does nothing, and sentence translation is the encoding of modal logic formulae described in Sect. 3.3.3.

Of course, CoCASL specifications containing modal formulae need to be interpreted in the modal CoCASL institution, while for CoCASL specifications without modal formulae, it does not really matter which of the two institutions is chosen.

The first two comorphisms come with adjoint morphisms: for PLAINCOCASL $\rightarrow$ CASL, the sees and sibling relations are forgotten, and MODALCOCASL $\rightarrow$PLAINCOCASL has the identity signature translation.

4.4 Relating CASL into HASCASL

There is an obvious institution morphism from HASCASL to CASL that keeps just the nullary type constructors (as sorts) and the first-order functions and predicates in the signatures, and their respective interpretations in the models.

The embedding of CASL into HASCASL is a slightly more involved:

Example 4.4 The standard embedding comorphism $\mu = (\Phi, \alpha, \beta): I \longrightarrow J$ from the CASL institution *without sort generation constraints* to the HASCASL institution is defined as follows. The functor Φ sends a CASL signature Σ to the HASCASL signature introducing the same sorts as Σ, while operations and predicates of Σ are turned into constants of appropriate higher-order types. (Recall that predicates in the HASCASL institution are partial functions into the unit type.) Also, the classicality axiom

$$\forall p : \mathit{Logical} \,.\, p \vee \neg p$$

is added. The sentence component α_Σ is the obvious inclusion (using HASCASL's internal logic). The model component β_Σ just forgets the interpretations of higher types.

4.4.1 Liberality

We now show that the comorphism from CASL into HASCASL satisfies some properties that lead to a good interaction with structured specifications.

Example 4.5 The comorphism from Example 4.4 is strongly persistently bi-liberal (in the sense of Def. 2.25), even with natural γ and δ when restricted to CASL without subsorting:

- γ_Σ constructs the interpretation of higher types by freely extending a CASL model; this amounts to interpreting the carriers with λ-terms modulo $\beta\eta$ equivalence. This is a persistent free construction by the persistent extension theorem in the applications section of [Sch]. Moreover, γ is even natural in Σ: Given a CASL signature morphism $\sigma: \Sigma_1 \longrightarrow \Sigma_2$ and a Σ_2-model M, we have to show

  $$\gamma_{\Sigma_2}(M)|_{\Phi(\sigma)} = \gamma_{\Sigma_1}(M|_\sigma)$$

  We show a more general result, namely that for each higher type t in Σ_1, terms $u : t$ in context Γ over $M|_\sigma$ are in one-one correspondence with terms $u : \sigma(t)$ in context $\sigma(\Gamma)$ over M. We proceed by induction over the (higher) types in $\Phi(\Sigma_2)$. For terms of base types, we are done since γ is persistent. For terms of higher higher types $u : \bar{s} \to t$, we have two cases: either $u = \lambda \bar{x} \,.\, v$, and we can apply the induction hypothesis, or u is a variable, and we are done.

  In a setting without subtypes, this implies that the equation displayed above holds, since equality of the freely generated elements of function types is determined only by the rules of higher order logic and hence is the same on both sides.

- δ_Σ extends a CASL model to the corresponding standard model in HASCASL (i.e. all higher types are interpreted by the full function space). This is obviously strongly persistent. We need to show that δ_Σ is right adjoint to β_Σ: The counit is just the identity. Given a homomorphism $h: \beta_\sigma(M') \longrightarrow M$ from a HASCASL $\Phi(\Sigma)$-model M' into a CASL Σ-model M, its unique extension $h^\# : M' \longrightarrow \delta_\Sigma(M)$ to the higher types is just given by mapping (inductively over the types) any value of functional type in the intensional Henkin model M' to its behaviour function, which is in $\delta_\Sigma(M)$ because the latter is a standard model. δ is obviously natural in Σ.

For naturality of γ, absence of subsorting is really needed: if σ is the extension of a signature Σ_1 with sorts s, t, u, v and operations $f : s \to u$, $f : t \to u$, $g : v \to s$, $g : v \to t$ to a signature Σ_2 which differs from Σ_1 only by having $s < t$, then the terms $\lambda x : v \bullet f((g\,x) : s)$ and $\lambda x : v \bullet f\,((g\,x) : t)$ are equal in $\gamma_{\Sigma_2}(M)$ and hence in $\gamma_{\Sigma_2}(M)|_{\Phi(\sigma)}$, but in general different in $\gamma_{\Sigma_1}(M|_\sigma)$.

Proposition 4.6 The embedding of CASL to HASCASL preserves the semantics of structured specifications, including free specifications when restricted to CASL without subsorting (but excluding sort generation constraints), in the sense that the set of HASCASL models of a CASL specification, when restricted to the basic types, coincides with the original model semantics in CASL.

PROOF: See Prop. C.9. □

4.4.2 Sort generation constraints

Due to the flexibility of interpretation of higher types in Henkin models, the higher-order reformulation of a sort-generation constraint is weaker than the original constraint in CASL. In particular, not all non-standard models are excluded, and it is impossible to do so. Otherwise, one would need to equip as least some parts of HASCASL with a standard semantics. Since the Henkin semantics has very pleasant properties ([Sch] and [SMM] above), we have deliberately refrained from doing so.

Concerning unstructured and structured free types, this leads to the following situation:

$$\begin{array}{ccc} \mathbf{Mod}^{\mathrm{CASL}}(\textbf{free types } DD) & \subset & \beta(\mathbf{Mod}^{\mathrm{HASCASL}}(\textbf{free types } DD)) \\ = & & \subset \\ \mathbf{Mod}^{\mathrm{CASL}}(\textbf{free } \{\textbf{types } DD\}) & = & \beta(\mathbf{Mod}^{\mathrm{HASCASL}}(\textbf{free } \{\textbf{types } DD\})) \end{array}$$

where the inclusion from the top right to the bottom right corner also holds inside the $\beta(\ldots)$.

Hence, unstructured free types (and in general, the translation of sort generation constraints) have more models in HASCASL than in CASL. Still, this does not violate conservativity: any semantic consequence of an embedding of a CASL specification into HASCASL is also a semantic consequence of the original CASL specification. The latter specification may have more semantic consequences. But even this difference disappears w.r.t. to proof theory — at least if the standard CASL proof system with the usual finitary induction rule is used. Only if stronger (e.g. infinitary) forms of induction are used, the difference becomes relevant. It also becomes relevant for monomorphicity: due to possible non-standard interpretations of higher types, the usual free datatypes are no longer monomorphic in HASCASL.

4.5 CSP-CASL

There is an obvious subinstitution comorphism from CASL to CSP-CASL, adding the empty set of process names to CASL signatures. However, since CSP-CASL has restrictions on the signature category (only sort-injective signature morphisms are allowed), this comorphism actually starts from a subinstitution of CASL. Dually, there is an obvious morphism from CSP-CASL to CASL, forgetting the set of process names in the signatures and the processes in the models.

Interestingly, there is also a simple theoroidal semi-comorphism LTL: CSP-CASL $\longrightarrow$ MODALCASL. A signature is extended by a sort lab, and injection operations $inj\colon s \longrightarrow lab$ for each sort s. These are axiomatized to be injective and to jointly generate lab. Signature morphisms are just extended to map the extra structure on the nose in the obvious way.

A MODALCASL-model is translated to a CSP-CASL-model by forgetting the interpretation of the Kripke structure part to get a CASL-model, and equipping it with the LTS determined by the transition relation of the MODALCASL model.

A part of the graph(s) of logics developed so far is shown in Fig. 4.4: namely those logics that are part of the Heterogeneous Tool Set.

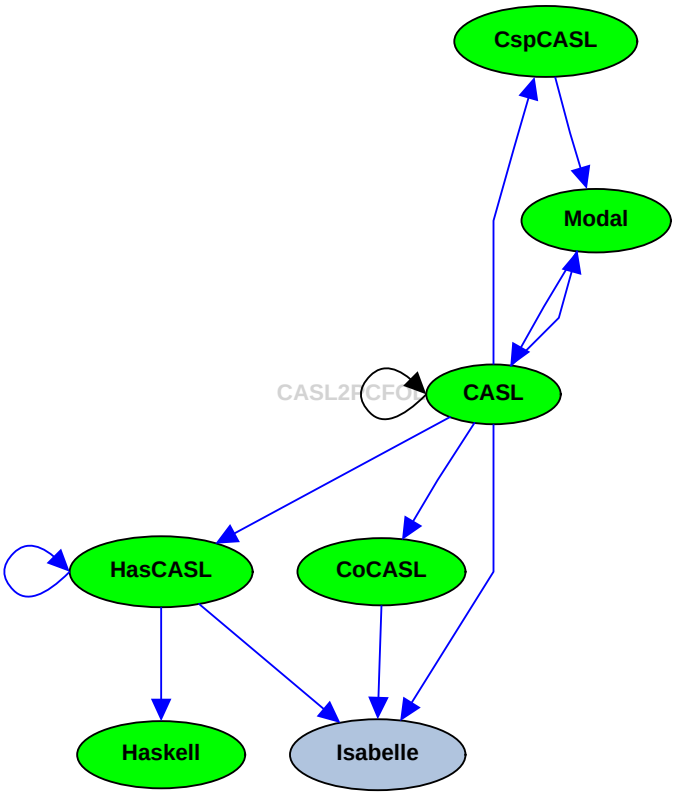


Figure 4.4: Graph of logics.

4.6 Bibliographical Notes

The translation among CASL sublanguage have been described in [Mos02]. The translations for COCASL are from [MSRR]. The standard translation for modal logic can be found e.g. in [AGM92].

Chapter 5

Structured Specification and Development Graphs

As a preparation for the discussion of heterogeneous (structured) specification, in this chapter we will introduce homogeneous structured specifications over an arbitrary institution. We first recall a popular kernel language for structured specifications, and then propose development graphs as a kernel formalism for structured theorem proving. In Chap. 6, we then will realize heterogeneous specification by just considering structured specification over a so-called Grothendieck institution.

5.1 Specifications Over an Arbitrary Institution

In this section we recall a popular set of institution-independent structuring operations, which seems to be quite universal and which can also be seen as a kernel language for the CASL structuring constructs.

In the sequel, let us fix an arbitrary institution $I = (\mathbf{Sign}, \mathbf{Sen}, \mathbf{Mod}, \models)$. Based on an arbitrary such I, the following kernel language for specifications in an arbitrary institution has been proposed [ST88b]. Simultaneously with the notion of specification, we define functions *Sig* and **Mod** yielding the signature and the model class of a specification.

***presentations*:** For any signature $\Sigma \in |\mathbf{Sign}|$ and finite set $\Psi \subseteq \mathbf{Sen}(\Sigma)$ of Σ-sentences, the *presentation* $\langle \Sigma, \Psi \rangle$ is a specification with:

$$\begin{aligned} \mathit{Sig}(\langle \Sigma, \Psi \rangle) &:= \Sigma \\ \mathbf{Mod}(\langle \Sigma, \Psi \rangle) &:= \{M \in \mathbf{Mod}(\Sigma) \mid M \models \Psi\} \end{aligned}$$

***union*:** For any signature $\Sigma \in |\mathbf{Sign}|$, given Σ-specifications SP_1 and SP_2, their *union* $SP_1 \cup SP_2$ is a specification with:

$$\begin{aligned} \mathit{Sig}(SP_1 \cup SP_2) &:= \Sigma \\ \mathbf{Mod}(SP_1 \cup SP_2) &:= \mathbf{Mod}(SP_1) \cap \mathbf{Mod}(SP_2) \end{aligned}$$

***translation*:** For any signature morphism $\sigma\colon \Sigma \longrightarrow \Sigma'$ and Σ-specification SP, **translate** SP **by** σ is a specification with:

$$\begin{aligned} \mathit{Sig}(\textbf{translate } SP \textbf{ by } \sigma) &:= \Sigma' \\ \mathbf{Mod}(\textbf{translate } SP \textbf{ by } \sigma) &:= \{M' \in \mathbf{Mod}(\Sigma') \mid M'|_\sigma \in \mathbf{Mod}(SP)\} \end{aligned}$$

***hiding*:** For any signature morphism $\sigma\colon \Sigma \longrightarrow \Sigma'$ and Σ'-specification SP', **derive from** SP' **by** σ is a specification with:

$$\begin{aligned} \mathit{Sig}(\textbf{derive from } SP' \textbf{ by } \sigma) &:= \Sigma \\ \mathbf{Mod}(\textbf{derive from } SP' \textbf{ by } \sigma) &:= \{M'|_\sigma \mid M' \in \mathbf{Mod}(SP')\} \end{aligned}$$

The above *specification-building operations*, although extremely simple, already provide flexible mechanisms for expressing basic ways of putting specifications together and thus building specifications in a structured manner. Hence, they can be considered to be a kernel language for structured specification. The specification language CASL provides more sophisticated structuring constructs, but it is possible to translate the CASL constructs (except the **free** construct, which will be examined in Appendix C) to the above kernel language, see [Mos00].

A specification SP is said to be *consistent*, if $\mathbf{Mod}(SP)$ is not empty.

Given two structured specifications SP_1 and SP_2, a *specification morphism* $\sigma\colon SP_1 \longrightarrow SP_2$ is a signature morphism $\sigma\colon Sig(SP_1) \longrightarrow Sig(SP_2)$ such that $M|_\sigma \in \mathbf{Mod}(SP_1)$ for each $M \in \mathbf{Mod}(SP_2)$. For presentations, this boils down to *presentation morphisms* $\sigma\colon \langle \Sigma, \Psi\rangle \longrightarrow \langle \Sigma', \Psi'\rangle$, which are just signature morphisms $\sigma\colon \Sigma \longrightarrow \Sigma'$ for which $\Psi' \models_{\Sigma'} \sigma(\Psi)$, that is, axioms are mapped to logical consequences.

A structured Σ-specification SP_2 *refines* a structured Σ-specification SP_1 (written $SP_1 \approx\!\!\gg SP_2$), if $\mathbf{Mod}(SP_2) \subseteq \mathbf{Mod}(SP_1)$.

5.2 Borrowing

Often, an institution does not have an entailment system itself, but can be encoded (via a comorphism) into another institution that has one. The borrowing technique allows for re-using entailment systems via comorphisms. This works not only for entailment between flat specifications, but also for structured specifications. Here, we introduce a semantic version of borrowing, relating the semantic consequence relations of two institutions, but using the results of the previous section, it should be clear that this carries over to entailment as well.

Definition 5.1 Let $\mu = (\Phi, \alpha, \beta)\colon I \longrightarrow J$ be an institution comorphism and $\mathcal{SP}$ a class of I-specifications. We say that μ *admits borrowing of entailment for* $\mathcal{SP}$, if for any Σ-specification $SP \in \mathcal{SP}$ and any Σ-sentence φ in I, we have

$$SP \models^I_\Sigma \varphi \text{ iff } \hat{\mu}(SP) \models^J_{Sig(\Phi(\Sigma))} \alpha_\Sigma(\varphi).$$

Moreover, we say that μ *admits borrowing of refinement for* $\mathcal{SP}$, if for any Σ-specifications $SP_1, SP_2 \in \mathcal{SP}$, we have

$$SP_1 \approx\!\!\gg SP_2 \text{ iff } \hat{\mu}(SP_1) \approx\!\!\gg \hat{\mu}(SP_2).$$

□

The importance of this definition lies in the following: If we have a sound proof calculus for entailment in J, and if we have an institution comorphism $\mu\colon I \longrightarrow J$ admitting borrowing of entailment for $\mathcal{SP}$, we can use the proof calculus also for proving entailment concerning I-specifications in $\mathcal{SP}$: we just have to translate our proof goals using $\hat{\mu}$ and α. If, moreover, the proof calculus is complete for proving entailment in J, then also its re-use for proving entailment in I is complete. A similar remark holds for proof calculi for refinement.

Since

$$SP \models_\Sigma \varphi \text{ iff } \langle \Sigma, \{\varphi\}\rangle \approx\!\!\gg SP,$$

μ admits borrowing of entailment for $\mathcal{SP}$ if μ admits borrowing of refinement for $\mathcal{SP}$ (provided that $\mathcal{SP}$ contains all specifications of form $\langle \Sigma, \{\varphi\}\rangle$).

Proposition 5.2 Let $\mu = (\Phi, \alpha, \beta)\colon I \longrightarrow J$ be an institution comorphism and $\mathcal{SP}$ a class of I-specifications.

1. Assume that for each $SP \in \mathcal{SP}$, $\mathbf{Mod}^I(SP) = \beta_{Sig(SP)}(\mathbf{Mod}^J(\hat{\mu}(SP)))$[1]. Then μ admits borrowing of entailment for $\mathcal{SP}$.

[1] This includes the condition that $\beta_{Sig(SP)}$ is defined on $\mathbf{Mod}^J(\hat{\mu}(SP))$.

2. Assume that μ admits model expansion and that for each $SP \in \mathcal{SP}$, $\beta^{-1}_{Sig(SP)}(\mathbf{Mod}^I(SP)) = \mathbf{Mod}^J(\hat{\mu}(SP))$[2]. Then μ admits borrowing of entailment and of refinement for $\mathcal{SP}$.

3. The assumption in (1) can be weakened to $\beta_{Sig(SP)}(\mathbf{Mod}^J(\hat{\mu}(SP))) \subseteq \mathbf{Mod}^I(SP)$, if additionally the simultaneous restriction and corestriction $\beta_{Sig(SP)}\colon \mathbf{Mod}^J(\hat{\mu}(SP)) \longrightarrow \mathbf{Mod}^I(SP)$ is isomorphism-dense, and satisfaction in I is closed under isomorphism.

4. The assumption of model expansion in (2) can be replaced by assuming that β is pointwise isomorphism-dense and for each $SP \in \mathcal{SP}$, $\mathbf{Mod}^I(SP)$ is isomorphism-closed.

PROOF: (1) Let $SP \in \mathcal{SP}$ be a Σ-specification and φ be a Σ-sentence. Then

$SP \models^I_\Sigma \varphi$
iff (by definition) $M \in \mathbf{Mod}^I(SP)$ implies $M \models^I_\Sigma \varphi$
iff (by the assumption) $M' \in \mathbf{Mod}^J(\hat{\mu}(SP))$ implies $\beta_\Sigma(M') \models^I_\Sigma \varphi$
iff (by the satisfaction condition)
$\quad M' \in \mathbf{Mod}^J(\hat{\mu}(SP))$ implies $M' \models^J_{Sig(\Phi(\Sigma))} \alpha_\Sigma(\varphi)$
iff (by definition) $\hat{\mu}(SP) \models^J_{Sig(\Phi(\Sigma))} \alpha_\Sigma(\varphi)$.

(2) Concerning borrowing of entailment, by surjectivity of β_Σ, we obtain $\beta_\Sigma(\beta_\Sigma^{-1}(\mathcal{M})) = \mathcal{M}$ for any $\mathcal{M} \subseteq \mathbf{Mod}^I(\Sigma)$. Thus, we obtain that the assumption of (1) is fulfilled, and the result follows.

Concerning borrowing of refinement, let $SP_1, SP_2 \in \mathcal{SP}$ be Σ-specifications. Assume that $SP_1 \leadsto SP_2$. If now $M' \in \mathbf{Mod}^J(\hat{\mu}(SP_2))$, by the assumption of the proposition, we get $\beta_\Sigma(M') \in \mathbf{Mod}^I(SP_2)$ and therefore $\beta_\Sigma(M') \in \mathbf{Mod}^I(SP_1)$. Again by the assumption of the proposition, we obtain $M' \in \mathbf{Mod}^J(\hat{\mu}(SP_1))$. Hence, $\hat{\mu}(SP_1) \leadsto \hat{\mu}(SP_2)$.

Conversely, assume that $\hat{\mu}(SP_1) \leadsto \hat{\mu}(SP_2)$. If now $M \in \mathbf{Mod}^I(SP_2)$, by the assumptions of the proposition, we get some $M' \in \mathbf{Mod}^J(\hat{\mu}(SP_2))$ with $\beta_\Sigma(M') = M$. Since then also $M' \in \mathbf{Mod}^J(\hat{\mu}(SP_1))$, by the assumption of the proposition also $\beta_\Sigma(M') = M \in \mathbf{Mod}^I(SP_1)$. Hence, $SP_1 \leadsto SP_2$.

(3) In the step of the proof of (1) where we use the assumption, we now only get some $M' \in \mathbf{Mod}^J(\hat{\mu}(SP))$ with $\beta_{Sig(SP)}(M') \cong M$, instead of $\beta_{Sig(SP)}(M') = M$. But this does no harm since satisfaction in I is closed under isomorphism.

(4) Similarly as (3). □

Borrowing of entailment is strictly weaker than borrowing of refinement, see [Bor02] for an example.

5.2.1 Borrowing For Structured Specifications

An important use of institution comorphisms is the re-use (also called borrowing) of proof calculi and theorem provers. Hence, we will study conditions under which an institution comorphism admits borrowing.

For comorphisms admitting model expansion, the well-known "Borrowing theorem" [CM97, Tar96] holds:

Theorem 5.3 Let I and J be two institutions and $\rho = (\Phi, \alpha, \beta)\colon I \longrightarrow J$ be an institution comorphism admitting model expansion. Then ρ admits borrowing of entailment and refinement for theories.

PROOF: Let $SP = \langle \Sigma, \Psi \rangle$ be a theory. Then $\beta_\Sigma(M')$ is defined and satisfies SP iff $\beta_\Sigma(M')$ is defined and satisfies Ψ iff (by the satisfaction condition) M' satisfies $Ax(\Phi(\Sigma)) \cup \alpha_\Sigma(\Psi)$ iff M' satisfies $\hat{\rho}(SP)$. Thus $\beta_\Sigma^{-1}\mathbf{Mod}^I(SP) = \mathbf{Mod}^J(\hat{\rho}(SP))$. The result now follows from Proposition 5.2 (2). □

[2]This precisely means $\beta_{Sig(SP)}(M')$ is defined and a member of $\mathbf{Mod}^I(SP)$ if and only if $M' \in \mathbf{Mod}^J(\hat{\mu}(SP))$.

That is, if we have a sound (and complete) theorem prover for theories in the target institution of an institution comorphism admitting model expansion, we can re-use it as a sound (and complete) theorem prover for theories in the source institution. In a word:

> *Institution comorphisms admitting model expansion also admit borrowing of entailment and refinement for theories.*

The following proposition has been proved in [Bor02]:

Proposition 5.4 Let I and J be two institutions, let $\mathcal{D}$ be a class of signature morphisms in I, and let $\mu = (\Phi, \alpha, \beta)\colon I \longrightarrow J$ be an simple theoroidal institution comorphism. Let $\mathcal{SP}$ be the set of structured specifications in I containing **derive**s only along morphisms in $\mathcal{D}$. Then

1. For any specification SP,
$$\beta_{Sig[SP]}(\mathbf{Mod}^J(\hat{\mu}(SP))) \subseteq \mathbf{Mod}^I(SP).$$
2. If μ has the weak $\mathcal{D}$-amalgamation property, then for $SP \in \mathcal{SP}$,
$$(\beta_{Sig[SP]})^{-1}(\mathbf{Mod}^I(SP)) = \mathbf{Mod}^J(\hat{\mu}(SP)).$$
3. If μ admits model expansion and has the weak $\mathcal{D}$-amalgamation property, then for $SP \in \mathcal{SP}$,
$$\beta_{Sig[SP]}(\mathbf{Mod}^J(\hat{\mu}(SP))) = \mathbf{Mod}^I(SP).$$
4. If μ admits model expansion and has the weak $\mathcal{D}$-amalgamation property, then μ admits borrowing of entailment and refinement for $\mathcal{SP}$. □

PROOF:
(1) and (2) The proof mainly goes along the lines of the proof in [Bor99]. The crucial difference here is that the model translation β_Σ is not defined on all of $\mathbf{Mod}^J(Sig[\Phi(\Sigma)])$, but only for those models satisfying $Ax[\Phi(\Sigma)]$. Let us understand sentences like $\beta_\Sigma(M) \in \mathbf{Mod}^I(SP)$ as "$\beta_\Sigma(M)$ is defined and in $\mathbf{Mod}^I(SP)$". We will prove for any $Sig[\Phi(\Sigma)]$-model M that

$$M \in \mathbf{Mod}^J(\hat{\mu}(SP)) \text{ iff } \beta_\Sigma(M) \in \mathbf{Mod}^I(SP)$$

where the "if" direction additionally needs weak $\mathcal{D}$-amalgamation. (1) and (2) then easy follow.

We first need to prove a Lemma:

Lemma 5.5 Given a signature morphism $\sigma\colon \Sigma \longrightarrow \Sigma'$ in I and an $Sig[\Phi(\Sigma')]$-model M', we have

$$\beta_{\Sigma'}(M') \text{ defined implies } \beta_\Sigma(M'|_{\Phi(\sigma)}) \text{ defined.}$$

PROOF: Assume that $\beta_{\Sigma'}(M')$ is defined. This means $M' \in \mathbf{Mod}(\Phi(\Sigma'))$. Since $\Phi(\sigma)\colon \Phi(\Sigma) \longrightarrow \Phi(\Sigma')$ is a theory morphism, $M'|_{\Phi(\sigma)} \in \mathbf{Mod}(\Phi(\Sigma))$, i.e. $\beta_\Sigma(M'|_{\Phi(\sigma)})$ is defined.

We prove

$$M \in \mathbf{Mod}^J(\hat{\mu}(SP)) \text{ iff } \beta_\Sigma(M) \in \mathbf{Mod}^I(SP)$$

by induction over SP (assuming additionally weak $\mathcal{D}$-amalgamation when proving the "if" direction).

- $SP = \langle \Sigma, \Psi \rangle$: $M \in \mathbf{Mod}^J(\hat{\mu}(SP))$ iff $M \in \mathbf{Mod}^J(\langle Sig[\Phi(\Sigma)], Ax[\Phi(\Sigma)] \cup \alpha_\Sigma(\Psi)\rangle)$. This holds iff $M \models^J_{Sig[\Phi(\Sigma)]} \alpha_\Sigma(\Psi)$ and $M \in dom\beta_\Sigma$. But this is equivalent to $\beta_\Sigma(M) \models^I_\Sigma \Psi$, i.e. $\beta_\Sigma(M) \in \mathbf{Mod}^I(SP)$.
- $SP = SP_1 \cup SP_2$: $M \in \mathbf{Mod}^J(\hat{\mu}(SP))$ iff $M \in \mathbf{Mod}^J(\hat{\mu}(SP_1) \cup \hat{\mu}(SP_2)$ iff $M \in \mathbf{Mod}^J(\hat{\mu}(SP_1)) \cap \mathbf{Mod}(\hat{\mu}(SP_2))$. By induction hypothesis, this holds iff $\beta_\Sigma(M) \in \mathbf{Mod}^I(SP_1) \cap \mathbf{Mod}^I(SP_2)$, i.e., iff $\beta_\Sigma(M) \in \mathbf{Mod}^I(SP_1 \cup SP_2)$.

- $SP = \mathbf{translate}\ SP_1\ \mathbf{by}\ \sigma\colon \Sigma_1 \longrightarrow \Sigma$: $M \in \mathbf{Mod}^J(\hat{\mu}(SP))$ iff $M \in \mathbf{Mod}^J(\mathbf{translate}\ \hat{\mu}(SP_1)\ \mathbf{by}\ \Phi(\sigma) \cup \Phi(\Sigma))$. By definition, this is equivalent to $M|_{\Phi(\sigma)} \in \mathbf{Mod}^J(\hat{\mu}(SP_1))$ and $M \in \mathbf{Mod}^J(\Phi(\Sigma))$ (the latter being equivalent to $\beta_\Sigma(M)$ defined). By induction hypothesis and the Lemma, this holds iff $\beta_{\Sigma_1}(M|_{\Phi(\sigma)}) \in \mathbf{Mod}^I(SP_1)$ and $\beta_\Sigma(M)$ is defined. By naturality of β, this is equivalent to $\beta_\Sigma(M)|_\sigma \in \mathbf{Mod}^I(SP_1)$, i.e. $\in \mathbf{Mod}^I(\mathbf{translate}\ SP_1\ \mathbf{by}\ \sigma\colon \Sigma_1 \longrightarrow \Sigma)$.
- $SP = \vdash SP'\sigma\colon \Sigma \longrightarrow \Sigma_1$, "only if" direction: $M \in \mathbf{Mod}^J(\hat{\mu}(SP))$ implies $M \in \mathbf{Mod}^J(\vdash \hat{\mu}(SP')\Phi(\sigma))$. This means that there is some $M' \in \mathbf{Mod}^J(\Sigma_1)$ with $M'|_{\Phi(\sigma)} = M$. Fix such an M'. By induction hypothesis, $\beta_{\Sigma_1}(M')$ is defined and in $\mathbf{Mod}^I(SP')$. By the Lemma, also $\beta_\Sigma(M'|_{\Phi(\sigma)}) = \beta_\Sigma(M)$ is defined. By naturality of β, $(\beta_{\Sigma_1}(M'))|_\sigma = \beta_\Sigma(M'|_{\Phi(\sigma)}) = \beta_\Sigma(M)$. Thus, $\beta_{\Sigma_1}(M')$ is a witness for $\beta_\Sigma(M) \in \mathbf{Mod}^I(\vdash SP'\sigma\colon \Sigma \longrightarrow \Sigma_1)$.
- $SP = \vdash SP'\sigma\colon \Sigma \longrightarrow \Sigma_1$, "if" direction: $\beta_\Sigma(M) \in \mathbf{Mod}^I(\vdash SP'\sigma\colon \Sigma \longrightarrow \Sigma_1)$ means that there is some $M_1 \in \mathbf{Mod}^I(SP')$ with $M_1|_\sigma = \beta_\Sigma(M)$. Since $\sigma \in \mathcal{D}$, by weak $\mathcal{D}$-amalgamation there is some $M_1' \in \mathbf{Mod}^J(\Phi(\Sigma_1))$ with $M_1'|_{\Phi(\sigma)} = M$ and $\beta_{\Sigma_1}(M_1') = M_1$. By induction hypothesis, $M_1' \in \mathbf{Mod}^J(\hat{\mu}(SP'))$. This shows that $M \in \mathbf{Mod}^J(\vdash \hat{\mu}(SP')\Phi(\sigma))$.

(3) By (2), $(\beta_\Sigma)^{-1}(\mathbf{Mod}^I(SP)) \subseteq \mathbf{Mod}^J(\hat{\mu}(SP))$. By surjectivity of β_Σ, $Mod^I(SP) = \beta_\Sigma((\beta_\Sigma)^{-1}(\mathbf{Mod}^I(SP))) \subseteq \beta_\Sigma(\mathbf{Mod}^J(\hat{\mu}(SP)))$. The converse inclusion is just (1).
(4) $SP \models^I_\Sigma \varphi$ iff for all $M \in \mathbf{Mod}^I(SP)$, $M \models^I_\Sigma \varphi$ iff (by (3)) for all $M' \in \mathbf{Mod}^J(\hat{\mu}(SP))$, $\beta_\Sigma(M) \models^I_\Sigma \varphi$ iff (by the satisfaction condition) for all $M' \in \mathbf{Mod}^J(\hat{\mu}(SP))$, $M' \models^J_{Sig[\Phi(\Sigma)]} \alpha_\Sigma(\varphi)$ iff $hat\mu(SP) \models^J_{Sig[\Phi(\Sigma)]} \alpha_\Sigma(\varphi)$.

The combination of (2) and (3) means that model classes (loose semantics) are preserved and reflected. This is especially important since CASL (like many other specification languages) has a model-theoretic semantics: a specification denotes a signature together with a class or category of models.

> *Loose semantics for structured specifications with hiding only along $\mathcal{D}$-morphisms can be lifted along and against simple theoroidal institution comorphisms admitting model expansion and weak $\mathcal{D}$-amalgamation.*

Loose semantics in this context means taking just the class of *all* models of a specification as its semantics. Of course, whether these are, e.g., all first-order models or just the finitely generated ones, depends on the model functor of the institution.

(4) means that tools for theorem proving within structured specifications in the target institution can be re-used for theorem proving within structured specifications in the source institution. That is:

> *simple theoroidal institution comorphisms admitting model expansion and weak $\mathcal{D}$-amalgamation admit borrowing of entailment and refinement for structured specifications with hiding only along $\mathcal{D}$-morphisms.*

5.3 A Proof Calculus for Structured Specifications

As explained above, the semantics of structured specifications is parameterized over an institution providing the semantics of basic specifications. The situation with the proof calculus is similar: here, we need a logic, i.e. an institution equipped with an entailment system. Based on this, it is possible to design a logic independent proof calculus [Bor02] for proving entailments of the form $SP \vdash \varphi$, where SP is a structured specification and φ is a formula, see Fig. 5.1. Fig. 5.2 shows an extension of the structured proof calculus to refinements between specifications. Note that for the latter calculus, an *oracle for conservative extensions* is needed. A specification morphism $\sigma\colon SP_1 \longrightarrow SP_2$ is conservative iff each SP_1-model is the σ-reduct of some SP_2-model.[3]

$$(CR)\ \frac{\{SP \vdash \varphi_i\}_{i\in I} \quad \{\varphi_i\}_{i\in I} \vdash \varphi}{SP \vdash \varphi} \qquad (basic)\ \frac{\varphi \in \Psi}{\langle \Sigma, \Psi\rangle \vdash \varphi}$$

$$(sum1)\ \frac{SP_1 \vdash \varphi}{SP_1 \cup SP_2 \vdash \varphi} \qquad (sum2)\ \frac{SP_1 \vdash \varphi}{SP_1 \cup SP_2 \vdash \varphi}$$

$$(trans)\ \frac{SP \vdash \varphi}{\sigma(SP) \vdash \sigma(\varphi)} \qquad (derive)\ \frac{SP \vdash \sigma(\varphi)}{\sigma^{-1}(SP) \vdash \varphi}$$

Figure 5.1: Proof calculus for entailment in structured specifications

$$(Basic)\ \frac{SP \vdash \Psi}{\langle \Sigma, \Psi\rangle \rightsquigarrow SP} \qquad (Sum)\ \frac{SP_1 \rightsquigarrow SP \quad SP_2 \rightsquigarrow SP}{SP_1 \cup SP_2 \rightsquigarrow SP}$$

$$(Trans_1)\ \frac{SP \rightsquigarrow \theta(SP') \quad \theta = \sigma^{-1}}{\sigma(SP) \rightsquigarrow SP'} \qquad (Trans_2)\ \frac{SP \rightsquigarrow \sigma^{-1}(SP')}{\sigma(SP) \rightsquigarrow SP'}$$

$$(Derive)\ \frac{SP \rightsquigarrow SP''}{\sigma^{-1}(SP) \rightsquigarrow SP'} \quad \text{if } \sigma\colon SP' \longrightarrow SP'' \text{ is a conservative extension}$$

$$(Trans\text{-}equiv)\ \frac{\theta(\sigma(SP)) \rightsquigarrow SP'}{\theta \circ \sigma(SP) \rightsquigarrow SP'}$$

Figure 5.2: Proof calculus for refinement of structured specifications

We will now discuss some important meta properties of the calculus. Before we can state a completeness theorem, we need to formulate some technical assumptions on the underlying institution I. We also rely on the notions of having amalgamation and Craig interpolation, which have been defined for an arbitrary institution in Sect. 2.3.

An institution *has conjunction*, if for any Σ-sentences φ_1 and φ_2, there is a Σ-sentence φ that holds in a model iff φ_1 and φ_2 hold. The notion of an institution *having implication* is defined similarly.

Theorem 5.6 (Soundness [Bor02]) The calculi for structured entailment and refinement between finite structured specifications given above are sound.

Theorem 5.7 (Completeness [Bor02]) Under the assumptions that

- the institution has the *Craig interpolation property*,
- the institution admits *weak amalgamation*,
- the institution has conjunction and implication and
- the logic is *complete*,

the calculi for structured entailment and refinement between finite structured specifications are complete. Note that for the refinement calculus, an *oracle for conservative extensions* is needed.

Actually, the assumption of Craig interpolation and weak amalgamation can be restricted to those diagrams for which it is really needed. Details can be found in [Bor02].

The problem with the above completeness theorem is that its prerequisites do not hold in most of the institutions introduced in Chap. 3. Firstly, these institutions are not closed under conjunction and implication due to the presence of special sentences like sort generation, cogeneration and

[3]Besides this model-theoretic notion of conservativeness, there also is a weaker consequence-theoretic notion: $SP_2 \models \sigma(\varphi)$ implies $SP_1 \models \varphi$, and a proof-theoretic notion: $SP_2 \vdash \sigma(\varphi)$ implies $SP_1 \vdash \varphi$ coinciding with the consequence-theoretic one for complete logics. For the calculus of refinement, we need the model-theoretic notion.

cofreeness constraints. While it is easy to close these institutions under conjunctions, such that conjunction of constraints with other constraints and with first-order formulas become possible, the situation is different for closure under implication. Implication between constraints or negation of constraints (note that negation can be obtained via an implication to $false$) increase expressiveness considerably. A similar remark holds for the various Horn clause subinstitution of CASL defined in Sect. 3.1.9: implications between Horn clauses are strictly stronger expressive than Horn clauses themselves.

Secondly, also Craig interpolation fails in the presence of e.g. sort generation constraints, see Sect. 3.1.8. But sort generation constraints are an essential ingredient of the CASL logic, because they are needed for the specification of the usual inductive datatypes used in programming. Our answer to this problem is to use a different institution independent proof calculus for structured specifications, based on the formalism of *development graphs*. This will be the subject of the subsequent sections.

5.4 Development Graphs

We now introduce so-called *development graphs* as a simple kernel formalism for structured theorem proving and proof management. A development graph consists of a set of nodes (corresponding to whole structured specifications or parts thereof), and a set of arrows called definition links, indicating the dependency of each involved structured specification on its subparts.

The *proof calculus* for development graphs is given by rules that allow for decomposing *global* theorem links into simpler ones, until eventually *local implications* are reached. The latter can be discharged using a logic-specific calculus as given by an entailment system (see Sect. 2.4).

The main advantage of this calculus over the one introduced in Sect. 5.3 is the weaker set of assumptions for completeness of the calculus: basically, we need completeness of the underlying entailment system plus existence of quasi-semi-exactness (cf. Def. 2.8). The latter property is much weaker than Craig interpolation, and easier to fulfill in a heterogeneous framework, see Sect. 6.2. We postpone the proof of completeness of the calculus to Sect. 6.4, where we will generalize it to the heterogeneous case.

In contrast to the language of structured specifications of Sect. 5.1, development graphs allow for expressing the *sharing* among specifications due to multiple references to named specifications. Moreover, the proof management tool of the Heterogeneous Tool Set HETS works directly on development graphs; hence, the material presented here can serve as a formal background for the use of HETS and for the understanding of how it works (see Chap. 7). Last but not least, development graphs also support management of change [AHMS00] and have been used in large-scale industrial applications [HLS+96]. The graph structure provides a direct visualization of the structure of specifications, and it also allows for managing large specifications with hundreds of sub-specifications.

Before we introduce development graphs, consider the following running example of specifying and refining a sorting function *sorter*.

Given some specification of total orders and lists, an abstract specification of this sorting function may be denoted in CASL syntax as follows:

spec SORTING
[TOTALORDER]
=
{
LIST [**sort** *Elem*]
then
preds $is\_ordered : List[Elem]$;
$permutation : List[Elem] \times List[Elem]$;
forall $x, y : Elem$;
$L, L1, L2 : List[Elem]$
- $is\_ordered([])$
- $is\_ordered([x])$

- $is\_ordered(x :: (y :: L)) \Leftrightarrow x \leq y \wedge is\_ordered(y :: L)$
- $permutation(L1, L2) \Leftrightarrow (\forall x : Elem \bullet x \in L1 \Leftrightarrow x \in L2)$

then

op $sorter : List[Elem] \to List[Elem]$;
forall $L : List[Elem]$
- $is\_ordered(sorter(L))$
- $permutation(L, sorter(L))$

}
hide $is\_ordered, permutation$
end

is_ordered and *permutation* are auxiliary predicates to specify *sorter*, and are hidden to the outside. A model of this specification is just an interpretation of the *sorter* function (together with a model of the imported specifications of total orders and lists) that can be extended to a model of the whole specification (including *is_ordered* and *permutation*).

During a development, we may refine SORTING into a design specification describing a particular sorting algorithm. For simplicity, we choose a sorting algorithm which recursively inserts the head element in the sorted tail list. In CASL we obtain the following specification:

spec INSERTSORT
[TOTALORDER]
=
{
LIST [**sort** *Elem*]
then
ops $insert : Elem \times List[Elem] \to List[Elem]$;
$insert\_sort : List[Elem] \to List[Elem]$;
forall $x, y : Elem$;
$L : List[Elem]$
- $insert(x, []) = [x]$
- $insert(x, y :: L) =$
  $x :: insert(y, L)$ *when* $x \leq y$ *else* $y :: insert(x, L)$
- $insert\_sort([]) = []$
- $insert\_sort(x :: L) = insert(x, insert\_sort(L))$

}
hide *insert*
end

Now we want to state INSERTSORT is actually a refinement of SORTING, i.e. that each INSERTSORT-model is also a SORTING-model. This is written as follows:

view INSERTSORTCORRECTNESS[TOTALORDER] : SORTING[TOTALORDER] **to** INSERTSORT[TOTALORDER]
=
$sorter \mapsto insert\_sort$

Fig. 5.3 shows the corresponding development graph.

Development graphs are structured as follows. Leaves in a graph correspond to basic specifications, which do not make use of other specifications. Inner nodes correspond to structured specifications. The links that capture the construction of structured specifications in the graph are called *definition links*. Arising proof obligations are attached as so-called *theorem links* to this graph.

Definition 5.8 A *development graph* is an acyclic, directed graph $\mathcal{DG} = \langle \mathcal{N}, \mathcal{L} \rangle$.

$\mathcal{N}$ is a set of nodes.[4] Each node $N \in \mathcal{N}$ is labelled with a pair (Σ^N, Ψ^N) such that Σ^N is a

[4]The structure of nodes is left unspecified here; we assume that they come from a given infinite set *Nodes* of nodes, to make the choice of 'new' nodes and edges deterministic, we may assume that *Nodes* comes equipped with a fixed enumeration – then 'new' always means 'first not used as yet'.

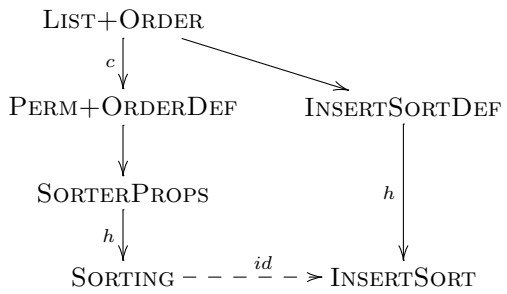


Figure 5.3: Development graph for the sorting example

signature and $\Psi^N \subseteq \mathbf{Sen}(\Sigma^N)$ is the set of *local axioms* of N.

$\mathcal{L}$ is a set of directed links, so-called *definition links*, between elements of $\mathcal{N}$. Each definition link from a node K to a node N is either

- *global* (denoted $K \xrightarrow{\sigma} N$), annotated with a signature morphism $\sigma : \Sigma^K \to \Sigma^N$, or
- *local* (denoted $K \xrightarrow{\sigma}\!\!\!\!\rightarrow N$), again annotated with a signature morphism $\sigma : \Sigma^K \to \Sigma^N$, or
- *hiding* (denoted $K \xrightarrow[hide]{\sigma} N$), annotated with a signature morphism $\sigma : \Sigma^N \to \Sigma^K$ *going against the direction of the link*, or
- *free* (denoted $K \xrightarrow[free]{\sigma} N$), annotated with a signature morphism $\sigma : \Sigma \to \Sigma^K$ for some signature Σ, with the requirement that $\Sigma^K = \Sigma^N$.[5]

To simplify matters, we write $K \xrightarrow{\sigma} N \in \mathcal{DG}$ instead of $K \xrightarrow{\sigma} N \in \mathcal{L}$ when $\mathcal{L}$ are the links of $\mathcal{DG}$. We use N, K, P, Q, P as variables for nodes, and L as variable for links.

Since development graphs are acyclic, we can use induction principles in definitions and proofs concerning development graphs.

The next definition captures the existence of a path of local and global definition links between two nodes. Notice that such a path must not contain any hiding links.

Definition 5.9 Let $\mathcal{DG}$ be a development graph. The notion of *global reachability* is defined inductively: a node N is globally reachable from a node K via a signature morphism σ, $K \xRightarrow{\sigma} N$ for short, iff

- either $K = N$ and $\sigma = id$, or
- $K \xrightarrow{\sigma'} P \in \mathcal{DG}$, and $P \xRightarrow{\sigma''} N$, with $\sigma = \sigma'' \circ \sigma'$, or
- $K \xrightarrow[free]{\sigma'} P \in \mathcal{DG}$ and $P \xRightarrow{\sigma} N$ (note that σ' is just ignored here).

A node N is *locally reachable* from a node K via a signature morphism σ, $K{>}\xRightarrow{\sigma} N$ for short, iff $K \xRightarrow{\sigma} N$ or there is a node P with $K \xrightarrow{\sigma'}\!\!\!\!\rightarrow P \in \mathcal{DG}$ and $P \xRightarrow{\sigma''} N$, such that $\sigma = \sigma'' \circ \sigma'$. Note that, in contrast to global reachability, local reachability is *not* transitive.

Obviously global reachability implies local reachability.

[5]Although freeness will be studied in more detail only in Appendix C, we include free definition links in order not to have to modify the semantics of nodes later on.

Definition 5.10 Given a node $N \in \mathcal{N}$, its associated class $\mathbf{Mod}_{\mathcal{DG}}(N)$ of models (or N-models for short) is inductively defined to consist of those Σ^N-models M for which

1. M satisfies the local axioms Ψ^N,
2. for each $K \stackrel{\sigma}{\Longrightarrow} N \in \mathcal{DG}$, $M|_\sigma$ is an K-model,
3. for each $K \stackrel{\sigma}{\longrightarrow} N \in \mathcal{DG}$, $M|_\sigma$ satisfies the local axioms Ψ^K,
4. for each $K \stackrel{\sigma}{\underset{hide}{\Longrightarrow}} N \in \mathcal{DG}$, M has a σ-expansion M' (i.e. $M'|_\sigma = M$) that is an K-model, and
5. for each $K \stackrel{\sigma}{\underset{free}{\Longrightarrow}} N \in \mathcal{DG}$, M is an K-model that is σ-free in $\mathbf{Mod}(K)$. The latter means that for each K-model M' and each model morphism $h: M|_\sigma \longrightarrow M'|_\sigma$, there exists a unique model morphism $h^\# : M \longrightarrow M'$ with $h^\#|_\sigma = h$.

Definition 5.11 Let $\mathcal{DG} = \langle \mathcal{N}, \mathcal{L} \rangle$ be a development graph. A node $N \in \mathcal{N}$ is *flattenable* iff for all nodes $K \in \mathcal{N}$ with incoming hiding or free definition links, it holds that N is not globally reachable from K.

Definition 5.12 Let $\mathcal{DG} = \langle \mathcal{N}, \mathcal{L} \rangle$ be a development graph. For $N \in \mathcal{N}$, the *theory* $Th_{\mathcal{DG}}(N)$ of N is defined by

$$\Psi^N \cup \bigcup_{P > \stackrel{\sigma}{\Longrightarrow} N} \sigma(\Psi^P)$$

Proposition 5.13 1. $K \stackrel{\sigma}{\Longrightarrow} N$ and $M \in \mathbf{Mod}(N)$ imply $M|_\sigma \in \mathbf{Mod}(K)$.

2. If $K > \stackrel{\sigma}{\Longrightarrow} N$ and $M \in \mathbf{Mod}(N)$, then $M|_\sigma \models \Psi^K$.

PROOF: 1. Easy induction over the definition of global reachability.
2. By 1 and Definition 5.10, 3.

Proposition 5.14 1. $\mathbf{Mod}(N) \subseteq \mathbf{Mod}(Th_{\mathcal{DG}}(N))$.

2. If N is flattenable, then $\mathbf{Mod}(N) = \mathbf{Mod}(Th_{\mathcal{DG}}(N))$.

PROOF: 1. By Proposition 5.13, 2 and Definition 5.10, 1.
2. By 1, it suffices to prove the '$\supseteq$' direction. Let M be a $Th_{\mathcal{DG}}(N)$-model. Let $len(p)$ be the length of a path p witnessing $K \stackrel{\tau}{\Longrightarrow} N$. Let $maxp$ be the maximal such length in $\mathcal{DG}$. We show that for any $K \stackrel{\tau}{\Longrightarrow} N$, $M|_\tau$ is an K-model. We proceed by induction over $maxp - len(p)$ with p witnessing $K \stackrel{\tau}{\Longrightarrow} N$. Since N is flattenable, we only have to show clauses 1 to 3 of Definition 5.10:

1. Since global implies local reachability, $K > \stackrel{\tau}{\Longrightarrow} N$, and $\tau(\Psi^K) \subseteq Th_{\mathcal{DG}}(N)$; hence $M \models \tau(\Psi^K)$. By the satisfaction condition for institutions, $M|_\tau \models \Psi^K$.
2. Let $P \stackrel{\theta}{\Longrightarrow} K$, hence $P \stackrel{\tau\circ\theta}{\Longrightarrow} N$. By the induction hypothesis, $M|_{\tau\circ\theta}$ is a P-model.
3. Let $P \stackrel{\theta}{\longrightarrow} K$, hence $P > \stackrel{\tau\circ\theta}{\Longrightarrow} N$. With a similar argument as for 1, we get $M|_{\tau\circ\theta} \models \Psi^P$.

This completes the induction. Since $N \stackrel{id}{\Longrightarrow} N$, M is an N-model. □

Definition 5.15 $\mathcal{DG}_1 = \langle \mathcal{N}_1, \mathcal{L}_1 \rangle$ is a *subgraph* of $\mathcal{DG}_2 = \langle \mathcal{N}_2, \mathcal{L}_2 \rangle$ if $\mathcal{N}_1 \subseteq \mathcal{N}_2$ and $\mathcal{L}_1 \subseteq \mathcal{L}_2$. It is a *faithful subgraph*, if all links in $\mathcal{L}_2 \setminus \mathcal{L}_1$ have target nodes in $\mathcal{N}_2 \setminus \mathcal{N}_1$. Also, in this case $\mathcal{DG}_2$ is called a *faithful supergraph* of $\mathcal{DG}_1$.

Model classes do not change when passing to faithful supergraphs:

Proposition 5.16 If $\mathcal{DG}_1$ is a faithful subgraph of $\mathcal{DG}_2$ and N a node in $\mathcal{DG}_1$, then

$$\mathbf{Mod}_{\mathcal{DG}_1}(N) = \mathbf{Mod}_{\mathcal{DG}_2}(N).$$

PROOF: The notion of N-model only depends on the local axioms of N and definition links going into N. Both do not change when passing to a faithful supergraph. □

Complementary to definition links, which *define* the theories of related nodes, we introduce the notion of a *theorem link* with the help of which we are able to *postulate* relations between different theories. Theorem links are the central data structure to represent proof obligations arising in formal developments. Theorem links come, like definition links, in four different versions:

- *global* theorem links $K \overset{\sigma}{=\!=\!\Rightarrow} N$, where $\sigma\colon \Sigma^K \longrightarrow \Sigma^N$,
- *local* theorem links $K \overset{\sigma}{-\!-\!\succ} N$, where $\sigma\colon \Sigma^K \longrightarrow \Sigma^N$,
- *hiding* theorem links $K \underset{hide}{\overset{\sigma}{=\!=\!\Rightarrow}}_{\theta} N$, where for some Σ, $\theta\colon \Sigma \longrightarrow \Sigma^K$ and $\sigma\colon \Sigma \longrightarrow \Sigma^N$[6], and
- *free* theorem links $K \underset{free}{\overset{\sigma}{=\!=\!\Rightarrow}}_{\theta} N$, where $\sigma\colon \Sigma^K \longrightarrow \Sigma^N$ and for some Σ, $\theta\colon \Sigma \longrightarrow \Sigma^K$. In case that Σ is the initial signature and θ is the unique signature morphism, the link is written as $K \underset{free}{\overset{\sigma}{=\!=\!\Rightarrow}}_{!} N$.

Moreover, we will also need *local implications* of the form $N \Rightarrow \Psi$, where Ψ is a set of Σ^N-sentences. $N \Rightarrow \{\varphi\}$ also is written $N \Rightarrow \varphi$. The semantics of local implications and of theorem links is given by the next definition.

Definition 5.17 Let $\mathcal{DG}$ be a development graph and K, N nodes in $\mathcal{DG}$.

- $\mathcal{DG}$ implies a local implication $N \Rightarrow \Psi$, written $\mathcal{DG} \models N \Rightarrow \Psi$, if for all $M \in \mathbf{Mod}_{\mathcal{DG}}(N)$, $M \models \Psi$.
- $\mathcal{DG}$ implies a global theorem link $K \overset{\sigma}{=\!=\!\Rightarrow} N$ (denoted $\mathcal{DG} \models K \overset{\sigma}{=\!=\!\Rightarrow} N$) iff for all $M \in \mathbf{Mod}_{\mathcal{DG}}(N)$, $M|_\sigma \in \mathbf{Mod}_{\mathcal{DG}}(K)$.
- $\mathcal{DG}$ implies a local theorem link $K \overset{\sigma}{-\!-\!\succ} N$ (denoted $\mathcal{DG} \models K \overset{\sigma}{-\!-\!\succ} N$) iff for all $M \in \mathbf{Mod}_{\mathcal{DG}}(N)$, $M|_\sigma \models \Psi^K$. (Note that by the satisfaction condition, this is equivalent to $\mathcal{DG} \models N \Rightarrow \sigma(\Psi^K)$.)
- $\mathcal{DG}$ implies a hiding theorem link $K \underset{hide}{\overset{\sigma}{=\!=\!\Rightarrow}}_{\theta} N$ (denoted $\mathcal{DG} \models K \underset{hide}{\overset{\sigma}{=\!=\!\Rightarrow}}_{\theta} N$) iff for all $M \in \mathbf{Mod}_{\mathcal{DG}}(N)$, $M|_\sigma$ has a θ-expansion to some K-model.
- $\mathcal{DG}$ implies a free theorem link $K \underset{free}{\overset{\sigma}{=\!=\!\Rightarrow}}_{\theta} N$ (denoted $\mathcal{DG} \models K \underset{free}{\overset{\sigma}{=\!=\!\Rightarrow}}_{\theta} N$) iff for all $M \in \mathbf{Mod}_{\mathcal{DG}}(N)$, $M|_\sigma$ is an K-model which is θ-free in $\mathbf{Mod}_{\mathcal{DG}}(K)$.

Remark 5.18 Note that theorem links may be captured by inclusion between model classes of some (additional) nodes in the development graph. For instance, consider a hiding theorem link $K \underset{hide}{\overset{\sigma}{=\!=\!\Rightarrow}}_{\theta} N$ in a development graph $\mathcal{DG}$, where $\theta\colon \Sigma \longrightarrow \Sigma^K$ and $\sigma\colon \Sigma \longrightarrow \Sigma^N$. One can add nodes

[6]Here, σ is the translation morphism (comparable to that of global theorem links), and θ is the hiding morphism (extending a signature with hidden parts).

N' and N'' to $\mathcal{DG}$, with $\Sigma^{N'} = \Sigma$, $\Sigma^{N''} = \Sigma^N$, and $\Psi^{N'} = \Psi^{N''} = \emptyset$, together with definition links $K \xrightarrow[hide]{\theta} N'$ and $N' \xrightarrow{\sigma} N''$. For the thus obtained development graph $\mathcal{DG}'$, we then have

$$\mathcal{DG} \models K = \overset{\sigma}{\underset{hide}{\Rightarrow}}\, N \text{ iff } \mathbf{Mod}_{\mathcal{DG}'}(N) \subseteq \mathbf{Mod}_{\mathcal{DG}'}(N'')$$

A similar construction can be performed for the other types of theorem link. □

Finally, we introduce the analogues of the semantic annotations in CASL. A global theorem link $K = \overset{\sigma}{\Rightarrow} N$ can be strengthened to

- a *conservative extension*[7] (denoted as $K = \overset{\sigma}{\underset{cons}{\Rightarrow}} N$); it holds if, additionally to the holding of the theorem link, every K-model has a σ-expansion to an N-model,
- a *monomorphic extension* (denoted as $K = \overset{\sigma}{\underset{mono}{\Rightarrow}} N$); it holds if, additionally to the holding of the theorem link, every K-model has a σ-expansion to an N-model that is unique up to isomorphism, or
- a *definitional extension* (denoted as $K = \overset{\sigma}{\underset{def}{\Rightarrow}} N$); it holds if, additionally to the holding of the theorem link, every K-model has a unique σ-expansion to an N-model.

These annotations can be seen as another kind of proof obligations. If there happens to be a global definition link $K \xrightarrow{\sigma} N$ in the development graph, we also write $K \xrightarrow[cons]{\sigma} N$, $K \xrightarrow[mono]{\sigma} N$, or $K \xrightarrow[def]{\sigma} N$, respectively. In this case, the theorem link part holds trivially, and only the conservativity, monomorphicity or definitionality statement is relevant.

We also allow for annotating *nodes* with *cons*, *mono* or *def*. This shall express that the trivial theorem link using the unique signature morphism from the empty signature[8] could be annotated with the same word.[9] Thus, the annotation *cons* for a node means that there is a model of the node (consistency), *mono* means that the node has exactly one model up to isomorphism (i.e. it is monomorphic), and *def* means that the node has exactly one model (the latter will occur only rarely).

5.5 Verification Semantics for Structured Specifications

The link between structured specifications and development graphs is given by a *verification semantics*. Given a structured specification, the verification semantics constructs a development graph and a singles out a particular node in the graph that captures the semantics of the structured specification. The rules of the verification semantics use a suggestive concise notation for extending a given development graph $\mathcal{DG}$, like the notation $\mathcal{DG}' = \mathcal{DG} \uplus \{N' := (\Sigma', \Psi);\ N \xrightarrow{\sigma} N'\}$. This should be largely self-explanatory (in particular, '$N' := (\Sigma', \Psi)$' means that we introduce a new node N' with $\Sigma^{N'} = \Sigma'$ and $\Psi^{N'} = \Psi$).

$$\frac{}{\vdash \langle \Sigma, \Psi \rangle \rhd\!\!\rhd (N, \{N := (\Sigma, \Psi)\})}$$

$$\frac{\begin{array}{c}\vdash SP_1 \rhd\!\!\rhd (N_1, \mathcal{DG}_1)\\ \vdash SP_2 \rhd\!\!\rhd (N_2, \mathcal{DG}_2)\\ \Sigma^{N_1} = \Sigma^{N_2} = \Sigma\end{array}}{\vdash SP_1 \cup SP_2 \rhd\!\!\rhd (K, \mathcal{DG}_1 \uplus \mathcal{DG}_2 \uplus \{K := (\Sigma, \emptyset)\} \uplus \{\, N_i \xrightarrow{id} K \mid i = 1, 2\})}$$

[7] In the literature on model theory, this property is often called *model expansion property*, while the term *conservative extension* refers to a (weaker) proof-theoretic principle.

[8] We here assume that the empty signature is initial.

[9] Here we tacitly assume that there is some special node having the initial signature and the empty set of axioms.

$$\frac{\vdash SP \rhd\!\!\rhd\!\!\rhd (N, \mathcal{DG})}{\vdash \textbf{translate } SP \textbf{ by } \sigma\colon \Sigma^N \longrightarrow \Sigma' \rhd\!\!\rhd\!\!\rhd (K, \mathcal{DG} \uplus \{ N \xRightarrow{\sigma} K := (\Sigma', \emptyset)\})}$$

$$\frac{\vdash SP \rhd\!\!\rhd\!\!\rhd (N, \mathcal{DG})}{\vdash \textbf{derive from } SP' \textbf{ by } \sigma\colon \Sigma' \longrightarrow \Sigma^N \rhd\!\!\rhd\!\!\rhd (K, \mathcal{DG} \uplus \{ N \xRightarrow[hide]{(\sigma)} K := (\Sigma', \emptyset)\})}$$

We also provide a verification semantics for views between specifications. Note that it returns just a development graph.

$$\frac{\begin{array}{c}\vdash SP_1 \rhd\!\!\rhd\!\!\rhd (N_1, \mathcal{DG}_1) \\ \vdash SP_2 \rhd\!\!\rhd\!\!\rhd (N_2, \mathcal{DG}_2)\end{array}}{\vdash \textbf{view } SP_1 \textbf{ to } SP_2 = \sigma\colon \Sigma^{N_1} \longrightarrow \Sigma^{N_2} \rhd\!\!\rhd\!\!\rhd \mathcal{DG}_1 \uplus \mathcal{DG}_2 \uplus \{ N_1 = \overset{\sigma}{=} \Rightarrow N_2\}}$$

Theorem 5.19 The verification semantics preserves model classes of structured specifications. More precisely, given a structured specification SP with $\vdash SP \rhd\!\!\rhd\!\!\rhd (N, \mathcal{DG})$, we have

$$\mathbf{Mod}[SP] = \mathbf{Mod}_{\mathcal{DG}}(N)$$

PROOF: See Theorem 6.45 for a more general result. □

5.6 Proof Rules for Development Graphs

In this section, we introduce logic-independent proof rules for development graphs. These rely on a logic-specific entailment relation for basic specifications in the sense of Sect. 2.4, as well as on logic-specific proof rules for conservativity and freeness, as e.g. given in Sect. 3.1.5 and Appendix C.

The proof rules work on judgements of the form $\mathcal{DG} \vdash L$, where $\mathcal{DG}$ is a development graph and L is a theorem link (of any kind) over $\mathcal{DG}$. We follow a natural deduction style presentation and additionally use a graph-grammar like notation. We hope that this is still largely self-explanatory while improving readability.

The proof rules for development graphs presented below are typically applied backwards: given proof goal in form of a theorem link relative to some development graph, find a rule whose conclusion matches the proof goal, and recursively prove the premises of the rule. Note that within one rule, the judgements may refer to different development graphs. Often, the premises are formulated over development graphs that are larger than that for the conclusion. This means that applying rules backwards possibly adds some new nodes and edges to the development graph.

The rules allow for decomposing global theorem links into simpler ones. In a first step, one typically tries to get rid of hiding theorem links and to decompose global into local theorem links. This is done by applying the hiding decomposition rules. Thereby, new conservativity proof goals can be generated, which need to be tackled by the conservativity rules. The simple decomposition rules then allow for proving global theorem links when there is some parallel definition link, and for proving local theorem links and local implications by reasoning with the entailment system of the logic.

For the sake of readability, each rule is followed by its soundness proof.

5.6.1 Faithful Extension Rule

$$\frac{\mathcal{DG} \vdash \ K = \overset{\sigma}{=} \Rightarrow N}{\mathcal{DG}' \vdash \ K = \overset{\sigma}{=} \Rightarrow N}$$

if K and N occur in both $\mathcal{DG}$ and $\mathcal{DG}'$, and $\mathcal{DG}$ is a faithful sub- oder supergraph of $\mathcal{DG}'$

(Faithful-Extension)

Soundness of **(Faithful-Extension)**: by Prop. 5.16. □

5.6.2 Hiding Decomposition Rules

In order to get rid of hiding links going into the *source* of a global theorem link, one first applies **(Glob-Decomposition)**, ending up with some local and hiding theorem links. The rule **(Hide-Theorem-Shift)** allows to prove the latter, using conservativity of definition links. **(Borrowing)** can be used for shifting a proof goal along a conservative extension; hence, it also exploits conservativity of theorem links. Conservativity is dealt with in the next section. The central rule of the proof system is the rule **(Theorem-Hide-Shift)**. It is used to get rid of hiding definition links going into the *target* of a global theorem link.

$$\frac{\begin{array}{ccc} & & N' \\ & \overset{\sigma'}{\nearrow} & \uparrow{\scriptstyle \theta'} \\ K' & & N \end{array} \quad (\theta' \text{ is } cons)}{K' \overset{\sigma}{\underset{hide}{=\!\Rightarrow}}_{\theta} N \quad \text{if } \sigma' \circ \theta = \theta' \circ \sigma}$$

(Hide-Theorem-Shift)

The proof rules are written in a concise notation as above. We will spell out in detail what this notation means for the rule **(Hide-Theorem-Shift)**:

$$\frac{\begin{array}{c} \sigma' \circ \theta = \theta' \circ \sigma \\ N \overset{\theta'}{\Longrightarrow} N' \in \mathcal{DG} \\ \mathcal{DG} \vdash K' \overset{\sigma'}{=\!\Rightarrow} N' \\ \mathcal{DG} \vdash N \overset{\theta'}{\underset{cons}{=\!\Rightarrow}} N' \end{array}}{\mathcal{DG} \vdash K' \overset{\sigma}{\underset{hide}{=\!\Rightarrow}}_{\theta} N}$$

Soundness of **(Hide-Theorem-shift)**: assume that $\mathcal{DG} \models K' \overset{\sigma'}{=\!\Rightarrow} N'$ and $N \overset{\theta'}{\underset{cons}{=\!\Rightarrow}} N'$ is conservative. We have to show that $\mathcal{DG} \models K' \overset{\sigma}{\underset{hide}{=\!\Rightarrow}}_{\theta} N$. Let M be an N-model. Since $N \overset{\theta'}{\underset{cons}{=\!\Rightarrow}} N'$ is conservative, M can be expanded to an N'-model M' with $M'|_{\theta'} = M$. By the assumption, $M'|_{\sigma'}$ is an K'-model. Thus, $M'|_{\sigma' \circ \theta} = M'|_{\theta' \circ \sigma} = M|_{\sigma}$ has a θ-expansion to an K'-model. □

$$\frac{\begin{array}{ccc} & & G_N(i) \\ & & \Big\| {\scriptstyle \mu_i} \quad (i \in |J|) \\ K & \overset{\mu_{\langle N \rangle} \circ \sigma}{=\!=\!=\!\Rightarrow} & P \end{array}}{\begin{array}{ccc} & \overset{Diag_N}{\longrightarrow} & \\ & \searrow \quad \swarrow & \\ K & \overset{\sigma}{=\!=\!\Rightarrow} & N \end{array}}$$

with P isolated and (μ_i) a weakly amalgamable cocone for the diagram $Diag_N$ of nodes going into N (see explanation below)

(Theorem-Hide-Shift)

Since this rule is quite powerful, we need some preliminary notions. Given a node N in a development graph $\mathcal{DG} = \langle \mathcal{N}, \mathcal{L} \rangle$, the idea is that we unfold the subgraph below N into a tree and

form a diagram with this tree. More formally, define the *diagram* $Diag_N\colon J \longrightarrow \mathbf{Sig}$ *associated with* N together with a map $G_N\colon |J| \longrightarrow \mathcal{N}$ inductively as follows:

- $\langle N \rangle$ is an object in J, with $Diag_N(\langle N \rangle) = \Sigma^N$. Let $G_N(\langle N \rangle)$ be just N.
- if $i = \langle\ K \xrightarrow{L_1} \cdots \xrightarrow{L_n} N\ \rangle$ is an object in J with $L_1, \ldots, L_n$ non-local definition links in $\mathcal{L}$, and $L = P \xRightarrow{\sigma} K$ or $L = P \xrightarrow{\sigma} K$ is a local or global definition link in $\mathcal{L}$, then
$$j = \langle\ P \xrightarrow{L} K \xrightarrow{L_1} \cdots \xrightarrow{L_n} N\ \rangle$$
is an object in J with $Diag_N(j) = \Sigma^P$, and L is a morphism from j to i in J with $Diag_N(L) = \sigma$. We set $G_N(j) = P$.
- if $i = \langle\ K \xrightarrow{L_1} \cdots \xrightarrow{L_n} N\ \rangle$ is an object in J with $L_1, \ldots, L_n$ non-local definition links in $\mathcal{L}$, and $L = P \xRightarrow[hide]{\sigma} K$ is a hiding definition link in $\mathcal{L}$, then
$$j = \langle\ P \xrightarrow{L} K \xrightarrow{L_1} \cdots \xrightarrow{L_n} N\ \rangle$$
is an object in J with $Diag_N(j) = \Sigma^P$, and L is a morphism from i to j in J with $Diag_N(L) = \sigma$. We set $G_N(j) = P$.

Now in order to apply **(Theorem-Hide-Shift)**, take a weakly amalgamable cocone $(\Sigma, (\mu_i\colon Diag_N(i) \longrightarrow \Sigma)_{i \in |J|})$ for $Diag_N$ (in general, we know that such a cocone exists only if the institution is quasi-semi-exact), and let P be a new isolated node with signature Σ and with ingoing global definition links $G_N(i) \xRightarrow{\mu_i} P$ for $i \in |J|$ (if $G_N(i)$ has no ingoing free definition links, a local definition link $G_N(i) \xrightarrow{\mu_i} P$ would suffice). Here, an isolated node is one with no local axioms and no ingoing definition links other than those shown in the rule.

We once more spell the rule (this time **(Theorem-Hide-Shift)**) in detail:

$$\frac{\begin{array}{c}(\Sigma, (\mu_i\colon Diag_N(i) \longrightarrow \Sigma)_{i \in |J|}) \text{ is a weakly amalgamable cocone for } Diag_N \\ \mathcal{DG}' = \mathcal{DG} \uplus \{P \text{ with } (\Sigma, \emptyset)\} \uplus \{\, G_N(i) \xRightarrow{\mu_i} P \mid i \in |J|\,\} \\ \mathcal{DG}' \vdash K \overset{\mu_{\langle N \rangle} \circ \sigma}{=\!=\!\Rightarrow} P\end{array}}{\mathcal{DG} \vdash K \overset{\sigma}{=\!=\!\Rightarrow} N}$$

Here, if we want to extend a given development graph $\mathcal{DG}$, we use a suggestive concise notation like $\mathcal{DG}' = \mathcal{DG} \uplus \{N' \text{ with } (\Sigma', \Psi);\ N \xRightarrow{\Sigma \hookrightarrow \Sigma'} N'\}$ which should be largely self-explanatory (in particular, 'N' with (Σ', Ψ)' means that we introduce a new node N' with $\Sigma^{N'} = \Sigma'$ and $\Psi^{N'} = \Psi$). Note that **(Theorem-Hide-Shift)** is the only rule where an extension $\mathcal{DG}'$ of $\mathcal{DG}$ occurs in the premise of the rule. However, this extension is faithful, and hence, by rule **(Faithful Extension)**, the difference between $\mathcal{DG}$ and $\mathcal{DG}'$ in the above rule is inessential.

Soundness of **(Theorem-Hide-Shift)**: assume that $\mathcal{DG} \models K \overset{\mu_{\langle N \rangle} \circ \sigma}{=\!=\!\Rightarrow} P$. Let M be an N-model. We have to show $M|_\sigma$ to be an K-model in order to establish the holding of $K \overset{\sigma}{=\!=\!\Rightarrow} N$. We inductively define a family $(M_i)_{i \in |I|}$ of models $M_i \in \mathbf{Mod}(G_N(i))$ by putting

- $M_{\langle N \rangle} = M$,
- $M_{\langle P \xrightarrow{L} Q \xrightarrow{L_1} \cdots \xrightarrow{L_n} N \rangle} = M'|_\sigma$, where $L = P \xRightarrow{\sigma} Q$ or $L = P \xrightarrow{\sigma} Q$ and $M' = M_{\langle Q \xrightarrow{L_1} \cdots \xrightarrow{L_n} N \rangle}$, and
- $M_{\langle P \xrightarrow{L} Q \xrightarrow{L_1} \cdots \xrightarrow{L_n} N \rangle}$ is a σ-expansion of M' to a P-model (existing since M' is a Q-model), where $L = P \xRightarrow[hide]{\sigma} Q$ and $M' = M_{\langle Q \xrightarrow{L_1} \cdots \xrightarrow{L_n} N \rangle}$.

It is easy to show that this family is consistent with $Diag_N$. Since by the side condition of the rule, $(\Sigma, (\mu_i \colon Diag_N(i) \longrightarrow \Sigma)_{i \in |J|})$ is a weakly amalgamable cocone, there is a Σ^P-model M_K with $M_K|_{\mu_i} = M_i$. The latter implies that M_K is a P-model. By the assumption, $M_K|_{\mu_{\langle N \rangle} \circ \sigma} = M_{\langle N \rangle}|_\sigma = M|_\sigma$ is an K-model. □

$$\frac{\begin{array}{ccc} K & & N \\ \| & & \| \\ {\scriptstyle\theta}\| & & {\scriptstyle\theta'}\| {\scriptstyle cons} \\ \Downarrow & & \Downarrow \\ K' & \underset{\sigma'}{=\!=\!\Rightarrow} & N' \end{array}}{\begin{array}{ccc} K & \underset{\sigma}{=\!=\!\Rightarrow} & N \\ \| & & \| \\ {\scriptstyle\theta}\| & & {\scriptstyle\theta'}\| {\scriptstyle cons} \\ \Downarrow & & \Downarrow \\ K' & & N' \end{array}} \quad \text{if } \sigma' \circ \theta = \theta' \circ \sigma$$

(Borrowing)

Soundness of **(Borrowing)**: Assume that (1) $\mathcal{DG} \models K = \overset{\theta}{=\!=\!\Rightarrow} K'$, (2) $\mathcal{DG} \models N = \underset{cons}{\overset{\theta'}{=\!=\!\Rightarrow}} N'$, and that (3) $\mathcal{DG} \models K' = \overset{\sigma'}{=\!=\!\Rightarrow} N'$. Let M be an N-model. By (2), M has an expansion to an N'-model M' with $M'|_{\theta'} = M$. By (3), $M'|_{\sigma'}$ is an K'-model, and hence, by (1) $M'|_{\sigma' \circ \theta} = M'|_{\theta' \circ \sigma} = M|_\sigma$ is an K-model. □

$$\frac{\begin{array}{l} P -\!\!\!\!\!\!\sim\!\!\!\!\!\!\succ K \text{ for each } P \overset{\tau}{\Longrightarrow\!\!\!\!\!\!\rightarrow} N \\ Q \underset{hide}{=\!=\!\overset{\sigma \circ \tau}{\Rightarrow}} K \text{ for each } Q \underset{hide}{\overset{\theta}{\Longrightarrow}} P \text{ and } P \overset{\tau}{\Longrightarrow\!\!\!\!\!\!\rightarrow} N \\ Q \underset{free}{=\!=\!\overset{\sigma \circ \tau}{\Rightarrow}} K \text{ for each } Q \underset{free}{\overset{\theta}{\Longrightarrow}} P \text{ and } P \overset{\tau}{\Longrightarrow\!\!\!\!\!\!\rightarrow} N \end{array}}{N = \overset{\sigma}{=\!=\!\Rightarrow} K}$$

(Glob-Decomposition)

Soundness of **(Glob-Decomposition)**: assume that

1. $\mathcal{DG} \models P -\!\!\!\!\!\!\sim\!\!\!\!\!\!\succ K$ for each $P \overset{\tau}{\Longrightarrow\!\!\!\!\!\!\rightarrow} N$,
2. $\mathcal{DG} \models Q \underset{hide}{=\!=\!\overset{\sigma \circ \tau}{\Rightarrow}} K$ for each $Q \underset{hide}{\overset{\theta}{\Longrightarrow}} P$ and $P \overset{\tau}{\Longrightarrow\!\!\!\!\!\!\rightarrow} N$, and
3. $\mathcal{DG} \models Q \underset{free}{=\!=\!\overset{\sigma \circ \tau}{\Rightarrow}} K$ for each $Q \underset{free}{\overset{\theta}{\Longrightarrow}} P$ and $P \overset{\tau}{\Longrightarrow\!\!\!\!\!\!\rightarrow} N$.

In order to show $\mathcal{DG} \models N = \overset{\sigma}{=\!=\!\Rightarrow} K$, let M be an K-model. Let $len(p)$ be the length of a path p witnessing $P \overset{\tau}{\Longrightarrow\!\!\!\!\!\!\rightarrow} N$. Let $maxp$ be the maximal such length in $\mathcal{DG}$. We show that for any $P \overset{\tau}{\Longrightarrow\!\!\!\!\!\!\rightarrow} N$, $M|_{\sigma \circ \tau}$ is a P-model. We proceed by induction over $maxp - len(p)$ for p witnessing $P \overset{\tau}{\Longrightarrow\!\!\!\!\!\!\rightarrow} N$. We have to show clauses 1 to 5 of Definition 5.10:

1. By the first assumption, $M|_{\sigma \circ \tau} \models \Psi^P$.
2. By the induction hypothesis, $M|_{\sigma \circ \tau}$ satisfies any global definition link going into P.
3. By the first assumption, $M|_{\sigma \circ \tau}$ satisfies any local definition link going into P.
4. By the second assumption, $M|_{\sigma \circ \tau}$ satisfies any hiding definition link going into P.
5. By the third assumption, $M|_{\sigma \circ \tau}$ satisfies any free definition link going into P.

This completes the induction. Since $N \overset{id}{\Longrightarrow\!\!\!\!\!\!\rightarrow} N$, $M|_\sigma$ is an N-model. □

5.6.3 Conservativity rules

$$\frac{\begin{array}{ccc} K & \overset{\sigma}{=\!\!=\!\!\Rightarrow} & N \\ {\scriptstyle\theta}\Downarrow{\scriptstyle cons} & & {\scriptstyle\theta'}\Downarrow \\ K' & \overset{\sigma'}{\Longrightarrow} & N' \end{array}}{\begin{array}{c} N \\ {\scriptstyle\theta'}\Downarrow{\scriptstyle cons} \\ N' \end{array}} \quad \text{if} \quad \begin{array}{ccc} \Sigma^K & \xrightarrow{\sigma} & \Sigma^N \\ \downarrow{\scriptstyle\theta} & & \downarrow{\scriptstyle\theta'} \\ \Sigma^{K'} & \xrightarrow{\sigma'} & \Sigma^{N'} \end{array}$$

is weakly amalgamable and N' is isolated.

(Cons-Shift)

Soundness of **(Cons-Shift)**: Assume that $K =\!\!\underset{cons}{=\!\!=}\!\!\Rightarrow K'$ is conservative. We have to prove that $N =\!\!\underset{cons}{=\!\!=}\!\!\Rightarrow N'$ is conservative as well. Let M be an N-model. Since $K =\!\!\underset{cons}{=\!\!=}\!\!\Rightarrow K'$ is conservative, $M|_\sigma$ has a θ-expansion M' being an K'-model. By weak amalgamation, there is some $\Sigma^{N'}$-model M' with $M'|_{\sigma'} = M'$ and $M'|_{\theta'} = M$. Since N' is isolated, M' is an N'-model. □

$$\frac{\begin{array}{ccc} K & \overset{\sigma}{=\!\!=\!\!\Rightarrow} & N \\ {\scriptstyle\theta}\Downarrow{\scriptstyle def} & & {\scriptstyle\theta'}\Downarrow \\ K' & \overset{\sigma'}{\Longrightarrow} & N' \end{array}}{\begin{array}{c} N \\ {\scriptstyle\theta'}\Downarrow{\scriptstyle def} \\ N' \end{array}} \quad \text{if} \quad \begin{array}{ccc} \Sigma^K & \xrightarrow{\sigma} & \Sigma^N \\ \downarrow{\scriptstyle\theta} & & \downarrow{\scriptstyle\theta'} \\ \Sigma^{K'} & \xrightarrow{\sigma'} & \Sigma^{N'} \end{array}$$

is amalgamable and N' is isolated.

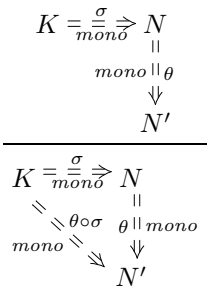
(Def-Shift)

Soundness of **(Def-Shift)**: assume that $K =\!\!\underset{def}{=\!\!=}\!\!\Rightarrow K'$ is definitional. We have to prove that $N =\!\!\underset{def}{=\!\!=}\!\!\Rightarrow N'$ is definitional as well. Let M be an N-model. By the argument used for the proof of soundness of **(Cons-shift)**, M has a θ'-expansion to an N'-model M'. Now let M'' be another N'-model with $M''|_{\theta'} = M = M'|_{\theta'}$. Then $M'|_{\sigma'\circ\theta} = M'|_{\theta'\circ\sigma} = M''|_{\theta'\circ\sigma} = M''|_{\sigma'\circ\theta}$. Since $K =\!\!\underset{def}{=\!\!=}\!\!\Rightarrow K'$ is definitional, $M'|_{\sigma'} = M''|_{\sigma'}$. By uniqueness of the amalgamation, $M' = M''$. □

$$\frac{\begin{array}{ccc} K & \overset{\sigma}{=\!\!=\!\!\Rightarrow} & N \\ & {\scriptstyle cons}\searrow\!\!\!=\!\!\!\searrow{\scriptstyle\theta} & \\ & & N' \end{array}}{\begin{array}{ccc} K & \overset{\sigma}{=\!\!=\!\!\Rightarrow} & N \\ & {\scriptstyle\theta\circ\sigma}\searrow\!\!\!=\!\!\!\searrow{\scriptstyle cons} & {\scriptstyle\theta}\Downarrow{\scriptstyle cons} \\ & & N' \end{array}}$$

(Cons-Composition)

Soundness of **(Cons-Composition)**: any K-model can be σ-expanded to an N-model, which in turn can be θ-expanded to an N'-model. Hence, each K-model can be $\theta \circ \sigma$-expanded to an N'-model. □



if θ is transportable, any hiding link going directly or indirectly into N' has a transportable signature morphism, and satisfaction in the institution is closed under isomorphism

(Mono-Composition)

For the rule **(Mono-composition)**, we need some technical notion: call a signature morphism $\sigma: \Sigma_1 \longrightarrow \Sigma_2$ *transportable*, if for any Σ_1-model M_1 and Σ_2-model M_2 and any isomorphism $h_1: M_2|_\sigma \longrightarrow M_1$, there is a Σ_2-model M_2' and an isomorphism $h_2: M_2 \longrightarrow M_2'$ with $h_2|_\sigma = h_1$ (which of course includes $M_2'|_\sigma = M_1$). Usually, transportability can be characterized syntactically. For example we have:

Proposition 5.20 In the CASL institution, a signature morphism is transportable iff it is injective on sorts.

PROOF: Let a sort-injective $\sigma: \Sigma_1 \longrightarrow \Sigma_2$, a Σ_1-model M_1, a Σ_2-model M_2 and an isomorphism $h_1: M_2|_\sigma \longrightarrow M_1$ be given. M_2' is constructed by taking M_1, and extending it with the carriers of M_2 for sorts in $\Sigma_2 \setminus \Sigma_1$. Operations and predicates in $\Sigma_2 \setminus \Sigma_1$ are interpreted as in M_2, possibly composed with appropriate parts of h_1 whenever sorts from Σ_1 are involved as source or target sorts. This construction works if σ is injective on sorts. If not, take sorts s, t with $\sigma(s) = \sigma(t)$, take M_2 arbitrary and take M_1 as $M_2|_\sigma$ except that t^{M_1} is not s^{M_1}, but is replaced by some isomorphic copy of t^{M_1} (and again operations are composed with this iso if necessary). Then it is impossible to find a Σ_2-expansion of M_1. □

Soundness of **(Mono-Composition)**: we first show that the model class of N' is closed under isomorphism. Let $len(p)$ be the length of a path p witnessing $P \stackrel{\tau}{\Longrightarrow} N'$. Let $maxp$ be the maximal such length in $\mathcal{DG}$. We show that for any $P \stackrel{\tau}{\Longrightarrow} N'$, the model class of P is closed under isomorphism. We proceed by induction over $maxp - len(p)$ for p witnessing $P \stackrel{\tau}{\Longrightarrow} N'$. We have to show that the conditions of clauses 1 to 5 of Definition 5.10 are invariant under model isomorphism:

1. Holding of sentences in a model is invariant under model isomorphism, by the assumption that satisfaction in the institution is closed under isomorphism.
2. Since reduct functors preserve isomorphisms, we can apply the induction hypothesis.
3. Here, a combination of the above two arguments applies.
4. Let $K \stackrel{\sigma}{\underset{hide}{\Longrightarrow}} P \in \mathcal{DG}$, M' be a P-model and M'' be isomorphic to M'. Since M' is a P-model, it has a σ-expansion to an K-model M. By transportability of σ, there is a Σ^K-model M' isomorphic to M with $M'|_\sigma = M''$. By induction hypothesis, $\mathbf{Mod}(K)$ is closed under isomorphism, hence $M' \in \mathbf{Mod}(K)$ as well, and thus $M'' \in \mathbf{Mod}(P)$.
5. Freeness is closed under isomorphism.

This completes the induction. Since $N' \stackrel{id}{\Longrightarrow} N'$, the model class of N' is closed under isomorphism.

Now we come to monomorphicity of the theorem link $K = \stackrel{\theta\circ\sigma}{=} \Rightarrow N'$. Let M be an K-model. By the two monomorphicity assumptions, it has at least one N'-expansion. So it remains to prove that all N'-expansions are isomorphic. Let M_3 and M_3' be two N'-expansions of M. By monomorphicity of $K = \stackrel{\sigma}{=} \Rightarrow N$, $M_3|_\theta$ and ${M_3}'|_\theta$ are isomorphic. By transportability of θ, there is some M_3'' isomorphic to M_3 with ${M_3}''|_\theta = {M_3}'|_\theta$. Since the model class of N' is closed under isomorphism, M_3'' is an N'-model as well. By monomorphicity of $N = \stackrel{\theta}{=} \Rightarrow N'$, M_3'' is isomorphic to M_3'. □

$$\frac{\begin{array}{c} K = \stackrel{\sigma}{\underset{def}{=}} \Rightarrow N \\ \qquad\qquad\quad \| \\ \qquad\qquad def \,\|\, \theta \\ \qquad\qquad\quad \Downarrow \\ \qquad\qquad\quad N' \end{array}}{\begin{array}{c} K = \stackrel{\sigma}{\underset{def}{=}} \Rightarrow N \\ \searrow \quad\quad \| \\ \quad \theta\circ\sigma \quad \theta \,\|\, def \\ def \searrow \quad\quad \Downarrow \\ \qquad\quad N' \end{array}}$$

(Def-Composition)

Soundness of **(Def-Composition)**: a global theorem link $K = \stackrel{\sigma}{=} \Rightarrow N$ is definitional iff $\mathbf{Mod}(\sigma)\colon \mathbf{Mod}(N) \longrightarrow \mathbf{Mod}(K)$ is bijective. Bijective maps compose. □

$$\frac{K = \stackrel{\sigma}{\underset{def}{=}} \Rightarrow N}{K = \stackrel{\sigma}{\underset{mono}{=}} \Rightarrow N}$$

(Def-to-mono)

Soundness of **(Def-to-mono)**: obvious. □

$$\frac{K = \stackrel{\sigma}{\underset{mono}{=}} \Rightarrow N}{K = \stackrel{\sigma}{\underset{cons}{=}} \Rightarrow N}$$

(Mono-to-cons)

Soundness of **(Mono-to-cons)**: obvious. □

$$\frac{\begin{array}{c} K \stackrel{\sigma}{\underset{free}{\Longrightarrow}} N \\ P = \stackrel{\sigma}{\underset{cons}{=}} \Rightarrow N \end{array}}{P = \stackrel{\sigma}{\underset{mono}{=}} \Rightarrow N}$$

(Free-is-mono)

Soundness of **(Free-is-mono)**: by the second premise, each P-model has a σ-expansion to an N-model. It remains to show that these σ-expansions are unique up to isomorphism. But this follows since N-models are free (and hence unique up to isomorphism) over their σ-reducts. (Notice that the same signature morphism is used in both premises.) □

$$\frac{\underset{mono}{K} \; = = \overset{\sigma}{=} = = \Rightarrow N}{\underset{mono}{K} \; = = \underset{free}{\overset{\sigma}{=}} \; ! \; = = \Rightarrow N}$$

(Mono-is-free)

Recall that $!: \emptyset \longrightarrow \Sigma^K$ is the signature morphism starting from the initial signature.

Soundness of **(Mono-is-free)**: recall that the free theorem link holds if for any N-model M, $M|_\sigma$ is an K-model that is !-free in $\mathbf{Mod}_{\mathcal{DG}}(K)$. Now for an N-model M, $M|_\sigma$ is an K-model by the premise of the rule, and it is !-free since in a monomorphic model class, any model is initial (and initiality is just !-freeness). □

5.6.4 Simple Structural Rules

The calculus finally provides a set of decomposition rules not interacting with hiding nor freeness, and a rule allowing for reducing local implications to inference in the calculus of the logic for basic specifications.

$$\frac{K \overset{\sigma}{\Longrightarrow\!\!\!\!\!\!\rightarrow} N}{K = \overset{\sigma}{=} \Rightarrow N}$$

(Subsumption)

Soundness of **(Subsumption)**: Obvious. □

$$\frac{P - \overset{\sigma}{-} \succ Q}{P - \overset{\tau}{-} \succ K} \text{ if } Q \overset{\theta}{\Longrightarrow\!\!\!\!\!\!\rightarrow} K \text{ and } \tau(\Psi^P) = \theta(\sigma(\Psi^P))$$

(Loc-Decomposition I)

Soundness of **(Loc-Decomposition I)**: assume $\mathcal{DG} \models P - \overset{\sigma}{-} \succ Q$ and $Q \overset{\theta}{\Longrightarrow\!\!\!\!\!\!\rightarrow} K$ and $\tau(\Psi^P) = \theta(\sigma(\Psi^P))$. In order to show $\mathcal{DG} \models P - \overset{\tau}{-} \succ K$, let M be an K-model. By Prop. 5.13, $M|_\theta$ is a Q-model, and by the assumption, $M|_{\theta\circ\sigma} \models \Psi^P$. By the satisfaction condition for institutions, $M \models \theta \circ \sigma(\Psi^P) = \tau(\Psi^P)$. Again by the satisfaction condition, $M|_\tau \models \Psi^P$. □

$$\frac{K > \overset{\theta}{\Longrightarrow\!\!\!\!\!\!\rightarrow} N}{K - \overset{\sigma}{-} \succ N} \text{ if } \sigma(\Psi^K) = \theta(\Psi^K)$$

(Loc-Decomposition II)

Soundness of **(Loc-Decomposition II)**: assume that $K > \overset{\theta}{\Longrightarrow\!\!\!\!\!\!\rightarrow} N$ and $\sigma(\Psi^K) = \theta(\Psi^K)$. Let M be an N-model. By Proposition 5.13, $M|_\theta \models \Psi^K$. By the satisfaction condition for institutions, $M \models \theta(\Psi^K) = \sigma(\Psi^K)$. Again by the satisfaction condition, $M|_\sigma \models \Psi^K$. □

$$\frac{N \Rightarrow \sigma(\Psi^K)}{K - \overset{\sigma}{-} \succ N}$$

(Local Inference)

$$\frac{K - \overset{\sigma}{-} \succ N}{N \Rightarrow \sigma(\Psi^K)}$$

(Reverse Local Inference)

Soundness of **(Local Inference)**: assume that $M \models \sigma(\Psi^K)$ for each N-model M. In order to show $\mathcal{DG} \models K - \overset{\sigma}{-} \succ N$, let M be an N-model. By assumption, $M \models \sigma(\Psi^K)$. By the satisfaction condition for institutions, $M|_\sigma \models \Psi^K$. Soundness of **(Reverse Local Inference)** is shown by reversing this argument. □

$$\frac{Th_{\mathcal{DG}}(N) \vdash_{\Sigma^N} \varphi \text{ for each } \varphi \in \Psi}{N \Rightarrow \Psi}$$

(Basic Inference)

Soundness of **(Basic Inference)**: assume that $Th_{\mathcal{DG}}(N) \vdash_{\Sigma^N} \varphi$ for each $\varphi \in \Psi$. By soundness of $\vdash_{\Sigma^N}$, we get $Th_{\mathcal{DG}}(N) \models_{\Sigma^N} \Psi$. In order to show $\mathcal{DG} \models N \Rightarrow \Psi$, let M be an N-model. By Proposition 5.14, $M \models Th_{\mathcal{DG}}(N)$. Since $Th_{\mathcal{DG}}(N) \models_{\Sigma^N} \Psi$, also $M \models \Psi$.

5.6.5 Soundness and Completeness

Proposition 5.21 The rules in Sect. 5.6 are sound.

PROOF: For each rule, in Sect. 5.6, a soundness proof has been given. □

Another question is the completeness of our rules. We have the following counterexample:

Proposition 5.22 Let *FOL* be the usual first-order logic with a recursively axiomatized complete entailment system. The problem to decide whether a global theorem link holds in a development graph with hiding over *FOL* is not recursively enumerable. Thus, any recursively axiomatized calculus for development graphs with hiding is incomplete.

PROOF: This can be seen as follows. Let Σ be the *FOL*-signature with a sort *nat* and operations for zero and successor, addition and multiplication. Consider the axiom set consisting of the usual second-order Peano axioms characterizing the natural numbers uniquely up to isomorphism, plus the defining axioms for addition and multiplication. Without loss of generality, we can assume that these axioms are combined into a single axiom of the form

$$\forall P\colon pred(nat) \;.\; \varphi$$

where φ is a first-order formula. Let ψ be any sentence over Σ. Let $\theta\colon \Sigma \longrightarrow \Sigma'$ add a predicate $P : pred(nat)$ to Σ. Consider the development graph

$$\begin{array}{ccc} \textsc{Peano} & \overset{h}{\underset{\theta}{\Longleftarrow}} & \textsc{PeanoDef} \\ \| & & \\ \|_{id} & & \\ \Downarrow & & \\ \Sigma & & \end{array}$$

where Σ and PEANO are nodes with signature Σ and no local axioms, whereas PEANODEF is a node with signature Σ' and local axiom $\varphi \Rightarrow \psi$.

Now we have that $\textsc{Peano} \overset{id}{=\!=\!\Rightarrow} \Sigma$ holds iff each Σ-model has a PEANODEF-expansion. It is easy to see that this holds iff the second-order formula $\exists P : pred(nat).\varphi \Rightarrow \psi$ is valid. This is

equivalent to $(\forall P : pred(nat).\varphi) \models \psi$, i.e. equivalent to the fact that ψ holds in the second-order axiomatization of Peano arithmetic. By Gödel's incompleteness theorem [Sho67], the problem to decide whether this holds is not recursively enumerable. □

In spite of this negative result, there is still the question of a relative completeness w.r.t. a given oracle deciding conservative extensions. Such a completeness result has been proved by Borzyszkowski [Bor02] in a similar setting. In Sect. 6.4, we are going to prove an analogous result, which additionally is based on oracles for freeness (the latter has not been covered by Borzyszkowski).

An *oracle for conservative extensions* is a sound logic-specific rule that allows to infer conservativity annotations for global definition links. It is called complete if for any global definition links that enjoys the model expansion property, the conservativity annotation may actually be inferred.

An *oracle for free theorem links* is a sound logic-specific rule that allows to infer free theorem links. It is called complete if any free theorem link that semantically holds also can be inferred by the rule.

An *elimination oracle for free definition links* is a sound logic-specific rule of the form

$$\frac{K = \stackrel{\sigma}{=} \Rightarrow P}{K = \stackrel{\sigma}{=} \Rightarrow N}$$

where $K = \stackrel{\sigma}{=} \Rightarrow N$ is arbitrary and P is constructed out of N such that P does not contain any directly or indirectly ingoing free definition links. Here, soundness just means $\mathbf{Mod}(N) \subseteq \mathbf{Mod}(P)$. Such a rule is called *complete*, if also $\mathbf{Mod}(P) \subseteq \mathbf{Mod}(N)$.

Theorem 5.23 (Completeness) Assume that the underlying logic is complete. Then the rule system for development graphs with hiding is complete relative to complete oracles for conservative extensions and free theorem links and a complete elimination oracle for free definition links.

Proof:

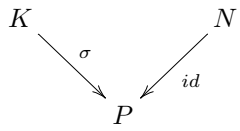
See [MAH]. A completeness theorem for an extended set of rules in a heterogeneous setting will be given in Sect. 6.4.

□

Corollary 5.24 If the underlying logic is complete, the simple structural rules are complete for proving theorem links between flattenable nodes.

Proof: See [MAH01]. □

We should note that a complete oracle for conservative extensions is very powerful: it can be used to obtain a complete proof calculus for development graphs. Namely, in order to decide whether $\mathcal{DG} \models K = \stackrel{\sigma}{=} \Rightarrow N$, we just add a node P with



and ask the oracle whether $N \stackrel{id}{\Longrightarrow} P$ is conservative.

Nevertheless, our completeness theorem is still meaningful. This is because the completeness proof uses the oracle for conservative extensions only in a limited way. The extensions considered are those obtained from hiding theorem links in the development graph (pushed along some morphism into a 'big' signature collecting everything). This means, for example, if we use hiding links only to hide symbols that have been defined using some logic-specific definition scheme, we will need the oracle for conservative extensions only for checking this definition scheme.

We cannot expect to check conservativity independently of the underlying institution. Therefore, institution-specific rules are needed. See Sect. 3.1.5 for checking conservativity in CASL.

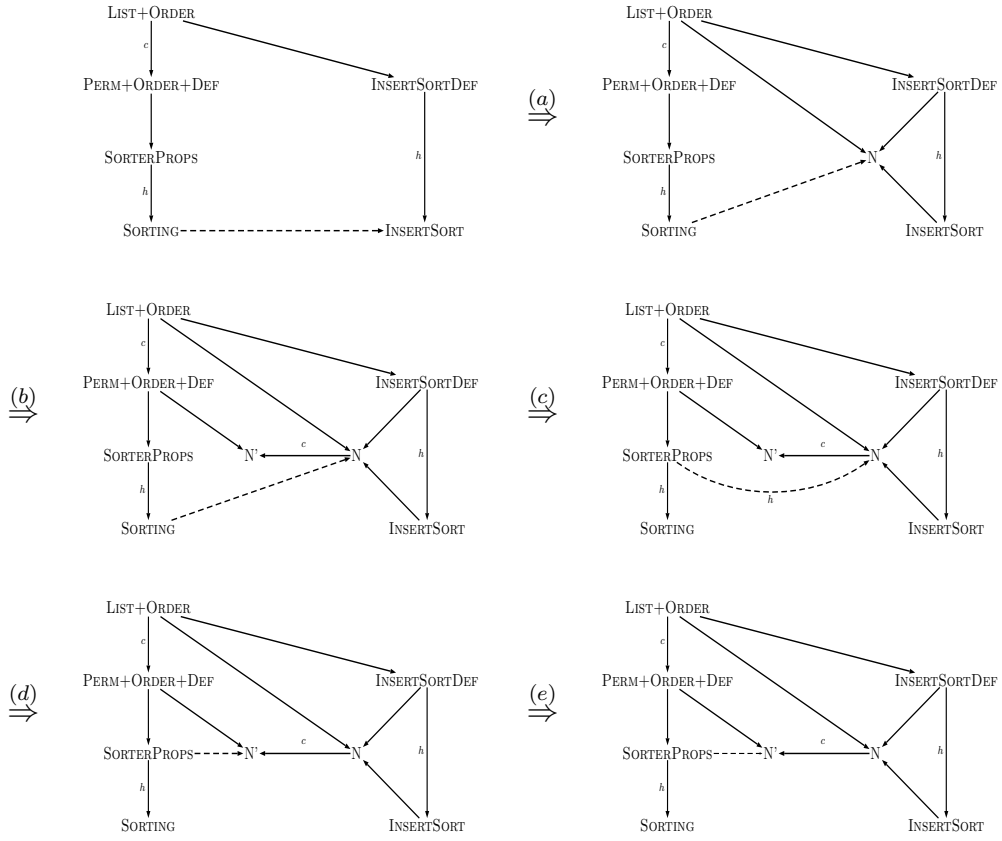


Figure 5.4: Reduction of theorem links in the running example.

5.7 A Sample Derivation in the Development Graph Calculus

We now demonstrate the (backward manner) use of the rules with the example development graph from Sect. 5.4. The goal is to reduce the theorem link between Sorting and InsertSort to theorem links between flattenable nodes. The derivation is shown in Fig. 5.4. In the first step (a) the *Theorem-Hide-Shift* rule is applied, which introduces the new node N and the new global definition links. In the second step (b), we infer conservative relationships by applying the rule *Cons-Shift*. This introduces the new node N' and the respective global definition links. Now the theorem link can be reduced to a hiding theorem link from SorterProps to N by *Glob-Decomposition* (step (c)). Now, this hiding theorem link can be reduced to the theorem link between SorterProps and N' using the rule *Hide-Theorem-Shift* (step (d)). Finally, using *Glob-Decomposition* again, we get three local theorem links, two of which can be immediately discarded with *Subsumption* (step (e)). The remaining local theorem link can then be proved by reasoning in the logic (via *Local Inference* and *Basic Inference*).

5.8 Bibliographical Notes

A kernel language for structured specifications in an arbitrary institution has been proposed by Sannella and Tarlecki [ST88b]. Borzyszkowski [Bor02] has provided a sound and (relatively) complete proof system. Note that these structured specifications are based on a model-theoretic semantics for specifications; there are other formalisms with a proof theoretic semantics [RG04].

There are quite a number of institution independent languages for structured specifications [ST88b, EM90a, DGS91, GT00, DM99, Mos97], one of which also has been extended to the heterogeneous case [Tar00]. Most of their constructs can be translated into the formalism of development graphs, which hence can be seen as a core formalism for structured theorem proving. For the language CASL, such a translation has been laid out explicitly in [AHMS00, CoF04, MAH], [MAH] also covers a few more languages.

Likewise, it should not be difficult to translate various module systems for *programming languages* to the formalism of development graphs (as it was done for the module system of the CASL specification language). Typically, imports of program modules will lead to definition links, while encapsulation of modules via some export interface will lead to hiding links. Actually, a translation from the Haskell module system to development graphs has been implemented in the Heterogeneous Tool Set (see Chap. 7).

The calculus for development graphs has been published in [MAH01] and [CoF04].

Chapter 6

Foundations of Heterogeneous Specification

The most prominent approach to heterogeneous specification so far is CafeOBJ with its cube of eight logics and twelve projections (formalized as *institution morphisms*) among them [DF02], and having a semantics based on Diaconescu's notion of Grothendieck institution [Dia02]. However, this approach has a limitation: only one type of translation between institution is used, namely institution morphisms. As we have seen in Chap. 4, institution comorphisms, forward morphisms and semi-morphisms arise naturally as well.

Tarlecki's treatment [Tar00] is more general; he introduces a heterogeneous constructs for all the different kinds of mappings between institutions that have been introduced in Chap. 2. Also, the beginnings of a proof calculus are provided [Tar00]. However, this calculus only addresses the question of entailment of a sentence in a specification, and not that of refinement between specifications. Moreover, completeness of the calculus is an open problem. The goal of the present chapter is to overcome these limitations while simultaneously staying as simple as possible.

The general idea is to start with a graph of institutions and morphisms, comorphisms and other types of translations, and then flatten this graph, using a so-called *Grothendieck construction*. This construction can be thought of an enriched disjoint union of the institutions involved: while the institutions are embedded as they are into the Grothendieck construction, the latter has not only inter-institution signature morphisms, but also new signature morphisms that correspond to intra-institution translations.

In this way, heterogeneous specification can be viewed as structured specification over a Grothendieck construction. In Sect. 6.1, we start with the most simple case of the Grothendieck institution over a graph of institution and comorphisms (the latter is formalised as a so-called *indexed coinstitution*), prove semantic properties (Sect. 6.2) and discuss several proof methods for the heterogeneous setting (Sect. 6.3–6.5). Later on, we proceed to the more complex case of mixed types of translations (Sect. 6.7–6.12).

6.1 Comorphism-Based Grothendieck Institutions

The basic data for comorphism-based heterogeneous specification is a graph of institutions, comorphisms and modifications. Remember from Sect. 2.12 that the modifications are needed because we want to express that certain compositions of comorphisms are the same. This means that we need to specify both compositions and modifications. The former amounts to starting with a category instead of just a graph, while the latter amounts to starting with a *2-category*, where the 2-cells correspond to modifications. We hence arrive at the following:

Definition 6.1 Given an index 2-category Ind, a *2-indexed coinstitution* is a 2-functor $\mathcal{I}: Ind^* \longrightarrow$

CoIns[1] into the 2-category of institutions, institution comorphisms and institution comorphism modifications. In cases where the 2-categorical structure is not needed, we omit the prefix "2-" and write $\mathcal{I}: Ind^{op} \longrightarrow \mathbf{CoIns}$.

Indeed, the name "2-indexed coinstitution" is justified by the fact that the category of institutions and institution comorphisms is isomorphic to the category of coinstitutions and coinstitution morphisms. A coinstitution is an institution with model translations covariant to signature morphisms, while sentence translations are contravariant.

A 2-indexed coinstitution can be flattened, using the so-called *Grothendieck construction*. The basic idea here is that all signatures of all institutions are put side by side, and a signature morphism in this large realm of signatures consists of an intra-logic signature morphism plus an inter-logic translation (along some logic comorphism). The other components (sentences, models, satisfaction) are then defined in a straightforward way.

The Grothendieck construction for indexed institutions has been described in [Dia02]; we develop its dual here. In order to keep the notation simple, we will rely on the definition of institutions as functors given in Sect. 2.13. In an indexed coinstitution $\mathcal{I}$, we use the notation $\mathcal{I}^i = (\mathbf{Sign}^i, \mathbf{Sen}^i, \mathbf{Mod}^i, \models^i)$ for $\mathcal{I}(i)$, (Φ^d, ρ^d) for the comorphism $\mathcal{I}(d)$, and $\mathcal{I}^u$ for the modification $\mathcal{I}(u)$.

Definition 6.2 Given a 2-indexed coinstitution $\mathcal{I}: Ind^* \longrightarrow \mathbf{CoIns}$, define the *Grothendieck institution* $\mathcal{I}^\#$ as follows:

- signatures in $\mathcal{I}^\#$ are pairs (i, Σ), where $i \in |Ind|$ and Σ a signature in $\mathcal{I}^i$,
- signature morphisms $(d, \sigma): (i, \Sigma_1) \longrightarrow (j, \Sigma_2)$ consist of a morphism $d: j \longrightarrow i \in Ind$ and a signature morphism $\sigma: \Phi^d(\Sigma_1) \longrightarrow \Sigma_2$ in $\mathcal{I}^j$,
- composition is given by $(d_2, \sigma_2) \circ (d_1, \sigma_1) = (d_1 \circ d_2, \sigma_2 \circ \Phi^{d_2}(\sigma_1))$,
- $\mathcal{I}^\#(i, \Sigma) = \mathcal{I}^i(\Sigma)$, and $\mathcal{I}^\#(d, \sigma) = \mathcal{I}^i(\Sigma_1) \xrightarrow{\rho^d} \mathcal{I}^j(\Phi^d(\Sigma_1)) \xrightarrow{\mathcal{I}^j(\sigma)} \mathcal{I}^j(\Sigma_2)$.

That is, the room $\mathcal{I}^\#(i, \Sigma)$ (consisting of sentences, models and satisfaction) for a Grothendieck signature (i, Σ) is defined component wise, while the corridor for a Grothendieck signature morphism is obtained by composing the corridor given by the inter-institution comorphism with that given by the intra-institution signature morphism. We also denote the Grothendieck institution by $(\mathbf{Sign}^\#, \mathbf{Sen}^\#, \mathbf{Mod}^\#, \models^\#)$.

While the comorphism based Grothendieck construction nearly satisfies all of our needs, one problem remains. Sometimes, the Grothendieck construction makes too many distinctions between signature morphisms (cf. Fig. 2.1). Therefore, we use the institution comorphism modifications to obtain a congruence on Grothendieck signature morphisms: the congruence is generated by

$$(d', \mathcal{I}^u_\Sigma: \Phi^{d'}(\Sigma) \longrightarrow \Phi^d(\Sigma)) \equiv (d, id: \Phi^d(\Sigma) \longrightarrow \Phi^d(\Sigma)) \tag{6.1}$$

for $\Sigma \in \mathbf{Sign}^i$, $d, d': j \longrightarrow i \in Ind$, and $u : d \Rightarrow d' \in Ind$. This congruence has the following crucial property:

Proposition 6.3 $\equiv$ is contained in the kernel of $\mathcal{I}^\#$ (considered as a functor).

PROOF: By the definition of comorphism modification, $(\mathcal{I}^j \cdot \mathcal{I}^u) \circ \rho^{d'} = \rho^d$. But this just means that equivalent signature morphisms induce the same corridors. □

Let $q^{\mathcal{I}}: \mathbf{Sign}^\# \longrightarrow \mathbf{Sign}^\# / \equiv$ be the quotient functor induced by $\equiv$ (see [Lan72] for the definition of quotient category). Note that it is the identity on objects. We easily obtain that the functor $\mathcal{I}^\#$ factors through the quotient category $\mathbf{Sign}^\# / \equiv$:

[1] Ind^* is the 2-categorical dual of Ind, where both 1-cells and 2-cells are reversed.

Corollary 6.4 $\mathcal{I}^{\#}\colon \mathbf{Sign}^{\#} \longrightarrow \mathbf{InsRoom}$ leads to a quotient Grothendieck institution $\mathcal{I}^{\#}/\equiv : \mathbf{Sign}^{\#}/\equiv \longrightarrow \mathbf{InsRoom}$.

By abuse of notation, we denote $\mathcal{I}^{\#}/\equiv$ by $(\mathbf{Sign}^{\#}/\equiv, \mathbf{Sen}^{\#}, \mathbf{Mod}^{\#}, \models^{\#})$.

When considering e.g. the comorphism going from partial first-order logic $PFOL^{=}$ to first-order logic $FOL^{=}$, and the composite comorphism going from $PFOL^{=}$ to CASL and then to $FOL^{=}$, we end up in different comorphisms, which are however related by a comorphism modification. The above identification process in the Grothendieck institution now tells us that it does not matter which way we choose.

Proposition 6.5 Assume that Ind has cocones for diagrams of 2-cells of shape $\bullet \Longrightarrow \bullet \Longleftarrow \bullet$ that are mapped to pushouts of 2-cells in **CoIns**. Then the congruence $\equiv$ defined above is explicitly given by

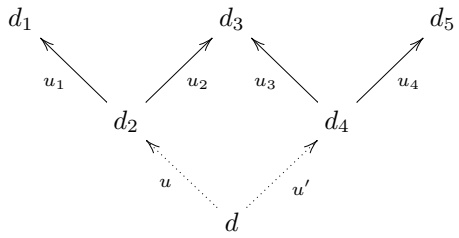
$$(d_1, \sigma \circ \mathcal{I}_\Sigma^{u_1}) \equiv (d_2, \sigma \circ \mathcal{I}_\Sigma^{u_2})$$

for $\Sigma \in \mathbf{Sign}^i$, $d, d_1, d_2 \colon j \longrightarrow i \in Ind$, $\sigma \colon \Phi^d(\Sigma) \longrightarrow \Sigma' \in \mathbf{Sign}^j$ and $u_1 : d \Rightarrow d_1, u_2 : d \Rightarrow d_2 \in Ind$.

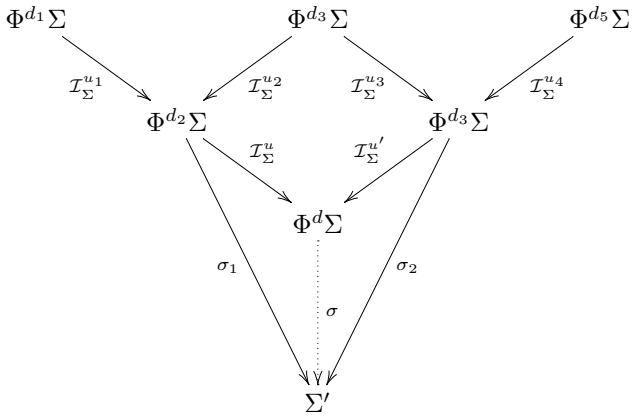
PROOF: It is easy to see that the above relation is contain in the relation generated by (6.1): just apply (6.1) twice. It remains to show that the above relation is a congruence. Reflexivity and symmetry are clear. Concerning transitivity, assume that

$$(d_1, \sigma_1 \circ \mathcal{I}_\Sigma^{u_1}) \equiv (d_3, \sigma_1 \circ \mathcal{I}_\Sigma^{u_2}) = (d_3, \sigma_2 \circ \mathcal{I}_\Sigma^{u_3}) \equiv (d_5, \sigma_2 \circ \mathcal{I}_\Sigma^{u_4}),$$

the first relation being witnessed by $u_1 : d_2 \Rightarrow d_1, u_2 : d_2 \Rightarrow d_3$, and the second by by $u_3 : d_4 \Rightarrow d_3, u_4 : d_4 \Rightarrow d_5$. Take the pullback in $Ind(j, i)$ of the two spans



By the construction of pushouts of 2-cells in **CoIns** (see Prop.2.37), the middle square in



is a pushout, and the mediating morphism σ leads to the desired form

$$(d_1, \sigma_1 \circ \mathcal{I}_\Sigma^{u_1}) = (d_1, \sigma \circ \mathcal{I}_\Sigma^{u_1 \circ u}) \equiv (d_5, \sigma \circ \mathcal{I}_\Sigma^{u_4 \circ u'}) = (d_5, \sigma_2 \circ \mathcal{I}_\Sigma^{u_4}).$$

Concerning composition, assume that

$$(d_1, \sigma \circ \mathcal{I}_\Sigma^{u_1}) \equiv (d_2, \sigma \circ \mathcal{I}_\Sigma^{u_2})$$

via $u_1 : d \Rightarrow d_1, u_2 : d \Rightarrow d_2$, and

$$(e_1, \tau \circ \mathcal{I}_{\Sigma'}^{v_1}) \equiv (e_2, \tau \circ \mathcal{I}_{\Sigma'}^{v_2})$$

via $v_1 : e \Rightarrow e_1, v_2 : e \Rightarrow e_2$. Then for $k = 1, 2$,

$$\begin{array}{lll} & (e_k, \sigma \circ \mathcal{I}_{\Sigma}^{u_k}) \circ (d_k, \tau \circ \mathcal{I}_{\Sigma'}^{v_k}) & \\ = & (d_k \circ e_k, \sigma \circ \mathcal{I}_{\Sigma}^{u_k} \circ \Phi^{e_k}(\tau) \circ \Phi^{e_k}(\mathcal{I}_{\Sigma'}^{v_k})) & \text{(definition of Grothendieck composition)} \\ = & (d_k \circ e_k, \sigma \circ \Phi^{e_k}(\tau) \circ \Phi^{e_k}(\mathcal{I}_{\Sigma'}^{v_k}) \circ \mathcal{I}_{\Phi^{e_k}(\Sigma')}^{u_k}) & \text{(naturality of } \mathcal{I}^{u_k}) \\ = & (d_k \circ e_k, \sigma \circ \Phi^{e_k}(\tau) \circ \mathcal{I}_{\Sigma'}^{v_k \cdot u_k}) & \text{(functoriality of } \mathcal{I}) \end{array}$$

which shows that we arrive at the desired form. □

Note that according to Prop. 2.37, under relatively mild assumptions, pushouts of 2-cells in **CoIns** exist. Hence, the assumption of Prop. 6.5 that Ind has cocones for diagrams of 2-cells of shape $\bullet \Longrightarrow \bullet \Longleftarrow \bullet$ that are mapped to pushouts of 2-cells in **CoIns** is quite realistic. In particular, it is possible to add suitable cocones to Hom-categories in Ind and interpret these as pushouts in **CoIns**.

6.2 Amalgamation and Exactness

The importance of amalgamation and exactness properties has been explained in Sect. 2.3. The theory of amalgamation and exactness in Grothendieck institutions for indexed institutions has been developed by Diaconescu [Dia02]. Actually, the corresponding theory for indexed coinstitutions turns out to be much simpler (of course, to compare complexity, we need to choose the 2-categorical structure to be trivial). Here, we focus on cocompleteness and amalgamation results, since these are needed for structured proofs (see Sect. 5.3 and 5.6).

Theorem 6.6 Let $\mathcal{I}\colon Ind^{op} \longrightarrow \mathbf{CoIns}$ be an indexed coinstitution and K be some small category such that

1. Ind is K-complete,
2. Φ^d is cocontinuous for each $d\colon i \longrightarrow j \in Ind$, and
3. the indexed category of signatures of $\mathcal{I}$ is locally K-cocomplete (the latter meaning that $\mathbf{Sign}^i$ is K-cocomplete for each $i \in |Ind|$).

Then the signature category $\mathbf{Sign}^\#$ of the Grothendieck institution has K-colimits.

PROOF: Apply Theorem 1 of [TBG91] with $C_i = \mathbf{Sign}^i$ and $C_m = \Phi^m$. Note that $\mathbf{Sign}^\#$ is then $Flat(C^{op})^{op}$. □

We cannot expect that this result directly carriers over to the quotient Grothendieck institution, since quotients of categories generally do not interact well with colimits. However, we can say something provided that we work with a quotient of the index category Ind:

Proposition and Definition 6.7 Given a 2-category Ind, the relation of being in the same connected component of a Hom-category defines a congruence $\equiv$ on the objects of the Hom-categories, i.e. the morphisms of Ind. $Ind/\equiv$ is the corresponding quotient 1-category.

Lemma 6.8 Given a 2-indexed coinstitution $\mathcal{I}\colon Ind^* \longrightarrow \mathbf{CoIns}$, if $(d_2, \sigma_1) \equiv (d_1, \sigma_2)$ in $\mathbf{Sign}^\#$, then $d_1 \equiv d_2$.

PROOF: Easy induction over the definition of $(d_1, \sigma_1) \equiv (d_2, \sigma_2)$. □

Proposition 6.9 Assume that Ind has cocones for diagrams of 2-cells of shape $\bullet \Longrightarrow \bullet \Longleftarrow \bullet$ that are mapped to pushouts of 2-cells in **CoIns**. Then the congruence $\equiv$ in Ind defined above is explicitly given by $d_1 : i \longrightarrow j \equiv d_2 : i \longrightarrow j$ iff there exist $d : i \longrightarrow j \in Ind$ and $u_1 : d \longrightarrow d_1, u_2 : d \longrightarrow d_2 \in Ind$.

PROOF: Analogous to the proof of Prop. 6.5. □

Theorem 6.10 Let $\mathcal{I} : Ind^* \longrightarrow \mathbf{CoIns}$ be a 2-indexed coinstitution such that

1. $Ind/\equiv$ is K-complete for some small category K,
2. each connected component (considered as a subcategory) of a Hom-category $Ind(i,j)$ has a distinguished canonical weakly terminal object, such that these canonical objects are stable under composition,
3. $(d, \sigma_1) \equiv (d, \sigma_2)$ in $\mathbf{Sign}^\#$ implies $\sigma_1 = \sigma_2$,
4. Φ^d is cocontinuous for each $d : i \longrightarrow j \in Ind$, and
5. the indexed category of signatures of $\mathcal{I}$ is locally K-cocomplete.

Then the signature category $\mathbf{Sign}^\#/\equiv$ of the quotient Grothendieck institution has K-colimits. (Note that assumptions 2 and 3 are vacuous in case of ordinary indexed coinstitutions; we then get Theorem 6.6 as a special case.)

PROOF: The proof idea follows that of Theorem 1 in [TBG91], the necessary modifications being caused by the congruences. By assumption 2, we can always choose representatives $d \in Ind$ of congruences classes $[d] \in Ind/\equiv$ in such a way that d is a canonical weakly terminal object. Similarly, we can always choose representatives (d, σ) of congruence classes $[(d, \sigma)]$ in $\mathbf{Sign}^\#/\equiv$ in such a way that d is the canonical weakly terminal object in its connected component: given an arbitrary $(d, \sigma : \Phi^d(\Sigma) \longrightarrow \Sigma')$ in $\mathbf{Sign}^\#$, let $u : d \Longrightarrow t$ be a 2-cell into the canonical weakly terminal object. Then $(t, \sigma \circ \mathcal{I}^u_\Sigma)$ is equivalent to (d, σ).

Given a diagram $D : K \longrightarrow \mathbf{Sign}^\#/\equiv$, we introduce the notation (i_k, Σ_k) for $D(k)$ $(k \in |K|)$ and $[(d_m, \sigma_m)] : (i_k, \Sigma_k) \longrightarrow (i_{k'}, \Sigma_{k'})$ for $D(m)$ $(m : k \longrightarrow k' \in K)$. Let $\bar{D} : K \longrightarrow Ind/\equiv$ be the projection of D to the first component; by Lemma 6.8 this is a well-defined diagram in $Ind/\equiv$. By assumption 1, $\bar{D}$ has a limit $([m_k] : i \longrightarrow i_k)_{k \in |K|}$.

Let the diagram $G : K \longrightarrow \mathbf{Sign}^i$ be defined by

$$G(k) = \Phi^{m_k}(\Sigma_k) \ (k \in |K|)$$
$$G(m) = \Phi^{m_k}(\sigma_m) \ (m : k' \longrightarrow k \in K)$$

Note that m_k is chosen to be canonical weakly terminal in $[m_k]$. By assumption 5, G has a colimit $(\sigma_k : G(k) \longrightarrow \Sigma)_{k \in |K|}$. We show that $([(m_k, \sigma_k)] : (i_k, \Sigma_k) \longrightarrow (i, \Sigma))_{k \in |K|}$ is a colimit of D.

Since equality implies congruence, $([(m_k, \sigma_k)])_{k \in |K|}$ is a cocone of D. Let $([(n_k, \theta_k)] : (i_k, \Sigma_k) \longrightarrow (i', \Sigma'))_{k \in |K|}$ be another cocone. By Lemma 6.8, $([n_k] : i' \longrightarrow i_k)_{k \in |K|}$ is a cocone for $\bar{D}$. Hence there is a unique $[d] : i' \longrightarrow i$ with $[m_k] \circ [d] = [n_k]$. Since we choose representatives canonically in a way closed under composition, $m_k \circ d = n_k$.

By assumption 4, $(\Phi^d(\sigma_k))_{k \in |K|}$ is a colimit of $\Phi^d \circ G$. Note that the source of $\Phi^d(\sigma_k)$ is $\Phi^d(G(k)) = \Phi^d(\Phi^{m_k}(\Sigma_k)) = \Phi^{n_k}(\Sigma_k)$. By the cocone property of $([(n_k, \theta_k)])_{k \in |K|}$, $(n_k, \theta_k) \equiv (d_m \circ n_{k'}, \theta_{k'} \circ \Phi^{n_{k'}}(\sigma_m))$ for $m : k \longrightarrow k' \in K$. By the assumption of weakly terminal canonical representatives, $n_k = d_m \circ n_{k'}$. By assumption 3, $\theta_k = \theta_{k'} \circ \Phi^{n_{k'}}(\sigma_m)$. This shows that $(\theta_k : \Phi^{n_k}(\Sigma_k) \longrightarrow \Sigma')_{k \in |K|}$ is a cocone for $\Phi^d \circ G$. Hence, there is a unique $\tau : \Phi^d(\Sigma) \longrightarrow \Sigma'$ with $\tau \circ \Phi^d(\sigma_k) = \theta_k$. Then $[(d, \tau)] : (i, \Sigma) \longrightarrow (i', \Sigma')$ is a unique morphism in $\mathbf{Sign}^\#/\equiv$ such that $[(d, \tau)] \circ [(m_k, \sigma_k)] = [(n_k, \theta_k)]$. □

By contravariancy of $\mathcal{I}$, assumption 2 of the above proposition means that if institution comorphisms are linked by modifications, there is always a "smallest" comorphism that can be embedded into the other ones. This is quite realistic in practice. However, it is not so realistic to assume that these smallest comorphisms are stable under composition. For example, the composition of the smallest embedding of $FOL^=$ into CASL with the smallest embedding of CASL into $SOL^=$ will give not given the smallest embedding of $FOL^=$ into $SOL^=$, but rather a more complex one.

Assumption 3 basically means that the congruence does not identify signature morphisms within one institution, i.e. that each signature category $\mathbf{Sign}^i$ is faithfully embedded into $\mathbf{Sign}^\#/\equiv$. This assumption is a reasonable and desirable property in practice. We record this explicity:

Proposition 6.11 Under the assumptions of Theorem 6.10, $emb^i\colon \mathbf{Sign}^i \longrightarrow \mathbf{Sign}^\#/\equiv$ is an embedding preserving colimits.

PROOF: Clearly, emb^i is injective on objects. Faithfulness follows from assumption 3. Preservation of colimits can be seen by inspecting the construction of the proof of Theorem 6.10: if the indices are all i, then the colimit is just that in $\mathbf{Sign}^i$. □

Let us now come to exactness. We extend the notion of semi-exactness (see Sect. 2.3) to the indexed case. An 2-indexed coinstitution $\mathcal{I}\colon Ind^* \longrightarrow \mathbf{CoIns}$ is called *(weakly) locally semi-exact*, if each institution I^i is (weakly) semi-exact ($i \in |Ind|$). Assuming that equivalence classes of 2-cells have canonical weakly terminal objects, $\mathcal{I}$ is called *(weakly) semi-exact* if for each pullback in $Ind/\equiv$

$$\begin{array}{ccc} i & \xleftarrow{[d_1]} & j1 \\ {\scriptstyle [d_2]}\uparrow & & \uparrow{\scriptstyle [e_1]} \\ j2 & \xleftarrow{[e_2]} & k \end{array}$$

the square

$$\begin{array}{ccc} \mathbf{Mod}^i(\Sigma) & \xleftarrow{\beta^{d_1}_\Sigma} & \mathbf{Mod}^{j1}(\Phi^{d_1}(\Sigma)) \\ {\scriptstyle \beta^{d_2}_\Sigma}\uparrow & & \uparrow{\scriptstyle \beta^{e_1}_\Sigma} \\ \mathbf{Mod}^{j2}(\Phi^{d_2}(\Sigma)) & \xleftarrow{\beta^{e_2}_\Sigma} & \mathbf{Mod}^k(\Phi^{e_1}(\Phi^{d_1}(\Sigma))) = \mathbf{Mod}^k(\Phi^{e_2}(\Phi^{d_2}(\Sigma))) \end{array}$$

is a (weak) pullback for each signature Σ in $\mathbf{Sign}^i$, where canonical weakly terminal representatives are used.

Theorem 6.12 Assume that the 2-indexed coinstitution $\mathcal{I}\colon Ind^* \longrightarrow \mathbf{CoIns}$ fulfills the assumptions of Theorem 6.10. Then the quotient Grothendieck institution $\mathcal{I}^\#/\equiv$ is (weakly) semi-exact if and only if

1. $\mathcal{I}$ is (weakly) locally semi-exact,
2. $\mathcal{I}$ is (weakly) semi-exact, and
3. for all canonical weakly terminal $d\colon i \longrightarrow j \in Ind$, in $\mathcal{I}^d$ is (weakly) exact.

PROOF: "Only if", 1: Following Prop. 2 in [Dia02], it is easy to see that for each $i \in |Ind|$, the model functor $\mathbf{Mod}^i$ is the restriction $\mathbf{Mod}^\#(i, \_)$ of the model functor of the Grothendieck

institution to the subcategory $\mathbf{Sign}^i$ of the Grothendieck signature category $\mathbf{Sign}^\#/\equiv$.

$$\begin{array}{ccc} (\mathbf{Sign}^i)^{op} & \xrightarrow{emb^i} & (\mathbf{Sign}^\#/\equiv)^{op} \\ & \searrow^{\mathbf{Mod}^i} & \downarrow{\scriptstyle \mathbf{Mod}^\#} \\ & & \mathcal{CAT} \end{array}$$

By Prop. 6.11, the canonical injection $emb^i\colon \mathbf{Sign}^i \longrightarrow \mathbf{Sign}^\#$ preserves colimits, hence $\mathbf{Mod}^i$ takes pushouts to (weak) pullbacks because $\mathbf{Mod}^\#$ does so.

"Only if", 2: Given a pullback in $Ind/\equiv$

$$\begin{array}{ccc} i & \xleftarrow{[d_1]} & j1 \\ {\scriptstyle [d_2]}\uparrow & & \uparrow{\scriptstyle [e_1]} \\ j2 & \xleftarrow{[e_2]} & k \end{array}$$

choose d_1, d_2, e_1, e_2 canonically. By the construction of colimits in Theorem 6.10, for any signature Σ in $\mathbf{Sign}^i$,

$$\begin{array}{ccc} (i,\Sigma) & \xrightarrow{[(d_1,id)]} & (j1, \Phi^{d_1}\Sigma \\ \downarrow{\scriptstyle [(d_2,id)]} & & \downarrow{\scriptstyle [(e_1,id)]} \\ (j2, \Phi^{d_2}\Sigma) & \xrightarrow{[(e_2,id)]} & (k, \Phi^{e_1}\Phi^{d_1}\Sigma) = (k, \Phi^{e_2}\Phi^{d_2}\Sigma) \end{array}$$

is a pushout in $\mathbf{Sign}^\#/\equiv$ and is therefore mapped to a (weak) pullback by the model functor. This gives exactly the desired property.

"Only if", 3: Let $d\colon j \longrightarrow i$ by canonical and $\sigma\colon \Sigma_1 \longrightarrow \Sigma_2$ a signature morphism in $\mathbf{Sign}^i$. By the construction of colimits in Theorem 6.10,

$$\begin{array}{ccc} (i,\Sigma_1) & \xrightarrow{[(id,\sigma)]} & (i,\Sigma_2) \\ \downarrow{\scriptstyle [(d,id)]} & & \downarrow{\scriptstyle [(d,id)]} \\ (j, \Phi^d\Sigma_1) & \xrightarrow{[(id,\Phi^d\sigma)]} & (j, \Phi^d\Sigma_2) \end{array}$$

is a pushout in $\mathbf{Sign}^\#/\equiv$ and is therefore mapped to a (weak) pullback by the model functor. Again, this gives exactly the desired property.

"If": Consider an arbitrary pushout in $\mathbf{Sign}^\#/\equiv$

$$\begin{array}{ccc} (i,\Sigma_0) & \xrightarrow{[(d_1,\sigma_1)]} & (j_1,\Sigma_1) \\ \downarrow{\scriptstyle [(d_2,\sigma_2)]} & & \downarrow{\scriptstyle [(e_1,\theta_1)]} \\ (j_2,\Sigma_2) & \xrightarrow{[(e_2,\theta_2)]} & (k,\Sigma') \end{array}$$

and assume that representatives are chosen canonically. By the construction of colimits in Theorem 6.10, the above pushout can be expressed as the following composition of four pushout squares:

$$\begin{array}{ccccc}
(i,\Sigma_0) & \xrightarrow{[(d_1,id)]} & (j_1,\Phi^{d_1}\Sigma_0) & \xrightarrow{[(id,\sigma_1)]} & (j_1,\Sigma_1) \\
\downarrow{\scriptstyle [(d_2,id)]} & & \downarrow{\scriptstyle [(e_1,id)]} & & \downarrow{\scriptstyle [(e_1,id)]} \\
(j_2,\Phi^{d_2}\Sigma_0) & \xrightarrow{[(e_2,id)]} & (k,\Phi^{e_1}\Phi^{d_1}\Sigma_0) = (k,\Phi^{e_2}\Phi^{d_2}\Sigma_0) & \xrightarrow{[(id,\Phi^{e_1}\sigma_1)]} & (k,\Phi^{e_1}\Sigma_1) \\
\downarrow{\scriptstyle [(id,\sigma_2)]} & & \downarrow{\scriptstyle [(id,\Phi^{e_2}\sigma_2)]} & & \downarrow{\scriptstyle [(id,\theta_1)]} \\
(j_2,\Sigma_2) & \xrightarrow{[(e_2,id)]} & (k,\Phi^{e_2}\Sigma_2) & \xrightarrow{[(id,\theta_2)]} & (k,\Sigma')
\end{array}$$

Now the model functor of the quotient Grothendieck institution maps the upper left pushout to a (weak) pullback because the 2-indexed coinstitution is (weakly) semi-exact, maps the lower right pushout to a (weak) pullback because the 2-indexed coinstitution is (weakly) locally semi-exact, and maps the remaining two squares to (weak) pullbacks because the comorphisms for canonical index morphisms are (weakly) exact. Since (weak) pullback squares compose, the result follows. □

Theorems 6.6, 6.10 and 6.12 already provide a good theoretical basis for heterogeneous specification. However, in some cases, these theorems are not general enough: Given a diagram $J \to Ind$, its limit must be the index of some institution that can serve to encode (via comorphisms) all the institutions indexed by the diagram. While the existence of such an institution may not be a problem (e.g. higher-order logic often serves as such a "universal" logic for coding other logics), the uniqueness condition imposed by the limit property is more problematic. This means that any two such "universal" institutions must have isomorphic indices and hence be isomorphic themselves. This might work well is some circumstances, but may not desirable in others: after all, a number of non-isomorphic logics, such as classical higher-order logic, the calculus of constructions and rewriting logic have been proposed as such a "universal" logic.[2]

A related problem[3] is that the assumptions of Theorem 6.12 are too strong to be met for all practical examples. E.g. the CASL institution is not weakly semi-exact, and its encoding into $HOL^=$ is neither exact, nor does it have a cocontinuous signature translation.

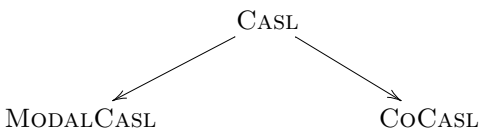
We hence now generalize the previous results by replacing weak exactness with quasi-exactness, i.e. amalgamable colimits with weakly amalgamable cocones, and thereby dropping the uniqueness requirement. Hence, several non-isomorphic "universal" institutions may coexist peacefully with our approach, and also non-exact institutions and comorphisms may be included in the indexed coinstitution serving as basis for heterogeneous specification.

We first extend Def. 2.8 to indexed coinstitutions:

Definition 6.13 An indexed coinstitution $\mathcal{I}\colon Ind^{op} \longrightarrow \mathbf{CoIns}$ is called *locally quasi-exact*, if each institution $\mathcal{I}^i$ is quasi-exact ($i \in |Ind|$). It is called *quasi-exact*, if for each diagram $D\colon J \longrightarrow Ind$, there is some cone $(l,(d_j)_{j\in|J|})$ over D whose image under $\mathcal{I}$ is weakly amalgamable. *Quasi-semi-exactness* is the restriction of these notions to diagrams of shape $\bullet \longleftarrow \bullet \longrightarrow \bullet$.

These notions will play a central role in Sect. 6.4. Here, we investigate their behaviour in Grothendieck institutions.

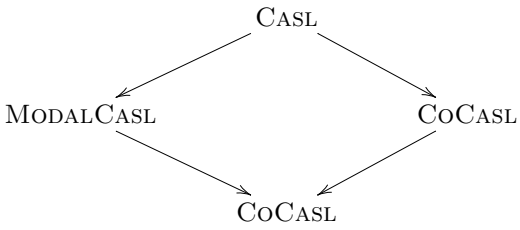
However, for the index level, even quasi-exactness may be too strong. Consider the diagram



[2]This problem can possibly be circumvented by formally adjoining limits to the index category, which are then interpreted using Grothendieck institutions over subdiagrams. However, this would add considerable complexity to the construction.

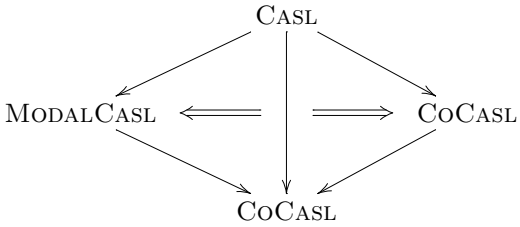
[3]This problem already has been noted by Diaconescu [Dia02] for his more special version of Theorem 6.12; see Sect. 6.7 why we consider it to be more special.

How do we obtain a weakly amalgamable cocone? A simple way is to use the embedding of ModalCasl into Casl (Sect. 4.2) and compose it with the inclusion of Casl into CoCasl:



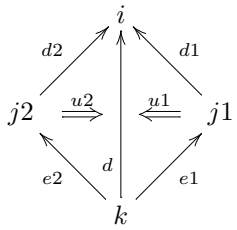
but the resulting square does not even commute.[4] The reason is that on the way from Casl to CoCasl via ModalCasl, ModalCasl adds an implicit set of worlds, which is made explicit by the embedding of ModalCasl into Casl.[5] To obtain a commuting square, we would need to have a comorphism from CoCasl to itself which adds an explicit set of worlds. However, this solution is rather inelegant, since it means that any (present of future) extension of Casl without possible world semantics (e.g. for HasCasl), we need a similar comorphism.

We hence prefer to split the square into two lax triangles:

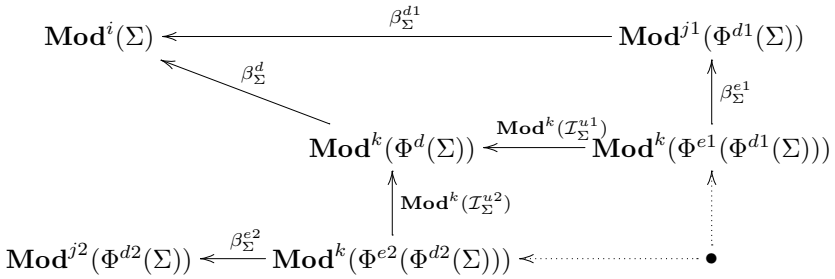


and indeed, the square weakly amalgamable in the following sense:

Definition 6.14 Given a 2-indexed coinstitution $\mathcal{I}\colon Ind^* \longrightarrow \mathbf{CoIns}$, a square consisting of two lax triangles of index morphisms



is called (weakly) amalgamable, if the following diagram is a (weak) pullback

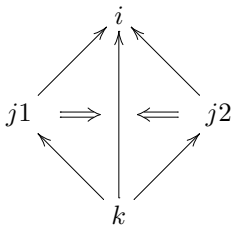


[4] Of course, we could also embed everything into HOL, which would not cause any relevant change to the subsequent discussion.

[5] See section 3.2.6 for the reason why the set of worlds cannot be omitted even for models of signatures without modalities.

where the lower right square is a pullback. That is, each pair consisting of a $\Phi^{d2}(\Sigma)$- and a $\Phi^{d1}(\Sigma)$-model with the same Σ-reduct is (weakly) amalgamable to a pair consisting of a $\Phi^{e2}(\Phi^{d2}(\Sigma))$- and a $\Phi^{e1}(\Phi^{d1}(\Sigma))$-model having the same $\Phi^{d}(\Sigma)$-reduct.

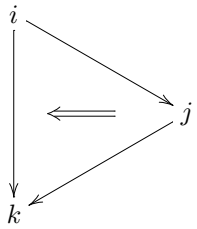
$\mathcal{I}$ is called *lax-quasi-exact*, if each for pair of arrows $j1 \xrightarrow{d_1} i \xleftarrow{d_2} j2$ in Ind, there is some square



consisting of a weakly amalgamable square of lax triangles, such that additionally $\mathcal{I}^k$ is quasi-semi-exact.

Note that this property is different from (and indeed, incomparable to) amalgamability of the individual lax triangles:

Definition 6.15 Given a 2-indexed coinstitution $\mathcal{I}\colon Ind^* \longrightarrow \mathbf{CoIns}$, a lax triangle of index morphisms



is called (weakly) amalgamable, if $\mathcal{I}$ maps it to a (weakly) amalgamable lax triangle in the sense of definition 2.36.

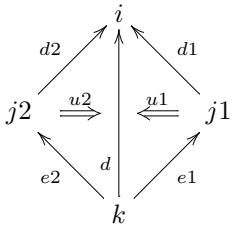
Theorem 6.16 For a 2-indexed coinstitution $\mathcal{I}\colon Ind^* \longrightarrow \mathbf{CoIns}$, assume that

- $\mathcal{I}$ is lax-quasi-exact, and
- all institution comorphisms in $\mathcal{I}$ are weakly exact.

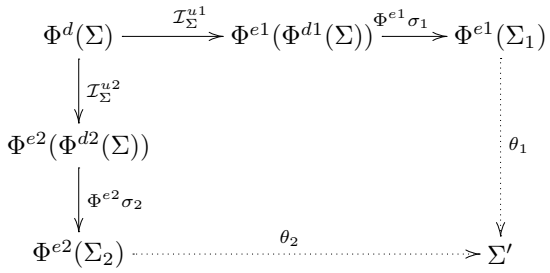
Then $\mathcal{I}^{\#}/\equiv$ is quasi-semi-exact.

Proof:

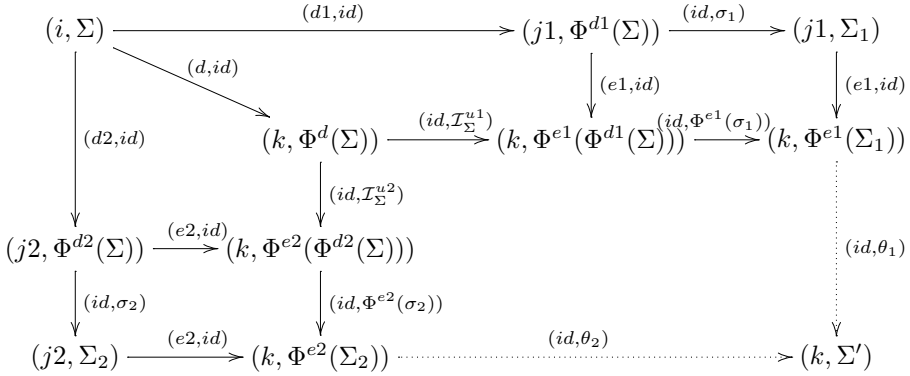
Let a diagram $(j_1, \Sigma_1) \xleftarrow{(d_1,\sigma_1)} (i, \Sigma) \xrightarrow{(d_2,\sigma_2)} (j_2, \Sigma_2)$ in $\mathbf{Sign}^{\#}$ be given. Let



be a weakly amalgamable square of two lax triangles with $\mathcal{I}^k$ quasi-semi-exact. By the latter property, there are θ_1, θ_2 such that



is a weakly amalgamable square, which leads to weak amalgamability of the lower right square in



The upper right and lower left squares are weakly amalgamable by weak exactness of $\mathcal{I}^{e_1}$ and $\mathcal{I}^{e_2}$. The pair of the remaining two squares is jointly weakly amalgamable since it is induced by a weakly amalgamable square of two lax triangles (and note that squares in $\mathbf{Sign}^\#/\equiv$ induced by lax triangles in Ind commute by definition of $\equiv$). Since weakly amalgamable squares can be pasted together, we get a weakly amalgamable cocone for the original diagram. □

Call a diagram *acyclic (connected)* if the graph underlying its index category is acyclic (connected) when the identity arrows are deleted.

Corollary 6.17 Let $\mathcal{I}$ satisfy the assumptions of Theorem 6.16. Then $\mathcal{I}^\#/\equiv$ admits weak amalgamation of finite acyclic connected diagrams.

PROOF: In the sequel, we will use terms like "connected", "maximal", "lower bound" for small categories, when we really mean the pre-order obtained from the category by collapsing the hom-sets into singletons. A maximal element in a pre-order is an element which is equivalent to any element above it.

Let $D\colon J \longrightarrow \mathbf{Sign}^\#$ be a connected diagram and let Max be the set of maximal nodes in J. We successively construct new diagrams out of J. Take two nodes in Max that have a common lower bound (if two such nodes do not exist, the diagram is not connected). By Theorem 6.16, there is a weak amalgamating cocone for the sub-diagram consisting of the two maximal nodes and the lower bound (together with the arrows from the lower bound into the maximal nodes). Extend the diagram with the cocone. The diagram thus obtained now has a set of maximal nodes whose size is decreased by one. By iterating this construction, we get a diagram with one maximal node. The maximal node then is just the tip of a weakly amalgamating cocone for the original diagram. □

6.3 Grothendieck Logics and Heterogeneous Borrowing

How to obtain an entailment system for the Grothendieck institution, i.e. turn it into a logic in the sense of Def. 2.9? The answer is easy if all the involved institutions are already logics, i.e. the indexed coinstinstitution is an *indexed cologic*. The latter notion is defined entirely analogous to that of an indexed coinstitution, as well as its Grothendieck constructions (just as the satisfaction relation, the entailment relation of a Grothendieck logic is constructed component wise).

Proposition 6.18 The Grothendieck logic $\mathcal{L}^\#$ of an indexed cologic $\mathcal{L}: Ind^{op} \longrightarrow \mathbf{coLog}$ is complete if and only if $\mathcal{L}$ is locally complete (i.e. each individual logic is complete).

PROOF: The logical rooms of the Grothendieck logic $\mathcal{L}^\#$ are just logical rooms of some individual logic in $\mathcal{L}$. Hence, global completeness is equivalent to local completeness. □

However, in many cases, not every institution will come with an entailment system (let alone a complete one); hence, it is difficult to apply the above proposition in practice.

We therefore follow a different path an assume that we have an indexed coinstitution where each institution is encoded (via a comorphism) in some expressive "universal" institution with good proof support, such that heterogeneous proof goals can be translated into homogeneous ones (over the "universal" institution), using a heterogeneous variant of the borrowing technique introduced in Sect. 5.2.

To this end, fix a logic $U = (\mathbf{USign}, \mathbf{USen}, \mathbf{UMod}, \models, \vdash)$ which we will very informally view as a "universal" logic (with sufficient expressiveness to represent many logics, and with suitable tool support). We will also denote the institution $(\mathbf{USign}, \mathbf{USen}, \mathbf{UMod}, \models)$ by U.

Definition 6.19 The *category of comorphisms into* U, denoted by $Comorphisms(U)$, has as objects comorphisms $\rho: I \longrightarrow U$ from some institution I into U, while morphisms (ρ, τ) from $\rho_1: I_1 \longrightarrow U$ to $\rho_2: I_2 \longrightarrow U$ consist of a comorphism $\rho: I_1 \longrightarrow I_2$ and a comorphism modification $\tau: \rho_1 \longrightarrow \rho_2 \circ \rho$. If $(\rho, \tau): \rho_1 \longrightarrow \rho_2$ and $(\rho', \tau'): \rho_2 \longrightarrow \rho_3$, the composition $(\rho', \tau') \circ (\rho, \tau)$ is given by $(\rho' \circ \rho, \tau\Phi^\rho \circ \Phi^{\rho'}\tau)$, where Φ^ρ is the signature translation component of ρ, and similarly for ρ'.

An *indexed comorphism* is just a functor $\mathcal{C}: Ind^{op} \longrightarrow Comorphisms(U)$.

Theorem 6.20 Given an indexed institution comorphism $\mathcal{C}: Ind^{op} \longrightarrow Comorphisms(U)$, we can form its *Grothendieck comorphism* $\mathcal{C}^\#: (\Pi_1 \circ \mathcal{C})^\# \longrightarrow U$, which is a comorphism going from the Grothendieck institution of the indexed coinstitution $\Pi_1 \circ \mathcal{C}$ formed from the source institutions and comorphisms involved in $\mathcal{C}$.

PROOF: Using the notation of institution as functors from Sect. 2.13, let $\mathcal{C}(i)$ be denoted by (Φ^i, ρ^i) and $\mathcal{C}(d: j \longrightarrow i)$ denoted by $((\Phi^d, \rho^d), \tau^d): (\Phi^i, \rho^i) \longrightarrow (\Phi^j, \rho^j)$. Note that the (Φ^d, ρ^d) are not used in the sequel, but they contribute to $\Pi_1 \circ \mathcal{C}$.

We need to construct a comorphism $\mathcal{C}^\# = (\Phi, \rho): (\Pi_1 \circ \mathcal{C})^\# \longrightarrow U$. On signatures, $\Phi(i, \Sigma) := \Phi^i(\Sigma)$, and on signature morphisms, $\Phi(\ (i, \Sigma_1) \xrightarrow{(d,\sigma)} (j, \Sigma_2)\ ) := \Phi^j(\sigma) \circ \tau^d_{\Sigma_1}$. On rooms, we just define $\rho_{(i,\Sigma)} := \rho^i_\Sigma$. □

Proposition 6.21 Given an indexed comorphism $\mathcal{C}: Ind^{op} \longrightarrow Comorphisms(U)$, if all the individual comorphisms $\mathcal{C}(i)$ admit model expansion, then also $\mathcal{C}^\#$ admits model expansion.

PROOF: Obvious by the component wise construction. □

Definition 6.22 Given an indexed comorphism $\mathcal{C}: Ind^{op} \longrightarrow Comorphisms(U)$ and a class of signature morphisms $\mathcal{D}$ in $(\Pi_1 \circ \mathcal{C})^\#$, $\mathcal{C}$ is said to *admit weak $\mathcal{D}$-amalgamation* if

- for each $i \in Ind$, $\mathcal{C}(i)$ admits weak $\mathcal{E}$-amalgamation, where $\mathcal{E} = \{\sigma \mid (d, \sigma) \in \mathcal{D}$ for some $d: i \longrightarrow j \in Ind\}$, and

- for each $d \in Ind$ such that $(d, \sigma) \in \mathcal{D}$ for some σ, $\mathcal{C}(d)$ admits weak amalgamation in the sense of Def. 2.36.

Proposition 6.23 Given an indexed comorphism $\mathcal{C}: Ind^{op} \longrightarrow Comorphisms(U)$ and a class of signature morphisms $\mathcal{D}$ in $(\Pi_1 \circ \mathcal{C})^{\#}$, if $\mathcal{C}$ admits weak $\mathcal{D}$-amalgamation, then $\mathcal{C}^{\#}$ also admits weak $\mathcal{D}$-amalgamation.

PROOF: Given $(d, \sigma) \in \mathcal{D}$, the relevant square (where u is the index for the "universal" institution U)

$$\begin{array}{ccccc}
\mathbf{Mod}^i(\Sigma_1) & \xleftarrow{\beta^d_{\Sigma_1}} & \mathbf{Mod}^j(\Phi^d(\Sigma_1)) & \xleftarrow{\mathbf{Mod}^i(\sigma)} & \mathbf{Mod}^j(\Sigma_2) \\
\uparrow{\scriptstyle \beta^i_{\Sigma_1}} & & \uparrow{\scriptstyle \beta^j_{\Phi^d(\Sigma_1)}} & & \uparrow{\scriptstyle \beta^j_{\Sigma_2}} \\
\mathbf{Mod}^u(\Phi^i(\Sigma_1)) & \xleftarrow{\mathbf{Mod}^u(\tau^d_{\Sigma_1})} & \mathbf{Mod}^u(\Phi^j(\Sigma_1)) & \xleftarrow{\mathbf{Mod}^u(\Phi^j(\sigma))} & \mathbf{Mod}^u(\Phi^j(\Sigma_2))
\end{array}$$

is obviously a composition of two weakly amalgamable squares, hence itself weakly amalgamable. □

Corollary 6.24 Given an indexed comorphism $\mathcal{C}: Ind^{op} \longrightarrow Comorphisms(U)$, if all the individual comorphisms $\mathcal{C}(i)$ admit model expansion, and moreover $\mathcal{C}$ admits weak $\mathcal{D}$-amalgamation, then $\mathcal{C}^{\#}$ admits global borrowing for development graphs containing hiding links only along signature morphisms in $\mathcal{D}$.

This means that global theorem links in heterogeneous development graphs can be derived using only the entailment relation of U (and the proof rules for development graphs from Sect. 5.6).

6.4 Heterogeneous Proofs

Often, heterogeneous borrowing is not feasible. One problem is a possible lack of weak amalgamation, as indicated in Sect. 6.2. Another problem is that while it is in principle often possible to encode every institution into some single "universal" logic as target for comorphisms, it may be much more desirable to use specific tools designed for specific (sub)logics, which may be far more efficient than a coding to some "universal" logic. This means that it would be desirable to have the possibility of heterogeneous proving, with proofs being constructed out of sub-proofs in different proof systems. A first attempt in this direction are *heterogeneous bridges* [BCL96, CBL99]. However, these have no clear semantical basis (a more detailed discussion is given in Sect. 6.6). We here aim at clear semantical basis for *heterogeneous specification and proofs*, where in the extreme for each proof goal the best-suited logic could be chosen individually, based on the proof calculus for development graphs given in Sect. 5.6. The central notion is that of heterogeneous development graphs:

Definition 6.25 A *heterogeneous development graph* is just a development graph over a Grothendieck institution.

In Sect. 6.3, we have introduced *heterogeneous borrowing* (through structured proving in U). Here, we use instead encodings of the individual institutions into (possibly varying) logics that are present in the indexed coinstitution. We call this *internalized* borrowing, which just means translation of a proof goal or refinement goal into another logic, *using heterogeneous development graphs*. Shifting of proof goals is sound if the involved comorphism admit model expansion. This allows us to choose the logic for carrying out proofs in a very flexible way — typically the most specific logic in which the theory and the goal can still be expressed. We therefore arrive at *truly heterogeneous proofs*, via structured proving in the Grothendieck institution. We will see, however, that due to the problems with amalgamation discussed in Sect. 6.2, some of the tools for development graphs need to be adapted to the heterogeneous case in order to work smoothly.

$$\frac{}{N \overset{cons}{\underset{(d,id)}{\Longrightarrow}} K}$$

if d is marked as model-expansive and K is isolated.

(Model-Expansion)

$$\frac{N \qquad K \overset{\theta'}{\underset{c}{\Longrightarrow}} N' \overset{}{\underset{\sigma'}{\dashrightarrow}} K'}{N \underset{\sigma}{\dashrightarrow} K \overset{\theta'}{\underset{c}{\Longrightarrow}} N' \qquad K'}$$

$\theta\colon \Sigma^N \longrightarrow \Sigma^{N'}$,
$\sigma' \circ \theta = \theta' \circ \sigma$,
$\theta(\Psi^N) \subseteq \Psi^{N'}$
K' is isolated

(Local Borrowing)

$$\frac{N \overset{\sigma}{\dashrightarrow} K \quad K \overset{\theta}{\Rightarrow} P}{N \overset{\theta\circ\sigma}{\dashrightarrow} P}$$

(Composition)

Figure 6.1: New proof rules for heterogeneous proofs

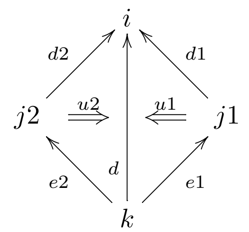
In particular, we add the three proof rules given in Fig. 6.1 to the proof calculus for development graphs given in Sect. 5.6. The rule **(Model-Expansion)** is a conservativity rule for comorphisms: if a comorphism is model-expansive, then we can treat the corresponding Grothendieck signature morphism as a semantically conservative morphism. The rules **(Local borrowing)** and **(Composition)** allow the simulation of "heterogeneous bridge" proofs.

Theorem 6.26 For a 2-indexed coinstitution $\mathcal{I}\colon Ind^* \longrightarrow$ **CoIns** (with some of the institutions also being logics), the proof calculus for heterogeneous development graphs given in Sect. 5.6, extended by the rules in Fig. 6.1 is sound for $\mathcal{I}^{\#}/\equiv$. If, moreover,

- $\mathcal{I}$ is lax-quasi-exact,
- all institution comorphisms in $\mathcal{I}$ are weakly exact,
- there is a set $\mathcal{L}$ of institutions in $\mathcal{I}$ that come as *complete* logics,
- the rule system is extended with a (sound and complete) oracle for conservative extension for each logic in $\mathcal{L}$,
- all institutions in $\mathcal{L}$ are quasi-semi-exact,
- from each institution in $\mathcal{I}$, there is some model-expansive comorphism in $\mathcal{I}$ going into some logic in $\mathcal{L}$,
- there is some set $\mathcal{D}$ of index morphisms in Ind such that for each diagram

  $$(j_1, \Sigma_1) \xleftarrow{(d_1,\sigma_1)} (i, \Sigma) \xrightarrow{(d_2,\sigma_2)} (j_2, \Sigma_2)$$

  in **Sign**$^{\#}$ with $d2 \in \mathcal{D}$, the corresponding weakly amalgamable square of lax triangles

  is such that the left lax triangle is weakly amalgamable and β^{e1} is model-expansive,

then the proof calculus complete for those heterogeneous development graphs over $\mathcal{I}^{\#}/\equiv$ that use hiding links are only with signature morphisms whose comorphism component is in $\mathcal{D}$.

PROOF: **Soundness:**

The soundness of most of the rules follows from the soundness proof in Sect. 5.6 (note that the rule **(Theorem-Hide-Shift)** has the needed weak amalgamabilities as side condition, hence we do not need quasi-exactness of the institutions). There are three new rules, which we now prove to be sound:

(Model-Expansion): By assumption, $\mathcal{I}^d$ is model-expansive. But ρ^d is just $\mathcal{I}^{\#}(d, id)$. Since N is isolated, any M-model can be (d, id)-expanded to an N-model.

(Local Borrowing): Let M be an K-model. Since by assumption $K = \stackrel{\theta'}{=\!\!\Rightarrow} K'$ is conservative and K' is isolated, M has a θ'-expansion to an K'-model M'. By the assumption $N' - \stackrel{\sigma'}{-\!\!\rightarrow} K'$, $M'|_{\theta'} \models \Psi^{N'}$, hence by the first side condition, $M'|_{\sigma'} \models \theta(\Psi^{N'})$. By the satisfaction condition, $M'|_{\sigma' \circ \theta} = M'|_{\theta' \circ \sigma} \models \Psi^{N'}$, i.e. $M|_{\sigma} \models \Psi^{N'}$.

(Composition): Let M be a P-model. By the second assumption, $M|_{\theta}$ is a K-model, and by the first assumption, $M|_{\theta \circ \sigma} \models \Psi^N$. □

Completeness:

We first need a preparatory lemma:

Lemma 6.27 If P is constructed as in the rule **(Theorem-Hide-Shift)**, then any Σ^P-model satisfying $Th_{\mathcal{DG}}(P)$ is already a P-model.

PROOF: We use the notation introduced in connection with the construction of P in the rule **(Theorem-Hide-Shift)**. For $i \in |I|$, let $len(i)$ be the length of the path i, and let p be the maximum of all $len(i)$, $i \in |I|$. We prove by induction over $p - len(i)$ that for all P-models M and all paths $i \in |I|$ containing no local definition link, $M|_{\mu_i}$ is a $G(i)$-model. Since $G(\langle N \rangle) = N$, the result then follows. Let $i = \langle\, K \xrightarrow{l_1} \cdots \xrightarrow{l_n} N \,\rangle \in |I|$ be a path containing no local definition link. By induction hypothesis, for each ingoing non-local link $O \stackrel{l}{\Longrightarrow} K$, $M|_{\mu_j}$ is an O-model for $j = \langle\, O \xrightarrow{l} K \xrightarrow{l_1} \cdots \xrightarrow{l_n} N \,\rangle$. Now

- if $l = O \xrightarrow{\sigma} K$, $M|_{\mu_i \circ \sigma} = M|_{\mu_j}$, and since $O = G(j) \xrightarrow{\mu_j} P \in S$, $M|_{\mu_j} \models \Sigma^O$;
- if $l = O \stackrel{\sigma}{\Longrightarrow} K$, $M|_{\mu_i \circ \sigma} = M|_{\mu_j}$, and $M|_{\mu_i}$ σ-reduces to an O-model;
- if $l = O \stackrel{\sigma}{\underset{hide}{\Longrightarrow}} K$, $M|_{\mu_j \circ \sigma} = M|_{\mu_i}$, and $M|_{\mu_i}$ σ-expands to an O-model.

Hence in all cases, the link l is satisfied by $M|_{\mu_i}$. Since $K = G(i) \xrightarrow{\mu_i} P$, $M|_{\mu_i}$ also satisfies the local axioms in K. Hence, $M|_{\mu_i}$ is a model of $K = G(i)$. □

We now come to the proof of the completeness theorem.

Assume $\mathcal{DG} \models K = \stackrel{\sigma}{=\!\!\Rightarrow} N$. We want to show $\mathcal{DG} \vdash K = \stackrel{\sigma}{=\!\!\Rightarrow} N$.

Let $D\colon I \longrightarrow \mathbf{Sign}$ and P be as in the rule **(Theorem-Hide-Shift)**, noting that by Corollary 6.17, a weakly amalgamable cocone exists, since the diagram constructed in the rule is acyclic and connected. Let (i^P, Σ^P) be the signature of P, and let M be an (i^P, Σ^P)-model satisfying $Th_{\mathcal{DG}}(P)$. By Lemma 6.27, M is a P-model. Hence, $M|_{\mu_{\langle N \rangle}}$ is an N-model, and by the assumption $\mathcal{DG} \models K = \stackrel{\sigma}{=\!\!\Rightarrow} N$, $M|_{\mu_{\langle N \rangle} \circ \sigma}$ is a K-model. We now have for any $O \stackrel{\theta}{\Longrightarrow\!\!\!\!\Rightarrow} K$:

1. $M|_{\mu_{\langle N\rangle}\circ\sigma\circ\theta} \models \Psi^O$ by Prop. 5.13. By the satisfaction condition for the Grothendieck institution $\mathcal{I}^{\#}/\equiv$, we obtain $M \models \mu_{\langle N\rangle}(\sigma(\theta(\Psi^O)))$. Since M was an arbitrary $Th_{\mathcal{DG}}(P)$-model, we have shown $Th_{\mathcal{DG}}(P) \models \mu_{\langle N\rangle}(\sigma(\theta(\Psi^O)))$. Let $d{:}\, l \longrightarrow i \in Ind$ such that $\mathcal{I}^d$ is a model-expansive comorphism from $\mathcal{I}^i$ to $\mathcal{I}^l$, where the latter also is a complete logic (this exists by the sixth assumption of the theorem). Obtain a new development graph $\mathcal{DG}'$ from $\mathcal{DG}$ by letting P' be a new node with signature $(l, \Phi^d(\Sigma^P))$ and with one ingoing definition link $P \overset{(d,id)}{\Longrightarrow} P'$.

$$\begin{array}{ccc} O & \overset{\mu_{\langle N\rangle}\circ\sigma\circ\theta}{= = = = =\Rightarrow} & P \\ \Big\Vert & & \Big\Vert \\ \Big\Vert id & & \Big\Vert (d,id) \\ \Big\Downarrow & & \Big\Downarrow \\ O & \overset{(d,id)\circ\mu_{\langle N\rangle}\circ\sigma\circ\theta}{= = = = =\Rightarrow} & P' \end{array}$$

By the satisfaction condition for comorphism corridors, we get $Th_{\mathcal{DG}}(P') \models \alpha^d(\mu_{\langle N\rangle}(\sigma(\theta(\Psi^O))))$. By completeness of the logic $\mathcal{I}^l$, we obtain $Th_{\mathcal{DG}}(P') \vdash \alpha^d(\mu_{\langle N\rangle}(\sigma(\theta(\Psi^O))))$. By **(Basic Inference)**, $\mathcal{DG} \vdash P' \Rightarrow \alpha^d(\mu_{\langle N\rangle}(\sigma(\theta(\Psi^O))))$. By **(Local Inference)**, $\mathcal{DG} \vdash O \overset{(d,id)\circ\mu_{\langle N\rangle}\circ\sigma\circ\theta}{= = = = =\Rightarrow} P'$. Since by assumption, all comorphisms are model-expansive, by **(Model-Expansion)**, (d, id) is conservative. By **(Subsumption)** and **(Borrowing)**, $\mathcal{DG} \vdash O \overset{\mu_{\langle N\rangle}\circ\sigma\circ\theta}{= = = = =\Rightarrow} P$.

2. For $Q \overset{(d_2,\sigma_2)}{\underset{hide}{\Longrightarrow}} O$, by Theorem 6.16, we obtain a weakly amalgamating cocone

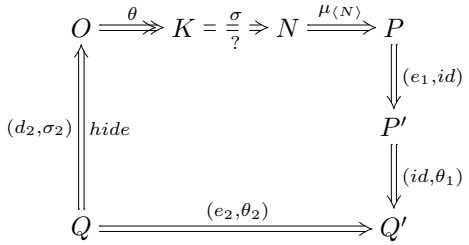
$$\begin{array}{ccc} \Sigma^O & \xrightarrow{\mu_{\langle N\rangle}\circ\sigma\circ\theta} & \Sigma^P \\ {\scriptstyle (d_2,\sigma_2)}\downarrow & & \downarrow{\scriptstyle (e_1,\theta_1)} \\ \Sigma^Q & \xrightarrow{(e_2,\theta_2)} & (k, \Sigma') \end{array}$$

By inspecting the proof of Theorem 6.16, we can assume that the above diagram is split up in the following way:

$$\begin{array}{ccccccc} (i,\Sigma) & & \xrightarrow{\quad(d1,id)\quad} & & (j1, \Phi^{d1}(\Sigma)) & \xrightarrow{(id,\sigma_1)} & (j1, \Sigma_1) \\ & \searrow^{(d,id)} & & & \downarrow{\scriptstyle (e1,id)} & & \downarrow{\scriptstyle (e1,id)} \\ \downarrow{\scriptstyle (d2,id)} & & (k, \Phi^d(\Sigma)) & \xrightarrow{(id,\mathcal{I}^{u1}_{\Sigma})} & (k, \Phi^{e1}(\Phi^{d1}(\Sigma))) & \xrightarrow{(id,\Phi^{e1}(\sigma_1))} & (k, \Phi^{e1}(\Sigma_1)) \\ & & \downarrow{\scriptstyle (id,\mathcal{I}^{u2}_{\Sigma})} & & & & \\ (j2, \Phi^{d2}(\Sigma)) & \xrightarrow{(e2,id)} & (k, \Phi^{e2}(\Phi^{d2}(\Sigma))) & & & & \vdots\ {\scriptstyle (id,\theta_1)} \\ \downarrow{\scriptstyle (id,\sigma_2)} & & \downarrow{\scriptstyle (id,\Phi^{e2}(\sigma_2))} & & & & \\ (j2, \Sigma_2) & \xrightarrow{(e2,id)} & (k, \Phi^{e2}(\Sigma_2)) & \cdots\cdots\cdots & {\scriptstyle (id,\theta_2)} \cdots\cdots\cdots & \cdots\cdots\cdots & \succ (k, \Sigma') \end{array}$$

With loss of generality, we can assume that $\mathcal{I}^k$ is in the set $\mathcal{L}$ of complete logics (if not, use the sixth assumption of the theorem to end up in such a logic, using model-expansiveness to keep the weak amalgamation property). We will interchangeably also use the notation of the diagram in the proof of Theorem 6.16, i.e. we put $(d_1, \sigma_1) := \mu_{\langle N\rangle} \circ \sigma \circ \theta$ and $(j_1, \Sigma_1) := (i^P, \Sigma^P)$.

We now construct a new development graph $\mathcal{DG}'$ from $\mathcal{DG}$ by introducing a new node Q' with signature (k, Σ'), a new node P' with signature $(k, \Phi^{e_1}(\Sigma_1))$. P' has one ingoing definition link $P \xrightarrow{(e_1,id)} P'$, while Q' has two ingoing definition links $Q \xrightarrow{(e_2,\theta_2)} Q'$ and $P' \xrightarrow{(id,\theta_1)} Q'$.



We now show the latter link to be conservative: Since by assumption, hiding is done only against signature morphisms with comorphism component in $\mathcal{D}$, $\mathcal{I}^{e_1}$ is model-expansive, and by **(Model-Expansion)**, $P \xrightarrow[cons]{(e_1,id)} P'$. For any P'-model M', $M'|_{(e_1,id)\circ(d_1,\sigma_1)}$ has a (d_2, σ_2)-expansion to a Q-model M_2. Since $d2 \in \mathcal{D}$, the upper left square in the diagram from Theorem 6.16 is weakly amalgamable, and hence $M'|_{(id,\Phi^{e1}(\sigma_1))\circ(id,\mathcal{I}^{u1}_{\Sigma_1})}$ and $M_2|_{(id,\sigma_2)}$ can be amalgamated to a $(k, \Phi^{e2}(\Phi^{d2}(\Sigma)))$-model. The latter can be amalgamated with M_2 to a $(k, \Phi^{e2}(\Sigma_2))$-model, which in turn can be amalgamated with M' to a (k, Σ')-model that by construction is a Q'-model. By the oracle for conservativity in logic $\mathcal{I}^k$ (note that by assumption all logics in $\mathcal{L}$ come with such an oracle), we get $P' \xrightarrow[cons]{(id,\theta_1)} Q'$. With **(Cons-Composition)**, we get $P \xrightarrow[cons]{(e_1,\theta_1)} Q'$. Now $\mathcal{DG}' \vdash Q \overset{(e_2,\theta_2)}{=\!=\!\Rightarrow} Q'$ by **(Subsumption)**. By **(Hide-Theorem-Shift)**, we get $\mathcal{DG}' \vdash Q \underset{hide}{\overset{\mu\langle N\rangle\circ\sigma\circ\theta}{=\!=\!=\!=\!\Rightarrow}}_{(d2,\sigma_2)} P$.

Let $\mathcal{DG}_1$ be the union of all the $\mathcal{DG}'$ constructed in steps 1 and 2 above (assuming that all the added nodes are distinct). By **(Glob-Decomposition)**, we get $\mathcal{DG}_1 \vdash K \overset{\mu\langle N\rangle\circ\sigma}{=\!=\!\Rightarrow} P$. By **(Theorem-Hide-Shift)**, we get $\mathcal{DG}_1 \vdash K \overset{\sigma}{=\!=\!\Rightarrow} N$. Finally, noting that all graph extensions in the proofs are faithful, by **(Faithful extension)**, we obtain $\mathcal{DG} \vdash K \overset{\sigma}{=\!=\!\Rightarrow} N$. □

Note that due to Gödel's incompleteness theorem, one cannot expect to drop the oracle for conservative extensions, see Prop. 5.22. The crucial achievement here is to restrict the oracle to *intra-logic* conservativity.

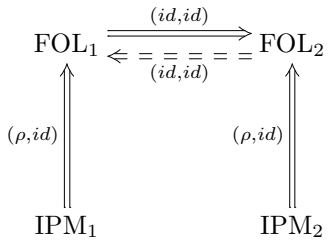
Further note that in contrast to Theorem 6.12, we need *neither* cocontinuity *nor* exactness of the comorphism signature translations here. Moreover, we need quasi-exactness only for some of the logics; this allows us to include logics which are not quasi-exact, such as CASL.

6.5 A Sample Heterogeneous Proof

Consider the sample heterogeneous specification in Fig. 6.2. It starts with a theory in IndexedProp-Modal (see Sect. 3.2.5), specifying something about persons that may be married, dead and immortal. (Intuitively, the modalities are interpreted as temporal ones here.) This theory is then translated to FOL along the comorphism making worlds explicit defined in Sect. refsec:ModalComorphisms. Then, the resulting theory is enriched by some first-order formula involving existential quantification. Call the thus obtained theory T. Finally, using the **then %implies** annotation, a proof obligation φ is expressed. The proof obligation is again written in IndexedPropModal, but it is implicitly coerced to FOL using the same comorphism as above in order to be compatible with T. The proof obligation expresses that T implies φ.

spec MARRIAGE =
 { **logic** IndexedPropModal
  sort *Person*
  props *isMarried*, *immortal*, *dead* : *Person*;
   isMarriedTo : *Person* × *Person*
  var x, y : *Person*
  • $isMarriedTo(x, y) \Rightarrow \neg dead(x)$
  • $immortal(x) \Leftrightarrow \neg\Diamond dead(x)$ **}** **with logic** $\longrightarrow$ FOL
then *var* x : *Person*; w : *World*
  • $isMarried(x, w) \Rightarrow \exists y : Person \bullet isMarriedTo(x, y, w)$
then **%implies**
  logic IndexedPropModal
  var x : *Person*
  • $(\Box isMarried(x)) \Rightarrow immortal(x)$
  } implicitly coerced to FOL
end

Figure 6.2: A sample heterogeneous specification



Local axioms:

IPM_1	$isMarriedTo(x, y) \Rightarrow \neg dead(x)$ $immortal(x) \Leftrightarrow \neg\Diamond dead(x)$
FOL_1	$isMarried(x, w) \Rightarrow \exists y : Person \bullet isMarriedTo(x, y, w)$
FOL_2	-
IPM_2	$\Box isMarried(x)) \Rightarrow immortal(x)$

Figure 6.3: The heterogeneous development graph for the specification MARRIAGE. ρ is the comorphism from IndexedPropModal to FOL.

As an example, the heterogeneous development graph for the specification MARRIAGE is shown in Fig. 6.3.

Suppose that we now want to prove the abovementioned proof obligation. There are several ways to do this. The simplest way is just to use heterogeneous borrowing, as descirbed in Sect. 6.3. This means that everything is translated to FOL (this is possible, because in our institution graph, each institution is embedded in FOL). Then, one can use the entailment system (resp. a corresponding theorem prover) for FOL.

However, this is unsatisfactory, because in practice it is more efficient to use the entailment system (resp. a corresponding theorem prover) for PropModal for those parts of the proof that can be carried out within PropModal. Therefore, we construct a truly heterogeneous proof (see also Sect. 6.4).

The global theorem link in the heterogeneous development graph is discharged by successive backwards applications of the proof rules, thereby reducing the theorem link to simpler ones, until all of them can be removed.

The entire proof is shown in Fig. 6.4. The first step just decomposes the global theorem link into several local ones. One of the latter can trivially be discharged in step 2, a second one in step

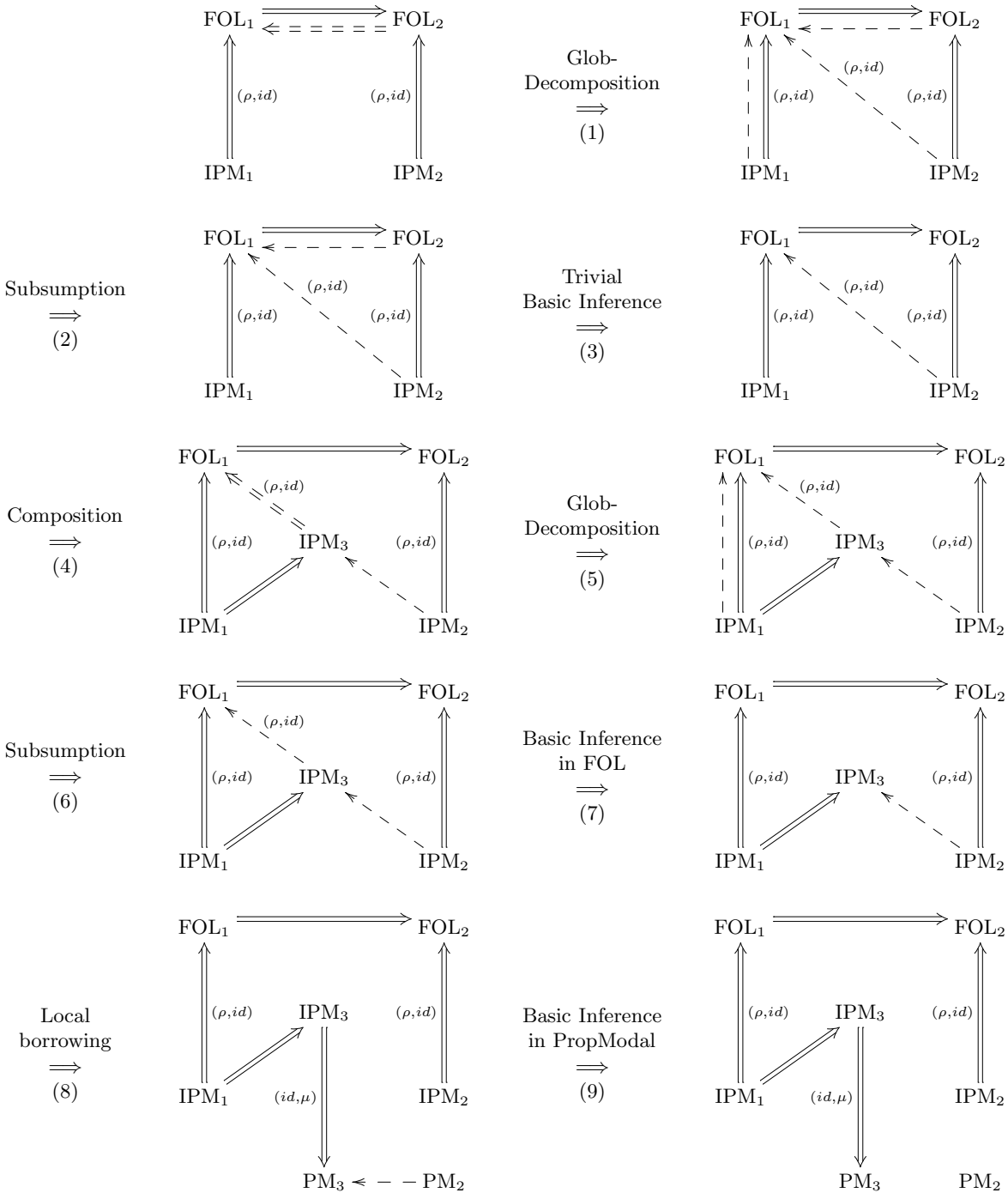


Figure 6.4: A sample heterogeneous proof

3. (The application of basic inference in the third step is trivial because the node FOL_2 does not contain local axioms.) At this point, we now could apply basic inference also for the remaining local theorem link. However, this would be basic inference entirely in FOL.

Instead, we try to decompose the proof goal into subgoals in FOL and in modal logic. This is done in step 4, using the rule **(Composition)**. This step is the key step of the whole proof, because here a "heterogeneous bridge" (in a sense that will be made precise in Sect. 6.6) is constructed. We therefore introduce a new node, IPM_3. It has an ingoing global definition link from IPM_1 and a local axiom $isMarried(x) \Rightarrow \neg dead(x)$. This local axiom can be seen as a lemma bridging between IndexedPropModal and FOL: it is formulated in IndexedPropModal, proved in FOL, and used in an IndexedPropModal proof. The fourth step thus leaves us with two new theorem links.

The first of these is a global one, and hence we decompose it in the step 5. One of the resulting local theorem links can be discharged trivially in step 6. The other one is solved by basic inference in FOL in step 7. This is done as follows: we need to show that the local axiom of IPM_3, namely $isMarried(x) \Rightarrow \neg dead(x)$, follows from FOL_1. Now the translation along ρ is $isMarried(x, w) \Rightarrow \neg dead(x, w)$, and this follows from $isMarried(x, w) \Rightarrow \exists y \bullet isMarriedTo(x, y, w)$ (a local axiom of FOL_1) and $isMarriedTo(x, y, w) \Rightarrow \neg dead(x, w)$ (the translation of $isMarriedTo(x, y) \Rightarrow \neg dead(x)$ coming from IPM_1) by inference in the FOL entailment system.

It remains to solve the remaining local theorem link, which lives entirely in IndexedPropModal. We now can exploit the way how IndexedPropModal was built; namely in such a way that there is a conservative comorphism μ into PropModal. We add two new nodes to the development graph, PM_2 and PM_3, whose signatures are the translations of those of IPM_2 and IPM_3, resp. PM_3 gets a global definition link coming from IPM_3 (via μ), while PM_2 only gets the translation of the local axiom in IPM_2 along μ as local axiom (the result of the translation is $\Box isMarried \Rightarrow immortal$, a purely modal propositional formula). We now can apply **(Local Borrowing)** in order to translate the local theorem link from IndexedPropModal into PropModal in step 8. Finally, in step 9, this link is discharged by basic inference in PropModal: from $\Box isMarried$, we get $\Box \neg dead$ by the bridge lemma (i.e. the local axiom of IPM_3). By propositional modal reasoning, we get $\neg \Diamond dead$. By the translation of one of the local IPM_1 axioms ($immortal \Leftrightarrow \neg \Diamond dead$), we get $immortal$. This completes the heterogeneous proof.

6.6 Heterogeneous Bridges

We now recall the original approach to heterogeneous bridges in [BCL96], but adapt it at a few points to integrate it more smoothly in our setting. One point of difference is that in [BCL96], signatures and models are translated covariantly (as by institution morphisms), while we translate them contravariantly (using institution comorphisms). See Sect. 6.10 for an explanation of why this makes no essential difference. We therefore recast the definition of [BCL96] for the comorphism case, make the indexing more obvious, and use heterogeneous development graphs instead of the heterogeneous specifications of [BCL96]. We also turn some of the technical conditions into (in our eyes) more natural ones, but the general idea remains that of [BCL96].

For the purpose of this section, we restrict ourselves to development graphs with flattenable nodes only (this assumption is implicit in [BCL96] as well).

Definition 6.28 A *heterogeneous framework* in the sense of [BCL96] is a tuple $(\mathcal{I}, \vdash)$ consisting of

- an indexed coinstitution $\mathcal{I}\colon Ind \longrightarrow \mathbf{CoIns}$,
- a family $(\vdash_{d,\Sigma})_{d:i \longrightarrow j \in Ind, \Sigma \in \mathbf{Sign}^i}$ with $\vdash_{d,\Sigma} \subset \mathcal{P}(\mathbf{Sen}^i(\Sigma)) \times \mathbf{Sen}^j(\Phi^d(\Sigma))$, called *heterogeneous inference bridge*,

such that

- $\Psi \vdash_{d,\Sigma} \varphi$ and $\Psi \subseteq \Psi'$ implies $\Psi \vdash_{d,\Sigma} \varphi'$ (monotonicity),

- for any $d\colon i \longrightarrow j \in Ind$, any signature morphism $\sigma\colon \Sigma \longrightarrow \Sigma'$ in $\mathbf{Sign}^i$, any $\Psi \subseteq \mathbf{Sen}^i(\Sigma)$, and any $\varphi \in \mathbf{Sen}^j(\Phi^d(\Sigma))$,

$$\Psi \Vdash_{d,\Sigma} \varphi \text{ implies } \sigma(\Psi) \Vdash_{d,\Sigma'} \Phi^d(\sigma)(\varphi) \quad (\Vdash\text{-translation}),$$

- for any $d\colon i \longrightarrow j \in Ind$, any $\Psi \subseteq \mathbf{Sen}^i(\Sigma)$, any $\varphi \in \mathbf{Sen}^j(\Phi^d(\Sigma))$, and any $M \in \mathbf{Mod}^j(\Phi(\Sigma))$ with $\Psi \Vdash_{d,\Sigma} \varphi$,

$$\text{if } \beta^d_\Sigma(M) \models^i_\Sigma \Psi, \text{ then } M \models^j_{\Phi(\Sigma)} \varphi \quad \text{(heterogeneous soundness)}.$$

Definition 6.29 Given a heterogeneous framework $(\mathcal{I}, \Vdash)$, the corresponding *heterogeneous inference system* is the least family (indexed by heterogeneous development graphs $\mathcal{DG}$ over $\mathcal{I}$) of binary relations $\Vdash\!\!\vdash_{\mathcal{DG}}$ between nodes N in $\mathcal{DG}$ and Σ^N-sentences such that

- $\Vdash\!\!\vdash_{\mathcal{DG}}$ is compatible with $\vdash$: if $\Sigma^N = (i, \Sigma)$ and $\Psi^N \vdash^i_\Sigma \varphi$, then $N \Vdash\!\!\vdash_{\mathcal{DG}} \varphi$;
- $\Vdash\!\!\vdash_{\mathcal{DG}}$ is compatible with $\Vdash$: if $K \overset{(d:i\longrightarrow j,\sigma)}{\Longrightarrow} N \in \mathcal{DG}$, $K \Vdash\!\!\vdash_{\mathcal{DG}} \Psi$ and $\Psi \Vdash_{d,\Sigma} \varphi$, then $N \Vdash\!\!\vdash_{\mathcal{DG}} \sigma(\varphi)$;
- $\Vdash\!\!\vdash_{\mathcal{DG}}$ is transitive: if $N \Vdash\!\!\vdash_{\mathcal{DG}} \Psi$, $\mathcal{DG}'$ results from $\mathcal{DG}$ by adding $N \overset{id}{\Longrightarrow} \overset{\Psi}{\bullet}$ (where $\overset{\Psi}{\bullet}$ is a new node with local axioms Ψ), and $\overset{\Psi}{\bullet} \Vdash\!\!\vdash_{\mathcal{DG}'} \varphi$, then $N \Vdash\!\!\vdash_{\mathcal{DG}} \varphi$.

Heterogeneous bridges have been the first proof-theoretic device for heterogeneous specification, and have provided a foundation for so-called brigde lemmas (see also Sect. 6.5). Our main criticism of this approach is the ad-hoc nature of the bridge relation $\Vdash$. We find it more natural to generate it via the entailment relation $\vdash$ of a logic in connection with the sentence translation of a comorphism:

Proposition 6.30 Any indexed coinstitution $\mathcal{I}$ equipped with a set $\mathcal{L}$ of logics satisfying the assumptions for the completeness result in Theorem 6.26 leads to a heterogeneous framework by just putting

$$\Psi \Vdash_{d:i\longrightarrow j,\Sigma} \varphi \quad \text{iff} \quad \alpha^{d'}(\alpha^d(\Psi)) \vdash^k_{\Phi^{d'}(\Phi^d(\Sigma))} \alpha^{d'}(\varphi)$$

where $d'\colon j \longrightarrow k$ is such that $\mathcal{I}(d')$ is a model-expansive comorphism into some logic in $\mathcal{L}$.

Proof: Monotonicity follows from monotonicity of the entailment relations, $\Vdash$-translation follows from naturality of α^d and $\alpha^{d'}$ and $\vdash$-translation for the entailment relations, and heterogeneous soundness follows from soundness of $\vdash$ and the satisfaction condition for $\mathcal{I}^d$ and $\mathcal{I}^{d'}$. □

Theorem 6.31 When using the construction of Prop. 6.30, the heterogeneous inference system $\Vdash\!\!\vdash$ induced by the constructed heterogeneous framework can be simulated using the proof rules for heterogeneous development (see Sect. 6.4). That is,

$$N \Vdash\!\!\vdash_{\mathcal{DG}} \varphi$$

implies

$$\mathcal{DG} \vdash N \Rightarrow \varphi$$

in the calculus for heterogeneous development graphs. □

Proof: By induction over the construction of the relation $\Vdash\!\!\vdash_{\mathcal{DG}}$.

The first clause in Def. 6.29 can be easily handeled using **(Basic Inference)**.

Concerning the second clause, by induction hypothesis, $\mathcal{DG} \vdash N \Rightarrow \Psi$, and the assumption $\Psi \Vdash_{d,\Sigma} \varphi$ means that for some $d' \colon j \longrightarrow k$, $\alpha^{d'}(\alpha^d(\Psi)) \vdash^k_{\Phi^{d'}(\Phi^d(\Sigma))} \alpha^{d'}(\varphi)$. Extend $\mathcal{DG}$ to $\mathcal{DG}'$as follows (for simplicity, we have decorated the new nodes with their local axioms, if any):

$$\begin{array}{ccc}
 & & \overset{\Psi}{\bullet} - \overset{id}{-} \succ K \\
 & & \Big\Downarrow (d,id) \qquad \Big\Downarrow (d,\sigma) \\
\overset{\varphi}{\bullet} - - \overset{id}{-} \succ & P & \overset{(id,\sigma)}{= = \Rightarrow} N \\
 & \Big\Downarrow (d',id) & \\
\overset{\alpha^{d'}(\varphi)}{\bullet} - \overset{id}{-} \succ & Q &
\end{array}$$

By **(Faithful Extension)**, $\mathcal{DG}' \vdash N \Rightarrow \Psi$, and by **(Local Inference)**, $\mathcal{DG}' \vdash \overset{\Psi}{\bullet} - \overset{id}{-} \succ K$. By **(Subsumption)** and **(Composition)**, $\mathcal{DG}' \vdash \overset{\Psi}{\bullet} - \overset{(d,\sigma)}{-} \succ N$. By **(Global Decomposition)**, $\mathcal{DG}' \vdash P \overset{(id,\sigma)}{= = \Rightarrow} N$. From $\alpha^{d'}(\alpha^d(\Psi)) \vdash^k_{\Phi^{d'}(\Phi^d(\Sigma))} \alpha^{d'}(\varphi)$ by **(Basic Inference)** and **(Local Inference)**, $\mathcal{DG}' \vdash \overset{\alpha^{d'}(\varphi)}{\bullet} - \overset{id}{-} \succ Q$. By **(Local Borrowing)**, $\mathcal{DG}' \vdash \overset{\varphi}{\bullet} - \overset{id}{-} \succ P$. By **(Subsumption)** and **(Composition)**, $\mathcal{DG}' \vdash \overset{\varphi}{\bullet} - \overset{(id,\sigma)}{-} \succ N$. By **(Reverse Local Inference)**, $\mathcal{DG}' \vdash N \Rightarrow \sigma(\varphi)$. By **(Faithful Extension)**, $\mathcal{DG} \vdash N \Rightarrow \sigma(\varphi)$.

Concerning the third clause in Def. 6.29, by induction hypothesis, $\mathcal{DG} \vdash N \Rightarrow \Psi$ and $\mathcal{DG}' \vdash \overset{\Psi}{\bullet} \Rightarrow \varphi$.

$$\overset{\varphi}{\bullet} - \overset{id}{-} \succ \overset{\Psi}{\bullet} \underset{id}{\overset{id}{= = \Rightarrow}} N$$

By **(Faithful Extension)** and **(Local Inference)**, $\mathcal{DG}' \vdash \overset{\Psi}{\bullet} - \overset{id}{-} \succ N$ and $\mathcal{DG}' \vdash \overset{\varphi}{\bullet} - \overset{id}{-} \succ \overset{\Psi}{\bullet}$. From the former, by **(Global Decomposition)** and **(Subsumption)**, $\mathcal{DG}' \vdash \overset{\Psi}{\bullet} \overset{id}{= = \Rightarrow} N$. By **(Composition)**, $\mathcal{DG}' \vdash \overset{\varphi}{\bullet} - \overset{id}{-} \succ N$. By **(Faithful Extension)** and **(Reverse Local Inference)**, $\mathcal{DG} \vdash N \Rightarrow \varphi$. □

From soundness of the development graph calculus, we immediately get:

Corollary 6.32 The heterogeneous inference system $\Vdash_{\mathcal{DG}}$ is sound, i.e. if $N \Vdash_{\mathcal{DG}} \varphi$, then $M \models \varphi$ for any $M \in \mathbf{Mod}_{\mathcal{DG}}(N)$. □

Heterogeneous frameworks as introduced in [BCL96] are more general than the ones introduced here, because in [BCL96], only semi-comorphisms (i.e. comorphisms without sentence translation maps) are used. [6] However, we believe that this extra generality is not of much use (any practical bridge will be obtained as in Prop. 6.30), and moreover comes with the cost of introducing the bridge relation $\Vdash$ in an ad-hoc manner. Note that it is possible (and important) also to include semi-(co)morphisms in our framework (see Sect. 6.10), but this is mainly for relating specification and programming languages, and not for heterogeneous bridges.

[6] Actually, it is easy to introduce indexed semi-coinstitutions, but the corresponding Grothendieck construction does not yield an institution: only homogeneous signature morphisms induce sentence translations. In principle, it is possible to introduce kind of "partial institution" for this. However, for the sake of simplicity, we have refrained from doing so. Note that the sentence translation maps of the comorphisms are not needed before Prop. 6.30. A proper treatment of semi-comorphisms is given in Sect. 6.10.

6.7 Morphism-Based Grothendieck Institutions

Diaconescu orignially developed the theory of Grothendieck institutions for indexed institutions, based on institution morphisms rather than comorphisms, as we have done. The relevant notions are easily dualized:

Given an index 2-category Ind, a *2-indexed institution* is a 2-functor $\mathcal{I}\colon Ind^* \longrightarrow \mathbf{Ins}$[7] into the 2-category of institutions, institution morphisms and institution morphism modifications. In cases where the 2-categorical structure is not needed, we omit the prefix "2-".

Definition 6.33 Given an indexed institution $\mathcal{I}\colon Ind^* \longrightarrow \mathbf{Ins}$, define the *Grothendieck institution* $\mathcal{I}^\#$ as follows (where we use the notation of institutions as functors and moreover write (Ψ^d, μ^d) for $\mathcal{I}(d)$):

- signatures in $\mathcal{I}^\#$ are pairs (i, Σ), where $i \in |Ind|$ and Σ a signature in $\mathcal{I}^i$,
- signature morphisms $(d, \sigma)\colon (i, \Sigma_1) \longrightarrow (j, \Sigma_2)$ consist of a morphism $d\colon i \longrightarrow j \in Ind$ and a signature morphism $\sigma\colon \Sigma_1 \longrightarrow \Psi^d(\Sigma_2)$ in $\mathcal{I}^j$,
- composition is given by $(d_2, \sigma_2) \circ (d_1, \sigma_1) = (d_2 \circ d_1, \Psi^{d_1}(\sigma_2) \circ \sigma_1)$,
- $\mathcal{I}^\#(i, \Sigma) = \mathcal{I}^i(\Sigma)$, and $\mathcal{I}^\#(d, \sigma) = \ \mathcal{I}^i(\Sigma_1) \xrightarrow{\mathcal{I}^i(\sigma)} \mathcal{I}^i(\Psi^d(\Sigma_1)) \xrightarrow{\mu^d} \mathcal{I}^j(\Sigma_2)$.

For 2-indexed institutions, we can also form a quotient Grothendieck institution, as in Sect. 6.1.

Diaconescu proves results about amalgamation properties of Grothendieck institutions for so-called *embedding-indexed institutions*, which means that each signature translation Ψ^d has a left adjoint Φ^d. But these left adjoints lead to a corresponding indexed coinstitution, and in fact, strictly speaking Diaconescu uses this induced indexed coinstitution in his proofs. This shows that indexed coinstitutions are simpler and more general than embedding-indexed institutions (and only for these, Diaconescu has results about exactness and amalgamability). In particular, a simpler proof of Diaconescu's results can be obtained by reducing them to our results in Sect. 6.2 via the following generalization of the adjointness between morphisms and comorphisms introduced in Sect. 2.9:

Proposition 6.34 Given an embedding-indexed institution $\mathcal{I}\colon Ind^{op} \longrightarrow \mathbf{Ins}$, define the indexed coinstitution $\mathcal{I}^{co}\colon Ind^{op} \longrightarrow \mathbf{CoIns}$ by

$$\mathcal{I}^{co}(i) := \mathcal{I}(i) \text{ and } \mathcal{I}^{co}(d\colon i \longrightarrow j) := (\Phi^d, (\mu^d \cdot \Phi^d) \circ (\mathcal{I}^j \cdot \eta^d)),$$

where η^d is the unit of the adjunction between Φ^d and Ψ^d.
Then $\mathcal{I}^\#$ is isomorphic to $(\mathcal{I}^{co})^\#$.

Proof: Let η^d be the unit and ε^d the counit of the adjunction between Ψ^d and Φ^d. Define an institution comorphism $(\Phi, \rho)\colon \mathcal{I}^\# \longrightarrow (\mathcal{I}^{co})^\#$ as follows: Φ sends (i, Σ) to (i, Σ) and $(d, \sigma)\colon (i, \Sigma_1) \longrightarrow (j, \Sigma_2)$ in $\mathcal{I}^\#$ to $(d, \varepsilon_{\Sigma_2} \circ \Phi^d(\sigma))\colon (i, \Sigma_1) \longrightarrow (j, \Sigma_2)$ in $(\mathcal{I}^{co})^\#$. Since $\varepsilon_{\Sigma_2} \circ \Phi^d(\sigma)$ is is just the arrow adjoint to σ, Φ is an isomorphism.

Now $\mathcal{I}^\#(d, \sigma)$ is $\mathcal{I}^i(\Sigma_1) \xrightarrow{\mathcal{I}^i(\sigma)} \mathcal{I}^i(\Psi^d(\Sigma_2)) \xrightarrow{\mu^d_{\Sigma_2}} \mathcal{I}^j(\Sigma_2)$. Since

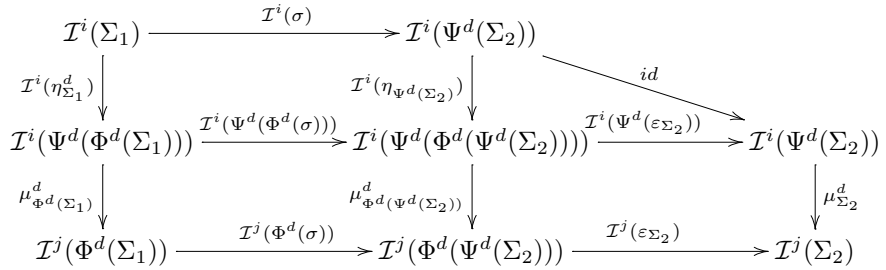
$$\mathcal{I}^{co}(d\colon j \longrightarrow i) = (\Phi^d, (\mu^d \cdot \Phi^d) \circ (I^i \cdot \eta^d)),$$

we get that $(\mathcal{I}^{co})^\#(\Phi(d, \sigma)) = (d, \mathcal{I}^{co})^\#(\varepsilon_{\Sigma_2} \circ \Phi^d(\sigma)) \ =$

$$\mathcal{I}^i(\Sigma_1) \xrightarrow{\mathcal{I}^i(\eta^d_{\Sigma_1})} \bullet \xrightarrow{\mu^d_{\Phi^d(\Sigma_1)}} \bullet \xrightarrow{\mathcal{I}^j(\Phi^d(\sigma))} \bullet \xrightarrow{\mathcal{I}^j(\varepsilon_{\Sigma_2})} \mathcal{I}^j(\Sigma_2) \ .$$

[7] Recall that Ind^* is the 2-categorical dual of Ind, where both 1-cells and 2-cells are reversed.

By the following diagram, both are the same, showing that ρ can be taken to be the identity:



The squares commute by naturality of η^d and μ^d, while the triangle commutes by a general adjointness law. □

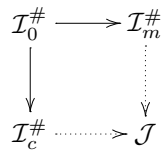
The Grothendieck construction does not obviously generalize to diagrams consisting of forward or semi-(co)morphisms, because of the lacking (contravariance between model and) sentence translation. We therefore concentrate for a moment on morphisms and comorphisms only, and postpone the treatment of forward and semi-(co)morphisms to Sect. 6.10.

6.8 The Bi-Grothendieck Institution

As seen in Chap. 4, both institution comorphisms and morphisms are needed for heterogeneous specification. The question is therefore how to obtain a kind of Grothendieck construction involving both comorphisms and morphisms. Consider heterogeneous specification over a set of institutions, a set of morphisms and a set of comorphisms. This can be formalized as an indexed institution $\mathcal{I}_m$ (collecting the morphisms) together with an indexed coinstitution $\mathcal{I}_c$ (collecting the comorphisms), both with the same underlying set of institutions, regarded as a discrete indexed institution $\mathcal{I}_0$.

Definition 6.35 Let $(\mathcal{I}_m, \mathcal{I}_c, \mathcal{I}_0)$, with $\mathcal{I}_m\colon Ind_m^{op} \longrightarrow \mathbf{Ins}$ an indexed institution, $\mathcal{I}_c\colon Ind_c^{op} \longrightarrow \mathbf{CoIns}$ an indexed coinstitution and $\mathcal{I}_0\colon Ind_0^{op} \longrightarrow \mathbf{Ins}$ a discrete indexed institution be given, such that $|Ind_m| = |Ind_c| = |Ind_0|$, and $\mathcal{I}_m$, $\mathcal{I}_c$ and $\mathcal{I}_0$ agree on these.

Then we form the Grothendieck institutions $\mathcal{I}_0^\#$, $\mathcal{I}_m^\#$ and $\mathcal{I}_c^\#$. Since $\mathcal{I}_0^\#$ obviously is included in $\mathcal{I}_m^\#$ and $\mathcal{I}_c^\#$ via a (co)morphism, we can take the pushout



in the category of institutions and institution morphisms (or comorphisms, this would make no difference here). The pushout in either category exists by results of [RG04]. By abuse of notation, we will denote the pushout $\mathcal{J}$ by $(\mathcal{I}_m, \mathcal{I}_c)^\#$. It will be called the *Bi-Grothendieck institution.*

Since at the level of signature categories, this is a pushout, we obtain as new signature morphisms paths consisting of both morphism-based and comorphism-based heterogeneous signature morphisms.

The heterogeneous development graph in Fig. 1.9 can now formally be understood as a development graph over the Bi-Grothendieck institution.

6.9 Inducibility

The Bi-Grothendieck institution is quite complex, and it is not immediately clear how to obtain e.g. proof support for it. It is therefore tempting to try to reduce the complexity of this construction by

mapping morphisms to comorphisms or vice versa. This can be done by weakening the adjunction between morphisms and comorphisms introduced in Sect. 2.9:

Definition 6.36 Given an institution comorphism $\rho = (\Phi, \alpha, \beta)\colon I \longrightarrow J$, a functor $\Psi\colon \mathbf{Sign}^J \longrightarrow \mathbf{Sign}^I$ and a natural transformation $\varepsilon\colon \Phi \circ \Psi \longrightarrow Id$, we say that ρ *ε-induces* the institution morphism $\mu = (\Psi, \bar{\alpha}, \bar{\beta})\colon J \longrightarrow I$ given by

$$\bar{\alpha} = (\mathbf{Sen}^J \cdot \varepsilon) \circ (\alpha \cdot \Psi)$$
$$\bar{\beta} = (\beta \cdot \Psi^{op}) \circ (\mathbf{Mod}^J \cdot \varepsilon^{op})$$

A morphism that is ε-induced by some comorphism is called *inducible*.

Dually, given an institution morphism $\mu = (\Psi, \bar{\alpha}, \bar{\beta})\colon J \longrightarrow I$, a functor $\Phi\colon \mathbf{Sign}^I \longrightarrow \mathbf{Sign}^J$ and a natural transformation $\eta\colon Id \longrightarrow \Psi \circ \Phi$, we say that μ *η-induces* the institution comorphism $\rho = (\Phi, \alpha, \beta)\colon I \longrightarrow J$ given by

$$\alpha = (\bar{\alpha} \cdot \Phi) \circ (\mathbf{Sen}^I \cdot \eta)$$
$$\beta = (\mathbf{Mod}^I \cdot \eta^{op}) \circ (\bar{\beta} \cdot \Phi^{op})$$

A comorphism that is η-induced by some morphism is called *inducible*. Moreover, it is straightforward to extend inducibility to the simple theoroidal case. Here, $\mathbf{Sign}^I \xrightarrow{\Phi} \mathbf{Sign}^J$ has to be replaced with $\mathbf{Sign} \xrightarrow{\Phi} \mathbf{Th}^J \xrightarrow{Sig} \mathbf{Sign}^J$, leading to the equations

$$\alpha = (\bar{\alpha} \cdot Sig \cdot \Phi) \circ (\mathbf{Sen}^I \cdot \eta)$$
$$\beta = (\mathbf{Mod}^I \cdot \eta^{op}) \circ (\bar{\beta} \cdot Sig \cdot \Phi^{op})$$

Furthermore, inducibility also extend to semi-(co)morphisms. □

With this, we can easily obtain the desired reduction:

Theorem 6.37 Let $(\mathcal{I}_m, \mathcal{I}_c, \mathcal{I}_0)$ as in Definition 6.35 be given.

If each morphism in $\mathcal{I}_m$ is ε-induced by some comorphism in $\mathcal{I}_c$, then there is a retraction of $(\mathcal{I}_m, \mathcal{I}_c)^\#$ onto $\mathcal{I}_c^\#$.

Dually, if each comorphism in $\mathcal{I}_c$ is η-induced by some morphism in $\mathcal{I}_m$, then there is a retraction of $(\mathcal{I}_m, \mathcal{I}_c)^\#$ onto $\mathcal{I}_m^\#$.

Proof: Consider the pushout construction in Definition 6.35. Clearly, $\mathcal{I}_m^\#$, $\mathcal{I}_c^\#$ and $\mathcal{I}_0^\#$ all have the same object class. Moreover, since $\mathcal{I}_0$ is discrete, the signature morphisms in $\mathcal{I}_0^\#$ are basically those of the individual institutions. With this, it is easy to see that the signature morphisms in $(\mathcal{I}_m, \mathcal{I}_c)^\#$ are paths of morphisms coming from $\mathcal{I}_m^\#$ and $\mathcal{I}_c^\#$ in an alternating way.

The retraction of $(\mathcal{I}_m, \mathcal{I}_c)^\#$ onto $\mathcal{I}_c^\#$ (having the obvious inclusion as right inverse) is therefore given by the identity for the objects, while for a path of alternating morphisms, each morphism

$$(\ i \xrightarrow{d} j\ ,\ \Sigma_1 \xrightarrow{\sigma} \Psi^d(\Sigma_2)\ )\colon (i, \Sigma_1) \longrightarrow (j, \Sigma_2)$$

from $\mathcal{I}_m^\#$ is replaced with

$$(\ j \xrightarrow{e} i\ ,\ \Phi^e(\Sigma_1) \xrightarrow{\Phi^e(\sigma)} \Phi^e(\Psi^d(\Sigma_2)) \xrightarrow{\varepsilon_{\Sigma_2}} \Sigma_2\ ),$$

where e is the index of the comorphism inducing $\mathcal{I}_m(d)$, and ε the corresponding natural transformation. Since all the resulting morphisms live in $\mathcal{I}_c^\#$, they can be composed to a single morphism.

The other statement follows by a dual argument. □

However, unfortunately there are practically relevant situations where this is not applicable.

Claim 6.38 Typical feature interaction morphisms are not inducible. Typical simple theoroidal coding comorphisms are not inducible. Of course, "typical" is informal here. However, we give two examples below which are typical in the sense that the proofs carry over to many similar examples.

Proposition 6.39 Neither the morphism $pr\colon$ CSP-CASL $\longrightarrow$ CASL nor the simple theoroidal semi-comorphism $toLTL\colon$ CSP-CASL $\longrightarrow$ MODALCASL from Sect. 4.5 is inducible.

PROOF: Assume that $pr = (\Psi, \bar{\alpha}, \bar{\beta})\colon$ CSP-CASL $\longrightarrow$ CASL is ε-induced by a comorphism $\rho = (\Phi, \alpha, \beta)\colon$ CASL $\longrightarrow$ CSP-CASL. Let Σ_1 consist of a sort s and Σ_2 of sorts t and u (both seen as CSP-CASL-signatures). Let $\sigma\colon \Psi(\Sigma_2) \longrightarrow \Psi(\Sigma_1)$ map both t and u to s (recall that Ψ is just an inclusion). Now $\bar{\beta}_{\Sigma_2}$ just forgets the optional LTS component, and hence is surjective. Since $\bar{\beta}_{\Sigma_2} = \beta_{\Psi(\Sigma_2)} \circ \varepsilon_{\Sigma_2}$, $\beta_{\Psi(\Sigma_2)}$ is surjective as well. Since all signature morphisms in CSP-CASL are injective (and carriers are assumed to be non-empty), the corresponding reduct functors are easily seen to be surjective. Hence, the lower right path in the naturality diagram for β

$$\begin{array}{ccc} \mathbf{Mod}^{\text{CSP-CASL}}(\Phi(\Psi(\Sigma_1))) & \xrightarrow{\beta_{\Psi(\Sigma_1)}} & \mathbf{Mod}^{\text{CASL}}(\Psi(\Sigma_1)) \\ \downarrow{\scriptstyle -|_{\Phi(\sigma)}} & & \downarrow{\scriptstyle -|_{\sigma}} \\ \mathbf{Mod}^{\text{CSP-CASL}}(\Phi(\Psi(\Sigma_2))) & \xrightarrow{\beta_{\Psi(\Sigma_2)}} & \mathbf{Mod}^{\text{CASL}}(\Psi(\Sigma_2)) \end{array}$$

is surjective as well. Hence, also the upper left path must be surjective, and hence its second component $-|_\sigma$. But $-|_\sigma$ just doubles the carrier set, and this is clearly not surjective.

The semi-comorphism $toLTL$ is more precisely defined on the subinstitution CSP-CASL-d consisting of identity signature morphisms only, i.e. $toLTL = (\Phi, \beta)\colon$ CSP-CASL-d $\longrightarrow$ MODALCASL. Assume that it is η-induced by a semi-morphism $\mu = (\Psi, \bar{\beta})\colon$ MODALCASL $\longrightarrow$ CSP-CASL-d. Since all signature morphisms in CSP-CASL-d are identities, η is the identity as well. Hence, $\bar{\beta} \cdot \Phi = \beta$, and one easily obtains a contradiction to the $\bar{\beta}$-naturality diagram for a signature morphism $\sigma\colon \Phi(\Sigma_1) \longrightarrow \Phi(\Sigma_2)$ in MODALCASL. This proof relies on the severe restrictedness of the CSP-CASL-d signature morphisms; however, also a proof not exploiting this is possible. □

Proposition 6.40 The "feature interaction" institution morphism μ going from first-order logic with equality $FOL^=$ to first-order logic FOL described in Sect. 2.10 is not ε-inducible.

The "coding" simple theoroidal comorphism $\rho\colon FOL^= \longrightarrow FOL$ described in Sect. 2.10 is not η-inducible.

PROOF: Suppose that $\mu = (\Psi, \bar{\alpha}, \bar{\beta})\colon FOL^= \longrightarrow FOL$ is ε-induced by some comorphism $\rho = (\Phi, \alpha, \beta)\colon FOL \longrightarrow FOL^=$. Let Σ_1 consist of a sort s and Σ_2 additionally of a predicate $R : s \times s$ (both seen as $FOL^=$-signatures). Since $\bar{\alpha}_{\Sigma_2}$ obviously is surjective and $\bar{\alpha}_{\Sigma_2} = \mathbf{Sen}^J(\varepsilon_{\Sigma_2}) \circ \alpha_{\Psi(\Sigma_2)}$, $\mathbf{Sen}^J(\varepsilon_{\Sigma_2})$ and therefore also $\varepsilon_{\Sigma_2}\colon \Phi(\Psi(\Sigma_2)) \longrightarrow \Sigma_2$ have to be surjective as well. Since Σ_2 contains a binary relation symbol, by surjectivity of ε_{Σ_2}, $\Phi(\Psi(\Sigma_2))$ must contain one as well. Now there exists a signature morphism $\sigma\colon \Psi(\Sigma_2) \longrightarrow \Psi(\Sigma_1)$ (mapping both R and eq_s to eq_s). But then $\varepsilon_{\Sigma_1} \circ \Phi(\sigma)\colon \Phi(\Psi(\Sigma_2)) \longrightarrow \Sigma_1$ has to map the binary relation symbol. However, this is not possible, since Σ_1 does not contain one. Hence, μ cannot be inducible.

Concerning the second statement of the proposition, assume that $\rho = (\Phi, \alpha, \beta)\colon FOL^= \longrightarrow FOL$ is η-induced by some morphism $\mu = (\Psi, \bar{\alpha}, \bar{\beta})\colon FOL \longrightarrow FOL^=$. Then η_Σ is injective (if not, $\mathbf{Mod}^{FOL^=}(\eta_\Sigma)$ has in its image only models where at least two components are identical, while this is not the case for β_Σ, contradicting $\beta = (\mathbf{Mod}^J \cdot \eta) \circ (\bar{\beta} \cdot Sig \cdot \Phi^{op})$). W.l.o.g. we can assume that η_Σ is an inclusion, hence for $\varphi \in \mathbf{Sen}^{FOL^=}(\Sigma)$, $\alpha_\Sigma(\varphi) = \bar{\alpha}_{Sig(\Phi(\Sigma))}(\varphi)$. Let Σ_2 be as in the above

proof, and let $\sigma\colon Sig(\Phi(\Sigma_2)) \longrightarrow Sig(\Phi(\Sigma_2))$ map both R and $\equiv$ to R.

$$\begin{array}{ccc}
\mathbf{Sen}(\Psi(Sig(\Phi(\Sigma_2)))) & \xrightarrow{\bar{\alpha}_{Sig(\Phi(\Sigma_2))}} & \mathbf{Sen}(Sig(\Phi(\Sigma_2))) \\
\downarrow{\scriptstyle \mathbf{Sen}^{FOL^=}(\Psi(\sigma))} & & \downarrow{\scriptstyle \mathbf{Sen}^{FOL}(\sigma)} \\
\mathbf{Sen}(\Psi(Sig(\Phi(\Sigma_2)))) & \xrightarrow{\bar{\alpha}_{Sig(\Phi(\Sigma_2))}} & \mathbf{Sen}(Sig(\Phi(\Sigma_2)))
\end{array}$$

Now for $\varphi \in \mathbf{Sen}^{FOL^=}(\Sigma_2)$, $\alpha_{\Sigma_2}(\varphi) = \bar{\alpha}_{Sig(\Phi(\Sigma_2))}(\varphi)$ is obtained by replacing $=$ with $\equiv$. Now by definition of $FOL^=$, $\mathbf{Sen}^{FOL^=}(\Psi(\sigma))$ leaves $=$ unchanged, and $\mathbf{Sen}^{FOL}(\sigma)$ maps $\equiv$ to R. Hence, the above naturality diagram for $\bar{\alpha}$ cannot commute. This shows that ρ cannot be inducible.

□

The examples from Prop. 6.40 are simple but quite typical. E.g. when considering the analogous translations between CASL and CASL-LT sketched in [Mos02b], the main structure of the above proofs carries over.

6.10 Spans of Comorphisms

The method of the previous section to use inducibility to reduce the complexity of heterogeneous specifications involving different kinds of translations between institutions works for some cases, but the counterexamples of Proposition 6.39 have shown that the method is not general enough.

A more general idea is to express all the different kinds of translations as *spans* of morphisms or of comorphisms. For reasons explained in Sect. 6.7, we work with (spans of) comorphisms here. Nevertheless, the results presented below easily dualize to spans of morphisms.

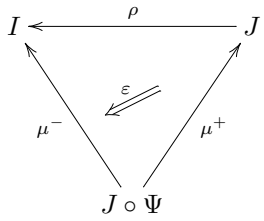
Each institution morphism $\mu\colon I \longrightarrow J = I \xrightarrow{\Psi} \xleftarrow{\alpha} \xrightarrow{\beta} J$ can be translated into a span $I \xleftarrow{\mu^-} J \circ \Psi \xrightarrow{\mu^+} J$ of institution comorphisms as follows:

$$\begin{array}{ccccc}
\mathbf{Sign}^I & \xleftarrow{id} & \mathbf{Sign}^I & \xrightarrow{\Psi} & \mathbf{Sign}^J \\
\mathbf{Sen}^I & \xleftarrow{\alpha} & \mathbf{Sen}^J \circ \Psi & \xrightarrow{id} & \mathbf{Sen}^J \circ \Psi \\
\mathbf{Mod}^I & \xrightarrow{\beta} & \mathbf{Mod}^J \circ \Psi^{op} & \xleftarrow{id} & \mathbf{Mod}^J \circ \Psi^{op}
\end{array}$$

Here, the "middle" institution $J \circ \Psi$ is the institution with signature category inherited from I, but sentences and models inherited from J via Ψ.

In the case that μ happens to be inducible, there is also a relation to the inducing comorphism:

Proposition 6.41 If a comorphism $\rho\colon I \longrightarrow J$ ε-induces the morphism $\mu\colon J \longrightarrow I$, then there is a comorphism modification

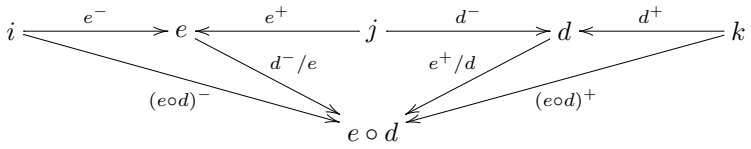


PROOF: Using the notation of institutions as functors, we need to show $(I \cdot \varepsilon) \circ (\rho \cdot \Psi) \circ \mu^+ = \mu^-$, where ρ, μ^+ and μ^- are just the natural transformations (i.e. without signature translation). But since $\mu^+ = id$ and $\mu^- = \mu$, we just arrive at the inducibility condition $(I \cdot \varepsilon) \circ (\rho \cdot \Psi) = \mu$. □

The span construction can also be lifted to the indexed level by freely adding formal objects and morphisms (modulo some quotient) to the index category of $\mathcal{I}_c$.

Definition 6.42 Given an indexed institution $\mathcal{I}_m : Ind_m^{op} \longrightarrow \mathbf{Ins}$ and an indexed coinstitution $\mathcal{I}_c : Ind_c^{op} \longrightarrow \mathbf{CoIns}$ (both over the same set of institutions in the sense of Definition 6.35), form an indexed coinstitution $Span(\mathcal{I}_m, \mathcal{I}_c) = \mathcal{I} : Ind^{op} \longrightarrow \mathbf{CoIns}$ as follows: Ind is the free category [Sch99] obtained by adjoining the following to Ind_c:

- for each $d : j \longrightarrow i \in Ind_m$, an object d and a pair of morphisms $i \xrightarrow{d^-} d \xleftarrow{d^+} j$,
- for each $k \xrightarrow{d} j \xrightarrow{e} i \in Ind_m$, a morphism $d^-/e : e \longrightarrow e \circ d$ and a morphism $e^+/d : d \longrightarrow e \circ d$, and
- for each $k \xrightarrow{d} j \xrightarrow{e} i \in Ind_m$, the following three commutativity conditions are imposed:



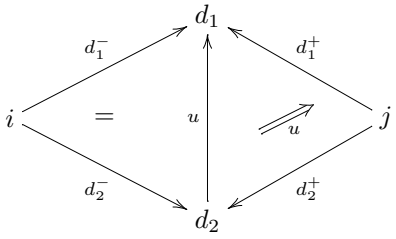
The action of $\mathcal{I}$ is $\mathcal{I}_c$ on Ind_c and as follows on the newly added structure (using the notation $(\mathbf{Sign}^i, \mathcal{I}_m^i)$ for $\mathcal{I}_m(i)$ and (Ψ^d, μ^d) for $\mathcal{I}_m(d)$):

- for $d : j \longrightarrow i \in Ind_m$, $\mathcal{I}(d) = (\mathbf{Sign}^i, \mathcal{I}_m^j \circ \Psi^d)$ (note that d becomes an object in Ind),
- for $d : j \longrightarrow i \in Ind_m$, $\mathcal{I}(d^-) = (id, \mu^d)$ and $\mathcal{I}(d^+) = (\Psi^d, id)$, and
- for $k \xrightarrow{d} j \xrightarrow{e} i \in Ind_m$, $\mathcal{I}(d^-/e) = (id, \mu^d \cdot \Psi^e)$ and $\mathcal{I}(e^+/d) = (\Psi^e, id)$.

It is straightforward to check that $\mathcal{I}$ actually maps that above three triangles to commutative triangles:

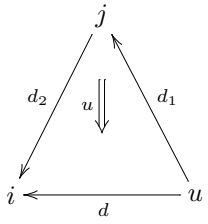
- $\mathcal{I}(e^-) \circ \mathcal{I}(d^-/e) = (id, \mu^e) \circ (id, \mu^d \cdot \Psi^e) = (id, \mu^e \circ (\mu^d \cdot \Psi^e)) = \mathcal{I}((e \circ d)^-)$
- $\mathcal{I}(e^+) \circ \mathcal{I}(d^-/e) = (\Psi^e, id) \circ (id, \mu^d \cdot \Psi^e) = (\Psi^e, \mu^d \cdot \Psi^e) = (id, \mu^d) \circ (\Psi^e, id) = \mathcal{I}(d^-) \circ \mathcal{I}(e^+/d)$
- $\mathcal{I}(d^+) \circ \mathcal{I}(e^+/d) = (\Psi^d, id) \circ (\Psi^e, id) = (\Psi^d \circ \Psi^e, id) = (\Psi^{e \circ d}, id) = \mathcal{I}((e \circ d)^+)$

We can extend this also two 2-cells. Given morphisms $d_1, d_2 : j \longrightarrow i$ and a 2-cell $u : d_2 \longrightarrow d_1$ in Ind_m, add a morphism $u : d_2 \longrightarrow d_1$ and a 2-cell $u : u \circ d_2^+ \Longrightarrow d_1^+$ in $Span(\mathcal{I}_m, \mathcal{I}_c)$.

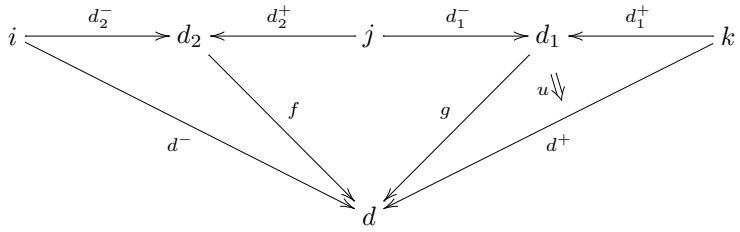


with $\mathcal{I}(u\colon d_2 \longrightarrow d_1) = (id, \mathcal{I}_m^j \cdot \mathcal{I}_m(u))$ and $\mathcal{I}(u\colon u \circ d_2^+ \Longrightarrow d_1^+) = \mathcal{I}_m(u)$.

For practical purposes (i.e. in the Heterogeneous Tool Set, see Chap. 7), we will not work with a category I_m, but rather with kind of composition graph, where the compositions are given via lax triangles:



leading to a diagram where $d_2 \circ d_1$ is not needed at all, but replaced with d instead:



and the new formal arrows f and g replacing $u \circ d_1^-/d_2$ and $u \circ d_2^+/d_1$, respectively. f and g are interpreted under $\mathcal{I}$ as the arrows they shall replace:

$$\mathcal{I}(f) = (id, (\mu^{d_1} \cdot \Psi^{d_2}) \circ (\mathcal{I}_m^k \cdot \mathcal{I}_m(u)))$$
$$\mathcal{I}(g) = (\Psi^{d_2}, \mathcal{I}_m^k \cdot \mathcal{I}_m(u))$$

Moreover,

$$\mathcal{I}(u\colon g \circ d_1^+ \Longrightarrow d^+) = \mathcal{I}_m(u).$$

Concerning the relation to the Bi-Grothendieck institution, unfortunately, we cannot expect that Theorem 6.37 carries over to the present situation. But we have some weaker property that still is sufficiently strong for practical needs:

Theorem 6.43 Given an indexed institution $\mathcal{I}_m$ and an indexed coinstitution $\mathcal{I}_c$ (both over the same set of institutions in the sense of Definition 6.35), each development graph over the Bi-Grothendieck institution $(\mathcal{I}_m, \mathcal{I}_c)^\#$ can be translated into a development graph over the Grothendieck institution over the span-based indexed coinstitution $Span(\mathcal{I}_m, \mathcal{I}_c)^\#$, such that model categories are preserved.

PROOF: As in the proof of Theorem 6.37, we rely on the fact that signature morphisms in the Bi-Grothendieck institution $(\mathcal{I}_m, \mathcal{I}_c)^\#$ are paths of morphisms coming from $\mathcal{I}_m^\#$ and $\mathcal{I}_c^\#$ in an alternating way. A global definition link therefore has the form

$$K \xrightarrow{\langle (e_1,\sigma_1),(d_2,\sigma_2),\ldots,(e_n,\sigma_n)\rangle} N\ ,$$

where the d_i are from $\mathcal{I}_m$ and the e_i are from $\mathcal{I}_c$ (with (e_1, σ_1), (e_n, σ_n) possibly not present). The definition link now is replaced by a sequence of definition and hiding links:

$$K \xrightarrow{(e_1,\sigma_1)} K_1 \xrightarrow{(id,\sigma_2)} K_2 \underset{h}{\xrightarrow{(d_2^+,id)}} K_3 \xrightarrow{(d_2^-,id)} K_4 \longrightarrow \cdots \xrightarrow{(e_n,\sigma_n)} N$$

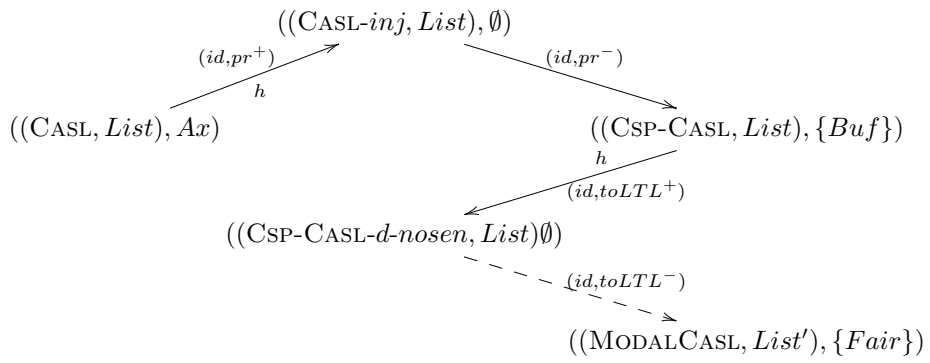


Figure 6.5: The sample heterogeneous development graph with spans of comorphisms.

Here, d_2^+ and d_2^- are the indices for the span of comorphisms associated to $\mathcal{I}_m(d_2)$, and $K_1, \ldots, K_4$ are new nodes with appropriate signatures and no local axioms. Of course, the path could also start and/or end with a d_i instead of an e_i, but this won't affect the general construction: each path element containing a d_i leads to a sequence of a definition link, a hiding link, and again a definition link, while path elements containing an e_i are just kept. The construction for theorem links is entirely analogous, except that only the last arrow in the sequence has to be a theorem link — the other ones must be definition links.

Let us now come to hiding links. The construction is very similar, so we restrict ourselves to the replacement of individual path elements of form (d_i, σ_i). Such an element leads to

$$\cdots \longrightarrow K_n \xrightarrow[h]{(d_i^-, id)} K_{n+1} \xrightarrow{(d_i^+, id)} K_{n+2} \xrightarrow[h]{(id, \sigma_i)} K_{n+3} \longrightarrow \cdots$$

In comparison to the construction above, here the arrows are reversed, and definition and hiding links interchanged.

It is straightforward to see that the model class is left unchanged by these translations. □

Example 6.44 Extend the institutions and (co)morphisms introduced in Sect. 4.5 by the following ones:

- CASL-*inj* is the restriction of CASL to signature that are injective on sorts.
- CSP-CASL-*d-nosen* is the restriction of CSP-CASL to the empty set of sentences, for each signature, and to identity signature morphisms.
- The comorphism pr^+: CASL-*inj* $\longrightarrow$ CASL is just the obvious subinstitution inclusion.
- The comorphism pr^-: CASL-*inj* $\longrightarrow$ CSP-CASL behaves very similar to pr: At the signature level, it is the identity, at the sentence level, it is the obvious inclusion. For models, just the LTS (if present) is forgotten.

When applying the construction of Theorem 6.43 to (a formalized variant of) the development graph given in Fig. 1.9, we arrive at the development graph shown in Fig. 6.5 (for simplicity, we index the involved institutions and comorphisms by themselves here).

Let us now consider the other types of morphisms between institutions that have been introduced in Chap. 2.[8] To begin with, a semi-comorphism $I \;\overset{\Phi}{\underset{\beta}{\rightleftarrows}}\; J$ can be translated into a span

[8] We adopt the convention that the μ^+-component always goes along with μ (and its signature translation), whereas the μ^--component goes against it.

$I \xleftarrow{\mu^-} I^\emptyset \xrightarrow{\mu^+} J$ of comorphisms

$$\begin{array}{ccccc}
\mathbf{Sign}^I & \xleftarrow{id} & \mathbf{Sign}^I & \xrightarrow{\Phi} & \mathbf{Sign}^J \\
\mathbf{Sen}^I & \xleftarrow{incl} & \emptyset & \xrightarrow{incl} & \mathbf{Sen}^J \circ \Phi \\
\mathbf{Mod}^I & \xrightarrow{id} & \mathbf{Mod}^I & \xleftarrow{\beta} & \mathbf{Mod}^J \circ \Phi
\end{array}$$

while a semi-morphism $I \xrightarrow[\beta]{\Phi} J$ is translated into a span $I \xleftarrow{\mu^-} J \circ \Phi^\emptyset \xrightarrow{\mu^+} J$ of comorphisms

$$\begin{array}{ccccc}
\mathbf{Sign}^I & \xleftarrow{id} & \mathbf{Sign}^I & \xrightarrow{\Phi} & \mathbf{Sign}^J \\
\mathbf{Sen}^I & \xleftarrow{incl} & \emptyset & \xrightarrow{incl} & \mathbf{Sen}^J \circ \Phi \\
\mathbf{Mod}^I & \xrightarrow{\beta} & \mathbf{Mod}^J \circ \Phi & \xleftarrow{id} & \mathbf{Mod}^J \circ \Phi
\end{array}$$

where in each case the "middle" institution has the indicated components.

Forward comorphisms $\mu = (\Phi, \alpha, \beta) \colon I \longrightarrow J = I \xrightarrow[\alpha]{\Phi} J$ are translated into spans of form $I \xleftarrow{\mu^-} J \circ \Phi^{\mathbf{Sen}} \xrightarrow{\mu^+} J$ consisting of institution comorphisms as follows:

$$\begin{array}{ccccc}
\mathbf{Sign}^I & \xleftarrow{id} & \mathbf{Sign}^I & \xrightarrow{\Phi} & \mathbf{Sign}^J \\
\mathbf{Sen}^I & \xleftarrow{\alpha} & \mathbf{Sen}^J \circ \Phi & \xrightarrow{id} & \mathbf{Sen}^J \circ \Phi \\
\mathbf{Mod}^I & \xrightarrow{id} & \mathbf{Mod}^I & \xleftarrow{\beta} & \mathbf{Mod}^J \circ \Phi
\end{array}$$

The "middle" institution $J \circ \Phi^{\mathbf{Sen}}$ inherits signatures and models from I, but sentences (via Φ) from J. The satisfaction relation $M \models_\Sigma^{J \circ \Phi^{\mathbf{Sen}}} \varphi$ holds iff $M \models_\Sigma \alpha_\Sigma(\varphi)$ in I.

Dually, a forward morphism $\mu = (\Phi, \alpha, \beta) \colon I \longrightarrow J = I \xrightarrow[\alpha]{\Phi} J$ can be translated into a span $I \xleftarrow{\mu^-} J \circ \Phi^{\mathbf{Mod}} \xrightarrow{\mu^+} J$ of institution comorphisms as follows:

$$\begin{array}{ccccc}
\mathbf{Sign}^I & \xleftarrow{id} & \mathbf{Sign}^I & \xrightarrow{\Phi} & \mathbf{Sign}^J \\
\mathbf{Sen}^I & \xleftarrow{id} & \mathbf{Sen}^I & \xrightarrow{\alpha} & \mathbf{Sen}^J \circ \Phi \\
\mathbf{Mod}^I & \xrightarrow{\beta} & \mathbf{Mod}^J \circ \Phi & \xleftarrow{id} & \mathbf{Mod}^J \circ \Phi
\end{array}$$

The "middle" institution $J \circ \Phi^{\mathbf{Mod}}$ inherits signatures and sentences from I, but models (via Φ) from J. The satisfaction relation $M \models_\Sigma^{J \circ \Phi^{\mathbf{Mod}}} \varphi$ holds iff $M \models_{\Phi(\Sigma)} \alpha_\Sigma(\varphi)$ in J.

(One could complete the picture and define, for a comorphism μ, μ^+ to be μ itself, and μ^- the identity.)

6.11 The Heterogeneous Verification Semantics

The various notions of institution translations introduced in Sect. 2.10 naturally lead to the following heterogeneous specification constructs [Tar00, Tar04], which can be seen as a core language for the language HetCasl as described in Appendix A.

***heterogeneous translation*:** For any institution comorphism, forward comorphism or semi-comorphism $\mu = (\Phi, \alpha, \beta)\colon I \longrightarrow I'$ and Σ-specification SP in I, **translate** SP **by** μ is a specification with:

$$\begin{aligned} &\mathit{Sig}[\textbf{translate } SP \textbf{ by } \mu] &&:= \Phi(\Sigma) \\ &\textbf{Mod}[\textbf{translate } SP \textbf{ by } \mu] &&:= \{M' \in \textbf{Mod}(\Phi(\Sigma)) \mid \beta_\Sigma(M') \in \textbf{Mod}[SP]\} \end{aligned}$$

***heterogeneous hiding*:** For any institution morphism, forward morphism or semi-morphism $\mu = (\Phi, \alpha, \beta)\colon I \longrightarrow I'$ and Σ-specification SP in I, **derive from** SP **by** μ is a specification with:

$$\begin{aligned} &\mathit{Sig}[\textbf{derive from } SP \textbf{ by } \mu] &&:= \Phi(\Sigma) \\ &\textbf{Mod}[\textbf{derive from } SP \textbf{ by } \mu] &&:= \{\beta_\Sigma(M') \mid M' \in \textbf{Mod}[SP]\} \end{aligned}$$

Some heterogeneous calculus rules have been given already in [Tar04]; however, they cover only the question whether a heterogeneous specification entails a sentence, but not the question whether a heterogeneous specification entails (or refines to) another one.

The purpose of the heterogeneous verification semantics is to provide a proof calculus for heterogeneous specifications covering both entailment and refinement. Its rules follow a similar verification semantics for (CASL) structured specifications given in Sect. 5.5 and in [MHAH04].

General assumption We assume to work with an indexed coinstitution that contains all the "middle" institutions as well as the μ^- and μ^+ comorphisms given by the constructions of the previous section, applied to those morphisms, semi-morphisms etc. that we expect to occur in our heterogeneous specifications. Of course, we assume that all institutions and comorphisms that are used directly are included as well. To formalize this, one needs to extend Def. 6.42 appropriately. In particular, we assume that the "middle" institutions are always indexed by the index morphism of the morphism, semi-morphism, forward comorphism etc. at hand. Actually, we do not spell out all the details for other types of (co)morphisms as we did in Def. 6.42 for the morphisms, because we expect that these (co)morphisms will not come with compositions anyway, but rather are given by a graph, while compositions are given by 2-cells.

The heterogeneous verification semantics now takes a heterogeneous specification and translates it into a development graph over the Grothendieck institution induced by the indexed coinstitution given by the above assumption.

In the sequel, we use the notation for constructing development graphs introduced in Sect. 5.5. Furthermore, by abuse of notation, we identify institutions and comorphisms with their respective indices in the index category of the indexed coinstitution (in general, it is expected that the indexed coinstitution is an embedding of categories; hence this abuse of notation will not lead to ambiguities).

The verification semantics uses judgements of form

$$\vdash SP \rightarrow\!\!\!\!\!\blacktriangleright\!\!\blacktriangleright (N, \mathcal{DG})$$

which read as: the specification SP is translated to the node N in development graph $\mathcal{DG}$.

$$\frac{\Sigma \in \textbf{Sign}^i}{\vdash \langle \Sigma, \Psi \rangle \rightarrow\!\!\!\!\!\blacktriangleright\!\!\blacktriangleright (N, \{N := ((i, \Sigma), \Psi)\})}$$

$$\frac{\vdash SP_1 \rightarrow\!\!\!\!\!\blacktriangleright\!\!\blacktriangleright (N_1, \mathcal{DG}_1) \qquad \vdash SP_2 \rightarrow\!\!\!\!\!\blacktriangleright\!\!\blacktriangleright (N_2, \mathcal{DG}_2) \qquad \Sigma^{N_1} = \Sigma^{N_2}}{\vdash SP_1 \cup SP_2 \rightarrow\!\!\!\!\!\blacktriangleright\!\!\blacktriangleright (K, \mathcal{DG}_1 \uplus \mathcal{DG}_2 \uplus \{K := (\Sigma^{N_1}, \emptyset)\} \uplus \{ N_i \xrightarrow{id} K \mid i = 1, 2\})}$$

$$\frac{\vdash SP \rightarrow\!\!\!\!\!\blacktriangleright\!\!\blacktriangleright (N, \mathcal{DG}) \qquad \Sigma^N = (i, \Sigma)}{\vdash \textbf{translate } SP \textbf{ by } \sigma\colon \Sigma \longrightarrow \Sigma' \rightarrow\!\!\!\!\!\blacktriangleright\!\!\blacktriangleright (K, \mathcal{DG} \uplus \{ N \xrightarrow{(\sigma, id)} K := ((i, \Sigma'), \emptyset)\})}$$

$$\frac{\vdash SP \rhd\!\!\rhd\!\!\rhd (N, \mathcal{DG}) \quad \Sigma^N = (i, \Sigma')}{\vdash \textbf{derive from } SP' \textbf{ by } \sigma\colon \Sigma \longrightarrow \Sigma' \rhd\!\!\rhd\!\!\rhd (K, \mathcal{DG} \uplus \{ N \xRightarrow[hide]{(\sigma,id)} K := ((i, \Sigma), \emptyset)\})}$$

$$\frac{\vdash SP \rhd\!\!\rhd\!\!\rhd (N, \mathcal{DG}) \quad \Sigma^N = (i, \Sigma) \quad d\colon j \longrightarrow i \text{ is the index of a comorphism}}{\vdash \textbf{translate } SP \textbf{ by } d \rhd\!\!\rhd\!\!\rhd (K, \mathcal{DG} \uplus \{ N \xRightarrow{(id,d)} K := ((j, \Phi^d(\Sigma)), \emptyset)\})}$$

$$\frac{\vdash SP \rhd\!\!\rhd\!\!\rhd (N, \mathcal{DG}) \quad \Sigma^N = (i, \Sigma) \quad d\colon j \longrightarrow i \text{ is the index of a morphism} \quad \mathcal{DG}' = \mathcal{DG} \uplus \{ N \xRightarrow[hide]{(id,d^-)} K := ((d, \Sigma), \emptyset); \quad K \xRightarrow{(id,d^+)} P := ((j, \Phi^d(\Sigma)), \emptyset)\}}{\vdash \textbf{derive from } SP \textbf{ by } d \rhd\!\!\rhd\!\!\rhd (P, \mathcal{DG}')}$$

$$\frac{\vdash SP \rhd\!\!\rhd\!\!\rhd (N, \mathcal{DG}) \quad \Sigma^N = (i, \Sigma) \quad d\colon j \longrightarrow i \text{ is a semi-comorphism} \quad \mathcal{DG}' = \mathcal{DG} \uplus \{ N \xRightarrow[hide]{(id,d-)} K := ((d, \Sigma), \emptyset); \quad K \xRightarrow{(id,d+)} P := ((j, \Phi^d(\Sigma)), \emptyset)\}}{\vdash \textbf{translate } SP \textbf{ by } d \rhd\!\!\rhd\!\!\rhd (P, \mathcal{DG}')}$$

$$\frac{\vdash SP \rhd\!\!\rhd\!\!\rhd (N, \mathcal{DG}) \quad \Sigma^N = (i, \Sigma) \quad d\colon j \longrightarrow i \text{ is the index of a semi-morphism} \quad \mathcal{DG}' = \mathcal{DG} \uplus \{ N \xRightarrow[hide]{(id,d^-)} K := ((d, \Sigma), \emptyset); \quad K \xRightarrow{(id,d^+)} P := ((j, \Phi^d(\Sigma)), \emptyset)\}}{\vdash \textbf{derive from } SP \textbf{ by } d \rhd\!\!\rhd\!\!\rhd (P, \mathcal{DG}')}$$

$$\frac{\vdash SP \rhd\!\!\rhd\!\!\rhd (N, \mathcal{DG}) \quad \Sigma^N = (i, \Sigma) \quad d\colon j \longrightarrow i \text{ is the index of a forward comorphism} \quad \mathcal{DG}' = \mathcal{DG} \uplus \{ N \xRightarrow[hide]{(id,d^-)} K := ((d, \Sigma), \emptyset); \quad K \xRightarrow{(id,d^+)} P := ((j, \Phi^d(\Sigma)), \emptyset)\}}{\vdash \textbf{translate } SP \textbf{ by } d \rhd\!\!\rhd\!\!\rhd (P, \mathcal{DG}')}$$

$$\frac{\vdash SP \rhd\!\!\rhd\!\!\rhd (N, \mathcal{DG}) \quad \Sigma^N = (i, \Sigma) \quad d\colon j \longrightarrow i \text{ is the index of a forward morphism} \quad \mathcal{DG}' = \mathcal{DG} \uplus \{ N \xRightarrow[hide]{(id,d^-)} K := ((d, \Sigma), \emptyset); \quad K \xRightarrow{(id,d^+)} P := ((i, \Phi({}^d\Sigma)), \emptyset)\}}{\vdash \textbf{derive from } SP \textbf{ by } d \rhd\!\!\rhd\!\!\rhd (P, \mathcal{DG}')}$$

Although many of the clauses look very similar, note that the μ^--comorphisms for the various types of morphisms inherit their model translations from μ, while for the various types of comorphisms, it is the μ^+-comorphisms that inherit the model translation from μ.

We now state the important property of the heterogeneous verification semantics:

Theorem 6.45 The heterogeneous verification semantics preserves model classes of heterogeneous specifications. More precisely, given a heterogenous specification SP with $\vdash SP \rhd\!\!\rhd\!\!\rhd (N, \mathcal{DG})$, we have

$$\mathbf{Mod}[SP] = \mathbf{Mod}_{\mathcal{DG}}(N)$$

PROOF: Straightforward induction over the structure of SP. E.g. if d is the index of a comorphism,

$$\begin{array}{lll}
\mathbf{Mod}(\textbf{translate } SP \textbf{ by } d\colon j \longrightarrow i) & = & \\
\{M \in \mathbf{Mod}^j(\Phi^d(\Sigma)) \mid \beta^d_\Sigma(M) \in \mathbf{Mod}(SP)\} & = & \text{(induction hypothesis)} \\
\{M \in \mathbf{Mod}^j(\Phi^d(\Sigma)) \mid \beta^d_\Sigma(M) \in \mathbf{Mod}_{\mathcal{DG}}(N)\} & = & \\
\mathbf{Mod}_{\mathcal{DG}'}(K) & &
\end{array}$$

If d is the index of a morphism,

$$\begin{array}{lll}
\mathbf{Mod}(\textbf{derive from } SP \textbf{ by } d\colon j \longrightarrow i) & = & \\
\{\beta^d_\Sigma(M) \in \mathbf{Mod}^j(\Phi^d(\Sigma)) \mid M \in \mathbf{Mod}(SP)\} & = & \text{(induction hypothesis)} \\
\{\beta^d_\Sigma(M) \in \mathbf{Mod}^j(\Phi^d(\Sigma)) \mid M \in \mathbf{Mod}_{\mathcal{DG}}(N)\} & = & \\
\mathbf{Mod}_{\mathcal{DG}'}(K) & = & (\subseteq \mathbf{Mod}^d(\Sigma)) \\
\mathbf{Mod}_{\mathcal{DG}'}(P) & & (\subseteq \mathbf{Mod}^j(\Phi^d(\Sigma)))
\end{array}$$

If d is the index of a semi-comorphism,

$$\begin{array}{lll}
\mathbf{Mod}(\textbf{translate } SP \textbf{ by } d\colon j \longrightarrow i) & = & \\
\{M \in \mathbf{Mod}^j(\Phi^d(\Sigma)) \mid \beta^d_\Sigma(M) \in \mathbf{Mod}(SP)\} & = & \text{(induction hypothesis)} \\
\{M \in \mathbf{Mod}^j(\Phi^d(\Sigma)) \mid \beta^d_\Sigma(M) \in \mathbf{Mod}_{\mathcal{DG}}(N)\} & = & \\
\{M \in \mathbf{Mod}^d(\Phi^d(\Sigma)) \mid \beta^d_\Sigma(M) \in \mathbf{Mod}_{\mathcal{DG}}(K)\} & = & \\
\mathbf{Mod}_{\mathcal{DG}'}(P) & &
\end{array}$$

□

By inserting theorem links into a development graph generated by the heterogeneous verification semantics, it is now possible to tackle the problem of refinement between heterogeneous specifications with the proof calculus of Sect. 6.4 and 5.6.

A word concerning the difference between morphisms and semi-morphisms is in order: although they are treated quite similarly in the heterogeneous verification semantics, their difference shows up when using the proof calculus: the "middle" institution is much poorer in the case of semi-morphisms (it has no sentences). The latter makes it much harder to conduct heterogeneous proofs; indeed, for proofs along semi-morphisms, typically both the source and target institution have to be translatable into some common target institution (or some special proof rules for the particular semi-morphism has to be added). Of course, a similar remark applies to semi-comorphisms as well.

6.12 Representation Maps

The notion of institution representation map [Tar96] between institution comorphisms is an important concept for extending the well-known borrowing technique (i.e. translating proof systems along institution comorphisms) to the heterogeneous case. We here study its relation to our approach based on comorphism modifications.

Fix an institution $U = (\mathbf{USign}, \mathbf{USen}, \mathbf{UMod}, \models)$ which we will very informally view as a "universal" institution (with sufficient expressiveness to represent many logics, and with suitable tool support).

Given two institution comorphisms (called representations in [Tar96]) $\rho_1\colon I_1 \longrightarrow U$ and $\rho_2\colon I_2 \longrightarrow U$, a *representation map* $(\mu, \theta)\colon \rho_1 \longrightarrow \rho_2$ consists of

- a morphism $\mu = (\Psi, \alpha, \beta)\colon I_1 \longrightarrow I_2$ and
- a natural transformation $\theta\colon \Phi_2 \circ \Psi \longrightarrow \Phi_1$

such that $(\mathbf{Sen}^U \cdot \theta) \circ (\alpha_2 \cdot \Psi) = \alpha_1 \circ \alpha$ and $(\beta_2 \cdot \Psi) \circ (\mathbf{Mod}^U \cdot \theta) = \beta \circ \beta_1$.

The latter conditions mean that that for each signature $\Sigma \in |\mathbf{Sign}|$ the following diagrams commute:

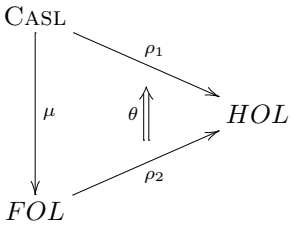
$$\begin{array}{ccc}
\mathbf{Sen}^{I_1}(\Sigma) & \xrightarrow{(\alpha_1)_\Sigma} & \mathbf{USen}(\Psi_1(\Sigma)) \\
\uparrow{\scriptstyle \alpha_\Sigma} & & \uparrow{\scriptstyle \mathbf{USen}(\theta_\Sigma)} \\
\mathbf{Sen}^{I_2}(\Psi(\Sigma)) & \xrightarrow{(\alpha_2)_{\Psi(\Sigma)}} & \mathbf{USen}(\Psi_2(\Psi(\Sigma)))
\end{array}$$

$$\begin{array}{ccc}
\mathbf{Mod}^{I_1}(\Sigma) & \xleftarrow{(\beta_1)_\Sigma} & \mathbf{UMod}(\Psi_1(\Sigma)) \\
\downarrow{\scriptstyle \beta_\Sigma} & & \downarrow{\scriptstyle \mathbf{UMod}(\theta_\Sigma)} \\
\mathbf{Mod}^{I_2}(\Psi(\Sigma)) & \xleftarrow{(\beta_2)_{\Psi(\Sigma)}} & \mathbf{UMod}(\Psi_2(\Psi(\Sigma)))
\end{array}$$

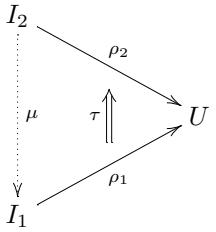
With an obvious composition, this gives us a category $Repr(U)$ of institution representations (=comorphisms) into U and representation maps.

Example 6.46 The representation map from CASL $\to$ HOL to $FOL \to HOL$ is defined as follows:

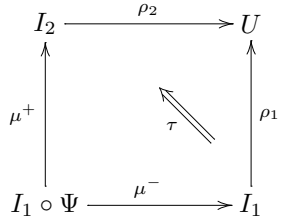
- μ forgets partiality and subsorting,
- ρ_1 encodes partiality and subsorting,
- ρ_2 is the inclusion, and
- θ is just the inclusion as well.



The concept of institution comorphism modification (see Sect. 2.12) is a specialization of the concept of representation map: just take the institution morphism component of the representation map to be the identity. However, when considering the span construction, it is also a generalization. Each representation map



leads to a comorphism modification in the square



Moreover, in case of adjointness, we have an even closer connection between the two notions:

Theorem 6.47 Given comorphisms $\rho_1 \colon I_1 \longrightarrow U$, $\rho_2 \colon I_1 \longrightarrow U$, and functors $\Phi \dashv \Psi \colon \mathbf{Sign}^{I_2} \longrightarrow \mathbf{Sign}^{I_1}$, the one-one correspondence of

- comorphisms $\rho\colon I_1 \longrightarrow I_2$ (extending Φ) and
- morphisms $\mu\colon I_2 \longrightarrow I_1$ (extending Ψ)

described in Proposition and Definition 2.18 extends to a one-one correspondence of

- pairs $(\rho\colon I_1 \longrightarrow I_2, \tau\colon \rho_1 \longrightarrow \rho_2 \circ \rho)$ of comorphisms and modifications, and
- representation maps $(\mu, \theta)\colon \rho_2 \longrightarrow \rho_1$.

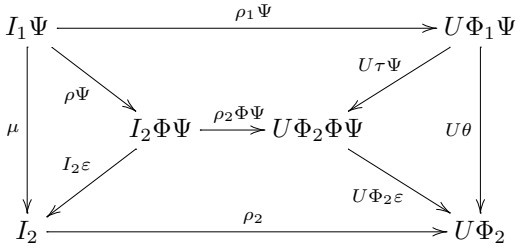
PROOF: We will use the notation of institutions as functors from Sect. 2.13. By a slight abuse of notation, we will write ρ_1 as $(\Phi_1, \rho_1)\colon I_1 \longrightarrow U$, where $\Phi_1\colon \mathbf{Sign}^{I_1} \longrightarrow \mathbf{Sign}^U$ and $\rho_1\colon I_1 \longrightarrow U \circ \Phi_1$, and similarly $(\Phi_2, \rho_2)\colon I_2 \longrightarrow U$ etc.

Given a comorphism $(\Phi, \rho)\colon I_1 \longrightarrow I_2$ and a modification $\tau\colon (\Phi_1, \rho_1) \longrightarrow (\Phi_2, \rho_2) \circ (\Phi, \rho)$, i.e. $\tau\colon \Phi_1 \longrightarrow \Phi_2 \circ U$ with $(U \cdot \tau) \circ \rho_1 = (\rho_2 \cdot \Phi) \circ \rho$, we define the associated representation map as

$$\mu = (I_2 \cdot \varepsilon) \circ (\rho \cdot \Psi)$$

$$\theta = (\Phi_2 \cdot \varepsilon) \circ (\tau \cdot \Psi)$$

The representation map condition follows easily from the condition for the comorphism modification and naturality of ρ_2:



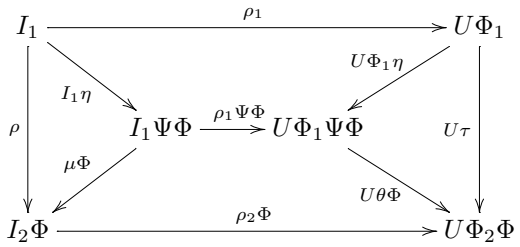
Vice versa, given a a representation map $((\Psi, \mu), \theta)\colon (\Phi_2, \rho_2) \longrightarrow (\Phi_1, \rho_1)$, i.e. $\mu\colon I_1 \circ \Psi \longrightarrow I_2$ and $\theta\colon \Phi_1 \circ \Psi \longrightarrow \Phi_2$ such that $(\rho_1 \cdot \Psi) \circ (U \cdot \theta) = \rho_2 \circ \mu$, define the comorphism from I_1 to I_2 by

$$\rho = (\mu \cdot \Phi) \circ (I_1 \cdot \eta)$$

and the comorphism modification $\tau\colon (\Phi_1, \rho_1) \longrightarrow (\Phi_2, \rho_2) \circ (\Phi, \rho)$ as

$$\tau = (\theta \cdot \Phi) \circ (\Phi_1 \cdot \eta)$$

The condition for the comorphism modification follows easily from the representation map condition and naturality of ρ_1:



By the adjunction laws, the two constructions are inverses of each other.

□

6.13 Bibliographical Notes

The early publications about institutions-based approaches to heterogeneous specification include [AC94], [KST94], [OP02] and [BCL96, CBL99]. Diaconescu [Dia98] and Tarlecki [Tar00] were the first to systematically study the semantics of heterogeneous specification languages, including amalgamation and exactness properties. Tarlecki [Tar00] has been the first noting that different types of (co)morphisms [Tar96] are needed for heterogeneous specification. This has been elaborated by the present author in [Mos03].

A breakthrough in the semantics of heterogeneous specification is the invention of Grothendieck institutions by Diaconescu [Dia02] as a generalization of Grothendieck categories.[9] The latter are obtained by the flattening construction on indexed categories introduced by Grothendieck [Gro63]. Actually, Diaconescu first had invented so-called *extra theory morphisms* [Dia98] in order to provide a semantics for CafeOBJ [DF02], and then later found that his results can be obtained within the framework of Grothendieck institutions in a much simpler way. The present author has dualized the Grothendieck construction to the comorphism case [Mos02a] (and thereby simplified the amalgamation results). The Grothendieck construction for entailment systems has already been presented even earlier than that for institutions: in an unpublished report by Dimitrakos, Bicarregui and Maibaum [DBM99].

Tarlecki [Tar04] argues that Grothendieck institutions are not necessary for heterogeneous specification; instead, he defines syntax and semantics and proof rules for heterogeneous specification directly w.r.t. a given set of institutions and (co)morphisms. However, the proof rules only cover the question whether a heterogeneous specification entails a sentence, but not the question whether a heterogeneous specification entails (or refines to) another one. The latter question is inevitably intertwined with amalgamation and interpolation properties of (and conservativity checks in) the Grothendieck institution. Although Tarlecki [Tar00] studies some heterogeneous amalgamation properties without constructing a Grothendieck institution, we think that a study of all the needed properties is much easier when working with the Grothendieck construction. Moreover, many proof rules work independently of whether the involved signature morphisms are homogeneous or heterogeneous. Imagine e.g. how complicated the proof system of [Bor02] for proving refinements gets if homogeneous and heterogeneous signature morphisms have to be distinguished. A similar remark holds for the proof calculus for development graphs.

The notion of heterogeneous development graph has been introduced by the present author in [Mos02b]. Our completeness theorem for the proof calculus for development graphs has been published in [MAH01] for the homogeneous case. The heterogeneous case (published in [Mos02a]) mainly adds one complication, caused by the desired restriction of conservativity checks to intra-logic signature morphisms. The main advantage of this calculus over the one for structured specifications introduced in [Bor02] (see also Sect. 5.3) is the weaker set of assumptions for completeness of the calculus: basically, we need completeness of the underlying entailment system plus existence of quasi-semi-exactness (cf. Def. 2.8). The latter property is much weaker than the assumptions in [Bor02] (Craig interpolation, conjunction and implication).

The use of 2-cells between institutions (co)morphisms for heterogeneous specification has its origin in Tarlecki's representation maps [Tar96]. The notion of 2-indexed (co)institution has been introduced by the present author in [Mos02a].

Diaconescu's forthcoming book [Dia] will cover a good portion of the theory of Grothendieck institutions. See also the FLIRTS bibliography (`http://www.tzi.de/flirts/flirtslibrary.html`) for literature about heterogeneous specification.

[9] Indeed, the present author has re-invented Grothendieck institutions in Spring 2001, but then found out that Diaconescu already had written a paper about them.

Chapter 7

The Heterogeneous Tool Set (HETS)

The Heterogeneous Tool Set (HETS) implements the theory developed so far. It is the main analysis tool for the specification language heterogeneous CASL. Heterogeneous CASL (HETCASL) combines the specification language CASL with CASL extensions and sublanguages, as well as completely different logics and even programming languages such as Haskell. HETCASL extends the structuring mechanisms of CASL, while HETCASL *basic specifications* are unstructured specifications or modules written in a specific logic (possibly completely different from CASL). Hence, only syntax and semantics the logic for specification-in-the-small has to be adapted individually, while the concepts in-the-large can be used for any logic. The graph of currently supported logics is shown in Fig. 7.1, and the degree of support by HETS in Fig. 7.2. *It should be stressed that the name "*HETCASL*" only refers to* CASL*'s structuring constructs. The individual logics used in connection with* HETCASL *and* HETS *can be completely orthogonal to* CASL.

CASL provides institution-independent structuring constructs for writing specifications-in-the-large. With *heterogeneous structured specifications* in HETCASL, it is possible not only to combine and rename specifications, hide parts thereof (as in the language of CASL structured specifications), but also translate them to other logics. Like in CASL, also HETCASL provides *architectural specifications* that prescribe the structure of implementations, and *specification libraries*, which are collections of named structured and architectural specifications. A detailed language summary of HETCASL is given in Appendix A.

Actually, the capabilites of HETS now go even slightly beyond HETCASL, since HETS also supports Haskell's module system as a structuring language. This enables HETS to directly read in Haskell programs. Moreover, support of further structuring languages is planned. The central device gluing together the different structuring languages is the formalism of *development graphs*, that have been introduced in Sect. 5.4 and have been extended for heterogeneous specification in Chap. 6. Development graphs have been used for industrial-scale applications with hundreds of specifications. They serve as a core structuring language. Tools such as MAYA [AHMS02] provide a management of proofs, based on the formalism of development graphs. The goal of HETS is to make MAYA heterogeneous.[1]

The architecture of HETS is depicted in Fig. 7.3. HETS has an abstract interface corresponding to concept of logic in Haskell. Of course, since model theory is not directly implementable, the interface mainly is concerned with the entailment system of the logic. However, also some amalgamability tests (refering to the model theory of the institution) need to be implemented.

HETS implements this by providing a type class `Logic`. `Logic` is a multiparameter type classes with functional dependencies [PJM97]. Such a type class can be thought of as a formal parameter

[1] Indeed, MAYA supports some kind of ad-hoc heterogeneity: while every theory is translated to higher-order logic, there are special nodes indicating the original logic of a specification.

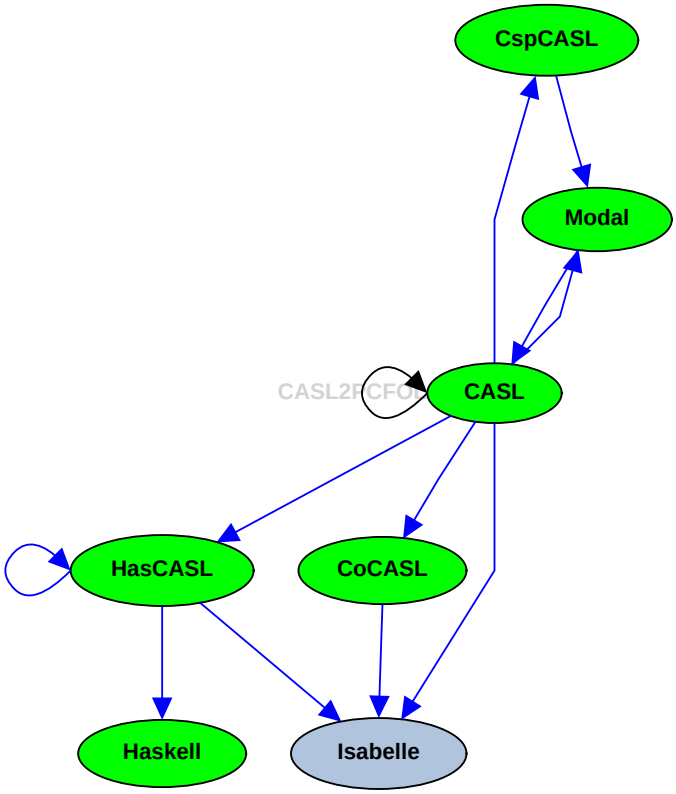


Figure 7.1: Graph of logics currently supported by Hets.

Language	Parser	Static Analysis	Prover
Casl	x	x	(x)
CoCasl	x	x	(x)
ModalCasl	x	x	-
HasCasl	x	(x)	(x)
Haskell	x	x	-
Csp-Casl	(x)	-	-
Structured specifications	x	x	(x)
Architectural specifications	x	x	-

Figure 7.2: Current degree of Hets support for the different languages.

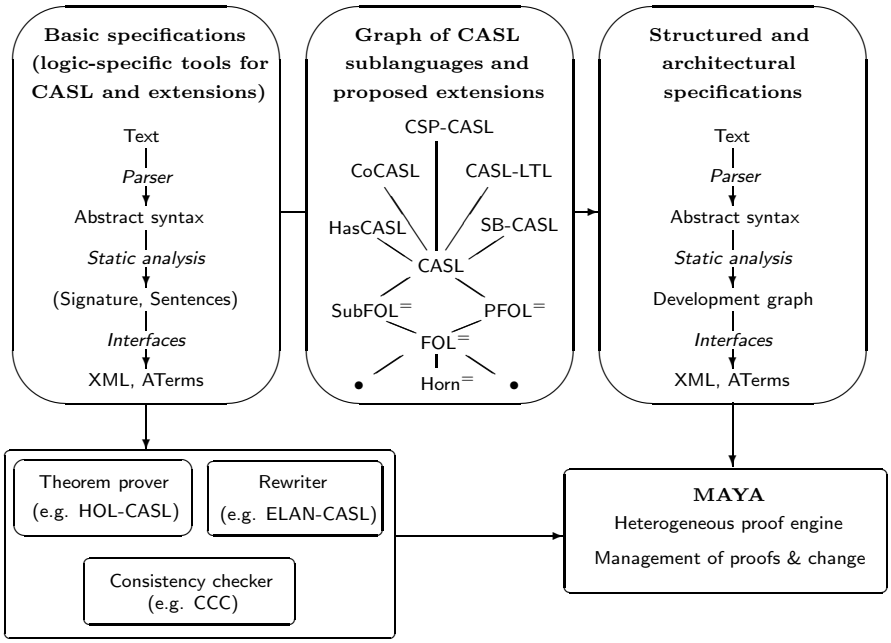


Figure 7.3: Architecture of the heterogeneous tool set

signature. `Logic` contains types for signatures, signature morphisms, sentences, abstract syntax of basic specifications etc., and functions for parsing, printing, static analysis, and proving. Based on this abstract interface, we have implemented heterogeneous tools for parsing and static analysis of heterogeneous CASL (using a semantics similar to the verification semantics in Sect. 6.11 above)[2]. The heterogeneous tools of course needs to call the logic-specific tools whenever a basic specification is encountered. The heterogeneous parser yields an abstract syntax tree, which is fed into the heterogeneous static semantic analysis. The latter in turn yields a development graph over the Grothendieck institution, which is then the basis for heterogeneous proofs. Part of the proof calculus for heterogeneous development graphs has been implemented, the support for hiding is currently being completed.

Technically, heterogeneity is realized as follows. On top of the type class `Logic`, an existential datatype is constructed. Usually, existential types are used to realize e.g. heterogeneous lists, where each element may have a different type. We use lists of (components of) institutions and comorphisms instead. This leads to an implementation of the Grothendieck institution over an indexed coinstitution.

We have instantiated this general framework with institution-specific analysis tools for CASL, HASCASL, Haskell, COCASL, CSP-CASL and MODALCASL. Future work will interface existing theorem proving tools with specific institutions in HETS. We already have implemented an experimental interface to the theorem prover Isabelle, which is realized as an own logic within HETS.

The Heterogeneous Tool Set is available at `www.tzi.de/cofi/hets`. There, a user guide is available as well. A brief introduction into HETS is given in [BM04].

In the sequel, we will describe a toy heterogeneous language and a toy Haskell program analysing it (with providing full code in detail). The central ideas of HETS can be grasped by studying this toy tool. Later on, an overview over the real HETS system is given.

[2]As noted above, also Haskell's module system can be parsed and analysed.

```
SPEC ::= logic ID SPEC
       | BASIC-SPEC
       | SPEC then SPEC
       | SPEC with SYMBOL-MAPPING
       | SPEC with logic ID
```

Figure 7.4: Abstract syntax of a simple subset of the heterogeneous specification language. `BASIC-SPEC` and `SYMBOL-MAPPING` have a logic specific syntax, while `ID` stands for some form of identifiers.

7.1 Genericity Versus Heterogeneity

In [MK02], we outlined how the static analysis of CASL structured specifications can be turned into a generic program which is parameterized over a static analysis for an arbitrary logic. This has been realized as a Standard ML functor. If one now wants to implement an analysis tool for a new specification language, one can easily adopt the CASL structuring language for specification-in-the-large and only needs to implement an analysis tool for specifications-in-the-small based on the logic underlying the specification language. This language-specific tool has to fit with the signature of the Standard ML functor. Then, the functor can be instantiated with the language-specific tool. Along similar lines, various generic tools have been realised as Standard ML functors, e.g. the IsaWin system realizing a graphical user interface for LCF-style theorem provers [LW99, LTKKB99].

However, this approach has its strong limitations when moving from *genericity* to *heterogeneity*, as it is the case for the heterogeneous specification language proposed in [Mos03]. Here, we not only want to use structuring constructs that are generic over the underlying logic, but we also want to write specifications that *use several logics simultaneously*, in one and the same specification, while using logic translations for relating the logics (cf. the abstract syntax in Fig. 7.4). Of course, one can use several instantiations of standard ML functors in parallel. However, there is no way to deal simultaneously with several of these instantiations in a uniform manner (in particular, if the number of forthcoming instantiations is not known). It seems that the type system of Standard ML cannot handle this in an elegant way. We have therefore decided to move to Haskell (and its extensions provided by the Glasgow Haskell compiler [Uni]), which does provide a richer type system (although functors are missing).

7.2 The Type Class Logic

What is the abstract interface for a logic? While in Standard ML, we have collected the types and functions implementing a logic into a signature of a functor, in Haskell, all this is collected into a *multiparameter type class* [PJM97]. Now Haskell needs to be able to infer the correct type class instance from any instance of any of the functions in the type class. Since hardly any function has all the type class parameters occurring in its argument type, we need to specify *functional dependencies*. The easiest way to do this for an arbitrary interface signature is to add a new dummy type parameter `id` carrying the identity (in the sense of "personality", and not in the sense of $\lambda x \to x$) of the instance of the multiparameter type class. This new type parameter is usually instantiated as a singleton, and added as an extra argument type to each function helping to determine the correct instance.

An example of this method is given in Figs. 7.5 and 7.6. Here, the basic ingredients of a logic are formalized.

This leads to the following type classes:

- The type class `Language` merely carries the identity of the language (consisting of the type parameter `id` as indicated above, and the name of the language).
- The type class `Category` is for the category of signatures and signature morphisms (although

```
module Logic (module Logic, module Dynamic) where

import Dynamic
import Parsec

-- the identity of a language
class (Show id, Typeable id) => Language id where
    language_name :: id -> String
    language_name i = show i

-- categories, needed for signatures and signature morphisms
class (Language id, Eq sign, Show sign, Eq morphism) =>
      Category id sign morphism | id -> sign, id -> morphism where
         identity :: id -> sign -> morphism
         o :: id -> morphism -> morphism -> Maybe morphism
         dom, cod :: id -> morphism -> sign

-- abstract syntax, parsing and printing
class (Language id, Show basic_spec, Eq basic_spec, Typeable basic_spec,
                    Show symbol_mapping, Eq symbol_mapping, Typeable symbol_mapping,
                    Show sentence, Eq sentence) =>
      Syntax id sign sentence basic_spec symbol_mapping
        | id -> sign, id -> basic_spec, id -> symbol_mapping, id -> sentence where
         parse_basic_spec :: forall st . id ->  CharParser st basic_spec
         parse_symbol_mapping :: forall st . id -> CharParser st symbol_mapping
         parse_sentence  :: id -> sign -> String -> sentence
```

Figure 7.5: The basic ingredients of a logic — syntactic part

it could be re-used for other categories as well). Except from the additional type parameter `id`, this has be directly obtained from the mathematical definition of a category.

- Usually signatures and signature morphisms (as well as sentences) of a logic are more semantic entities of a condensed nature, while the syntax of a specification language built upon that logic provides a more verbose and user-friendly input syntax. We therefore introduce types `basic_spec` and `symbol_map`, serving as abstract syntaxes for basic specifications and symbol maps, and parsers and printers for these. All this is collected in the type class `Syntax`.

- Now a basic specification in the specification language corresponds to a *theory* (signature plus set of sentences) at the level of the logic. The type class `StaticAnalysis` contains a static analysis function for basic specifications that delivers such a theory. Note that a signature ("local environment") flows into the analysis; this corresponds to imported parts of a specification.

- The data type `Proof_Status` provides a simple abstract interface for theorem provers that prove sentences in a given theory. A prover returns whether the sentence could be proved or not, or remains open.

- Finally, the type class `Logic` adds, besides a prover, also the sentence functor. Since we do not want to complicate things by using dependent types, we have just collected all sentence sets into one type, the objects of which of course should also carry the signature information. Hence, due to the possibility of a signature mismatch, sentence translation along a signature morphism becomes a partial operation. We introduce sentence translation only at this late stage (compared with the mathematical definition of entailment system), since it is practically used only in the management of proofs.

The type class `Logic` also is depicted in the the upper left box in Fig. 7.3 (we also have included some interfaces to a compact output format such as XML or ATerms there, which for brevity are omitted in the shown Haskell code).

Based on the type class `Logic`, our goal is now to implement *heterogeneous* tools for parsing and static analysis of the heterogeneous language, as well as theorem proving tools for proving in multi-logic specifications. Together, these tools will form the *heterogeneous tool set* with architecture is depicted in Fig. 7.3. Of course, we will present only some very simplified version here; nevertheless, the shown pieces of code together form a complete set of Haskell modules.

7.3 Implementing the Grothendieck Logic

The Grothendieck logic (see Sect. 6.1) can be implemented as a bunch of *existential* types over the type class `Logic`, see Fig. 7.7. At this point, we can see the achievement compared to standard ML functors: we can now instantiate the type class logic with different logics and work with heterogeneous lists of logics, which is not possible in Standard ML. A disadvantage compared with Standard ML is that we cannot name signatures — we always have to list the whole list of type parameters, which is a bit tedious when often repeated.

We also have included some functions doing coercions among dynamic types. This will be explained below.

The abstract syntax of heterogeneous specifications is given in Fig. 7.8. A specification either consists of some basic specification in some logic, or a union of specifications, or a translation of a specification. Translations can be along symbol maps, or along logic translations. The datatype for environments follows roughly the same structure; however, environments are intended to carry fully statically checked informations, based on theories and their translations. Additionally, at each point, a theory corresponding to the flattened environment (i.e. with all translations performed and extensions united) is stored as well.

```
-- module Logic continued

-- a theory consists of a signature and a set of sentences
type Theory sign sentence = (sign,[sentence])

-- static analysis
class (Syntax id sign sentence basic_spec symbol_mapping,
       Typeable sign, Typeable morphism, Typeable sentence)  =>
      StaticAnalysis id sign morphism sentence basic_spec symbol_mapping
        | id -> morphism where
         basic_analysis ::
            id -> sign -> basic_spec -> Maybe (Theory sign sentence)
                -- the input signature contains imported stuff
         stat_symbol_mapping ::
            id -> symbol_mapping -> sign -> Maybe morphism

-- Proofs
data Proof_status = Open | Disproved | Proved deriving Show

-- logic (entailment system)
class (Category id sign morphism,
       StaticAnalysis id sign morphism sentence basic_spec symbol_mapping) =>
      Logic id sign morphism sentence basic_spec symbol_mapping
      where  empty_signature :: id -> sign
             empty_theory :: id -> Theory sign sentence
             empty_theory i = (empty_signature i,[])
             map_sentence :: id -> morphism -> sentence -> Maybe sentence
             inv_map_sentence :: id -> morphism -> sentence -> Maybe sentence
             prover :: id -> Theory sign sentence -- theory that shall be assumed
                       -> sentence                -- the proof goal
                       -> IO Proof_status

-- logic translations
data (Logic id1 s1 m1 sen1 b1 sy1, Logic id2 s2 m2 sen2 b2 sy2) =>
     Logic_translation id1 s1 m1 sen1 b1 sy1 id2 s2 m2 sen2 b2 sy2 =
     Logic_translation { source :: id1,
                         target :: id2,
                         tr_sign :: s1 -> s2,
                         tr_mor :: m1 -> m2,
                         tr_sen :: s1 -> sen1 -> Maybe sen2,
                         inv_tr_sen :: s1 -> sen2 -> Maybe sen1 }
```

Figure 7.6: The basic ingredients of a logic — semantic part

```
module Grothendieck (module Logic, module Grothendieck) where

import Logic

data AnyLogic =
        forall id s m sen b sy .
        Logic id s m sen b sy  =>
        G_logic id

data AnyTranslation =
        forall id1 s1 m1 sen1 b1 sy1 id2 s2 m2 sen2 b2 sy2 .
        (Logic id1 s1 m1 sen1 b1 sy1, Logic id2 s2 m2 sen2 b2 sy2) =>
        G_LTR (Logic_translation id1 s1 m1 sen1 b1 sy1 id2 s2 m2 sen2 b2 sy2)

type LogicGraph = ([(String,AnyLogic)],[(String,AnyTranslation)])

data G_basic_spec =
        forall id s m sen b sy .
        Logic id s m sen b sy  =>
        G_basic_spec id b

data G_symbol_mapping_list =
        forall id s m sen b sy .
        Logic id s m sen b sy  =>
        G_symbol_mapping_list id sy

data G_sentence =
        forall id s m sen b sy .
        Logic id s m sen b sy  =>
        G_sentence id sen

data G_theory =
        forall id s m sen b sy .
        Logic id s m sen b sy  =>
        G_theory id (Theory s sen)

data G_morphism =
        forall id s m sen b sy .
        Logic id s m sen b sy  =>
        G_morphism id m

-- auxiliary functions for conversion between different logics
coerce :: (Typeable a, Typeable b) => a -> Maybe b
coerce = fromDynamic . toDyn

coerce1 :: (Typeable a, Typeable b) => a -> b
coerce1 = Maybe.fromJust . coerce
```

Figure 7.7: The Grothendieck logic

```
module Structured (module Structured, module Grothendieck) where

import Grothendieck

data SPEC = Basic_spec G_basic_spec     -- unstructured specifications
          | Intra_Translation SPEC G_symbol_mapping_list  -- renaming within a logic
          | Inter_Translation SPEC AnyTranslation    -- translation between logics
          | Extension SPEC SPEC            -- hierarchical extension or union
          deriving Show

data Env = Basic_env G_theory
         | Intra_Translation_env G_theory Env G_morphism
         | Inter_Translation_env G_theory Env AnyTranslation
         | Extension_env G_theory Env Env
```

Figure 7.8: Abstract syntax and environments for heterogeneous specifications

7.4 Heterogeneous Parsing

The heterogeneous parser (cf. Fig. 7.9) transforms a string to an abstract syntax tree and is additionally parameterized over an arbitrary logic graph. It implements the grammar from Fig. 7.4, using the `Parsec` combinator parser and its expression parser `ParsecExpr` [LM]. Logic and translation names are looked up in the logic graph — this is necessary to be able to choose the correct parser for basic specifications. Indeed, the parser has a state that carries the current logic, and which is updated if an explicit specification of the logic is given, or if a logic translation is encountered (in the latter case, the state is set to the target logic of the translation). With this, it is possible to parse basic specifications by just using the logic-specific parser of the current logic as obtained form the state.

The parsing of logic translations is based on dynamic types and the function `coerce` coercing values between any two of these. This is necessary in order to be able to check whether the logic of the translated specification coincides with the logic of the logic translation.

7.5 Heterogeneous Static Analysis

The static analysis, given in Fig. 7.10, is based on the static analysis of basic specifications, and transforms an abstract syntax tree to an environment. Starting with an empty theory, it successively extends (using the static analysis of basic specifications) and/or translates (along the intra- and inter-logic translations) the theory, while simultaneously constructing an environment. The initial empty theory has to be computed by an auxiliary function, since the logic of this theory is not known in advance.

Within the static analysis of basic specifications and of (intra- and inter-logic) translations, dynamic types and the function `coerce` are heavily used. This is because we need to relate different instances of the existential types of the module `Grothendieck`, which fit together because of the way the parser works - but this is not known by the type system. In order to store this information in the types, we would need dependent types, which however are available only in experimental Haskell extensions. We have therefore chosen to use dynamic types, which works quite well. One limitation of the type system of the Glasgow Haskell compiler is that we often cannot annotate the result of of a call of `coerce` with its type. This is because this type here typically is computed using functional dependencies, and there is no notation that allows one to extract such a type from given types.

```
module Parser where

import Parsec
import ParsecExpr
import Structured

hetParse :: LogicGraph -> String -> SPEC
hetParse (logics@((_,defaultLogic):_), translations) input =
  case (runParser spec defaultLogic "" input) of
         Left err -> error ("parse error at "++show err)
         Right x  -> x
  where
  spec    :: CharParser AnyLogic SPEC
  spec    = buildExpressionParser table basic
          <?> "SPEC"
  basic = do { G_logic id <- getState;
               b <- parse_basic_spec id;
               return (Basic_spec (G_basic_spec id b))}
  table = [[Prefix (do {string "logic"; spaces;
                        name <- many1 alphaNum;
                        setState (case lookup name logics of
                          Nothing -> error ("logic "++name++" unknown")
                          Just id -> id);
                        spaces; return (\x->x) } )],
           [Postfix (do
             string "with"; spaces;
              do string "logic"; spaces
                 name <- many1 alphaNum
                 G_logic (id::src) <- getState
                 case lookup name translations of
                   Nothing -> error ("translation "++name++" unknown")
                   Just (G_LTR tr) ->
                     case coerce(source tr)::Maybe src of
                       Nothing -> error ("translation type mismatch")
                       Just _ -> do
                         setState (G_logic (target tr))
                         return (\sp -> Inter_Translation sp (G_LTR tr))
              <|> do G_logic id <- getState
                     sy <- parse_symbol_mapping id
                     spaces
                     return (\sp -> Intra_Translation sp (G_symbol_mapping_list id sy))
              )],
           [Infix (do{string "then"; spaces; return Extension}) AssocLeft]
          ]
```

Figure 7.9: The heterogeneous parser

```
module StaticAnalysis where

import Structured

staticAnalysis :: SPEC -> Maybe (Env, G_theory)
staticAnalysis sp = staticAna1 initial_theory sp
  where
  initial_theory = get_initial sp
  get_initial sp = case sp of
    Basic_spec (G_basic_spec logic _) ->
      G_theory logic (empty_theory logic)
    Intra_Translation sp _ -> get_initial sp
    Inter_Translation sp _ -> get_initial sp
    Extension sp _ -> get_initial sp
  staticAna1 :: G_theory -> SPEC -> Maybe (Env, G_theory)
  staticAna1 th@(G_theory id (sig,ax)) sp =
    case sp of
      Basic_spec (G_basic_spec _ b) ->
        do b' <- coerce b
           (sig1,ax1) <- basic_analysis id sig b'
           let th' = G_theory id (sig1,ax1++ax)
           return (Basic_env th',th')
      Intra_Translation sp (G_symbol_mapping_list  _ symap) ->
        do (env,G_theory id1 (sig1,ax1)) <- staticAna1 th sp
           symap' <- coerce symap
           mor <-  stat_symbol_mapping id1 symap' sig1
           tr_ax <- sequence (map (map_sentence id1 mor) ax1)
           let th' = G_theory id1 (cod id1 mor,tr_ax)
           let env' = Intra_Translation_env th' env (G_morphism id1 mor)
           return (env',th')
      Inter_Translation sp (G_LTR tr) ->
        do (env,G_theory id1 (sig1,ax1)) <- staticAna1 th sp
           let id_tar = target tr
           sig2 <- coerce sig1
           ax2 <- coerce ax1
           let tr_sig = tr_sign tr sig2
           tr_ax <- sequence (map (tr_sen tr sig2) ax2)
           let th' = G_theory id_tar (tr_sig,tr_ax)
           let env' = Inter_Translation_env th' env (G_LTR tr)
           return (env',th')
      Extension sp1 sp2 ->
        do (env1,th1) <- staticAna1 th sp1
           (env2,th2) <- staticAna1 th1 sp2
           return (Extension_env th2 env1 env2,th2)
```

Figure 7.10: The heterogeneous static analysis

```
module Proof where

import Structured
import Parser
import StaticAnalysis

prove :: LogicGraph -> Bool -> String -> [String] -> IO()
prove logicGraph flat spec raw_goals = do
   if flat then proveFlat th (getGoals th)
           else proveStruct th env (getGoals th)
   where
   as = hetParse logicGraph spec
   Just (env,th) = staticAnalysis as
   getGoals (G_theory id (sig,ax)) =
     map (G_sentence id . parse_sentence id sig) raw_goals

   proveFlat th goals = do
     res <- sequence (map (proveFlat1 th) goals)
     putStrLn (show res)

   proveFlat1 (G_theory id (sig,ax)) (G_sentence _ goal) =
     prover id (sig,ax) (coerce1 goal)

   proveStruct (G_theory id (sig,ax)) env goals = do
     res <- sequence (map (prove1 env) goals)
     putStrLn (show res)
     where
     prove1 :: Env -> G_sentence -> IO Proof_status
     prove1 env g@(G_sentence id goal) = case env of
       Basic_env (G_theory id' (sig,ax)) ->
         prover id' (sig,ax) (coerce1 goal)
       Intra_Translation_env th env' (G_morphism id' mor) ->
         let goal' = coerce1 goal in
         case inv_map_sentence id' mor goal' of
           Just goal'' -> prove1 env' (G_sentence id' goal'')
           Nothing -> proveFlat1 th g
       Inter_Translation_env th env' (G_LTR tr) ->
         prove_aux th
         where
         prove_aux (G_theory _ (sig,_)) =
          case inv_tr_sen tr (coerce1 sig) (coerce1 goal) of
           Just goal'' -> prove1 env' (G_sentence (source tr) goal'')
           Nothing -> proveFlat1 th g
       Extension_env _ env1 env2 -> do
         res <- prove1 env1 g
         case res of
           Proved -> return Proved
           _ -> prove1 env2 g
```

Figure 7.11: Homogeneous and heterogeneous proofs

7.6 Heterogeneous Proofs

Finally, the module `Proof` implements homogeneous as well as heterogeneous proofs. First, the heterogeneous specification is parsed and statically analysed, and then, the proof goals are parsed in the logic and signature of the specification. For a homogeneous (=flat) proof, for each proof goal just the prover for the logic of the specification is called. In contrast, for a heterogeneous proof, each proof goal is translated back along any intra- and inter-logic translations of the environment of the specification, until this is no longer possible, and then, it is proved in some flattened theory. With this, it is possible to do truly *heterogeneous* proofs: each proof goal is proved within a logic that is as minimal (w.r.t. the translations in the logic graph) as possible. In this way, it is possible to exploit the strengths of provers that are specialized towards particular logics.

7.7 Overview of HETS module structure

We now give a brief overview of the module structure of HETS, which is of course much more complicated than the structured of the toy modules presented above. The input language of HETS is the heterogeneous language summarized in Appendix A is of course also more complicated than the above toy language.

The HETS modules are grouped using hierarchical modules (where modules can be grouped into folders); we here only discuss the top view on this hierarchy.

The folder `Logic` contains the infrastructure needed for institution independence. The module `Logic.Logic` contains all the type classes for interfacing institutions mentioned above, including the type class `Logic`. The module `Logic.Prover` is for the interface to theorem provers, rewriters, consistency checkers, model checkers. The data types `Proof_Status` and `Prover` provides the interface to provers. In case of a successful proof, also the list of axioms that have been used in the proof can be returned. This will be crucial for an efficient management of change, see [AHMS00, AM02].

Module `Logic.Comorphism` provides type classes for the various kinds of mappings between institution (which have been introduced in Chap. 2), and module `Logic.Grothendieck` realizes the Grothendieck construction from Sect. 6.1 and also contains a type *LogicGraph*. This is complemented by folders working in the heterogeneous level — the code in modules in these folders is parameterized over an arbitrary but fixed logic graph. The folder `Syntax` provides abstract syntax and parsing of heterogeneous structured specifications. `Static` is for the static analysis (based on the verification static semantics given in Sect. 5.5 and 6.11). `Static.DevGraph` contains the data structures for heterogeneous development graphs. Finally, the folder `Proofs` contains an implementation of the proof calculus for heterogeneous development graphs as described in Sect. 6.4.

The folders `CASL`, `CoCASL`, `HasCASL`, `Haskell`, `CspCASL`, `Modal`, `Isabelle` contain different instances of the type class `Logic` of the module `Logic.Logic`. These instances always are contained in a module named `Logic_xxx`, where `xxx` is the name of the language at hand. Since the integration of a new logic into HETS requires writing a new instantiation of the type class `Logic`, it is advisable to consult the module `Logic_xxx` (and the modules imported there) for some logic that is in some sense similar to the new logic to be integrated. In particular, we have implemented the CASL logic in such a way that much of the folder `CASL` can be re-used for CASL extensions as well; this is achieved via "holes" (realized via polymorphic variables) in the types for signatures, morphisms, abstract syntax etc. This eases integration of CASL extensions and keeps the effort quite moderate.

The folder `Comorphisms` contains various comorphisms and other translations that constitute the logic graph. Note that these modules can be compiled independently of the logic independent heterogeneous modules listed above. The module `Comorphisms.LogicList` assembles all the logics into one (heterogeneous) list, while `Comorphism.LogicGraph` builds up the logic graph, i.e. it assembles all the (co)morphisms among the logics, and also specifies which ones are standard inclusions. This module also provides a partial union for logics, which is crucial for the static analysis of unions of specifications (which may occur explicitly or implicitly).

Last but not least, there are general purpose folders: `ATC` for conversion from and to the ATerm [BJKO00] format — most of the modules have been automatically created using `DriFT` from the

CASL	11.500
COCASL	1.300
CSP-CASL	1.700
MODALCASL	1.000
HASCASL	11.000
Haskell	3.500
Isabelle	1.000
Comorphisms between logics	3.000
Logic specific-code in total	34.000
Interface for Logics, Heterogeneous Tools	11.500
Common utilities, GUI	12.500
Logic independent code in total	24.000
UniForM Workbench	110.000
Other third party modules	15.000
Third party modules in total	125.000

Figure 7.12: Lines of Haskell code in the Heterogeneous Tool Set.

`utils` folder. The latter also contains a module `inlineAxioms` that can be used to write the axioms for theoroidal comorphisms in a concise way, namely in the input syntax of the respective target logic (the identifiers will turn into Haskell variables and can hence be used for easily producing instances of axiom schemes). The folder `Common` contains general purpose libraries, e.g. for sets, maps and relations, and for parsing and pretty printing. The command line interface is contained in `hetcats`, the graphical interface in `GUI`. A more detailed descriptions of the modules and their contents, including an index of all data types and functions, can be found via the HETS web page (`http://www.tzi.de/cofi/hets`). An overview of the numbers of lines of code is given in Fig. 7.12.

The reader is encouraged to have a look at the selected code parts of HETS in Appendix B. These code parts close the gap between the toy language presented above and the real HETS system.

Chapter 8

Conclusion

The central objective of this work is to provide a solid semantic foundation for heterogeneous specification of complex software systems and simultaneously show that this leads to a framework that can be used in practice, also supported by tools. We hope that the reader has got the general picture how this can be done, even if it is clear that there is some (not too large) gap between the present work and realistic formal heterogeneous developments.

Heterogeneous specification adds a certain degree of (meta-level) complexity to the complexity already inherent in the various specification frameworks. However, we think that in the end this extra meta-level complexity pays off: indeed, it is implicitly already there in the various structuring languages and translations between formalism that are already in use. Heterogeneous specification just studies this more systematically and thoroughly, in order to provide the foundation at this meta-level once and for all, instead of re-inventing it numerous times in an ad-hoc manner.

The most promising way of giving a semantics to such specifications has turned out to be the formalization of logics as institutions, logic translations as institution comorphisms, and graphs of logics as indexed coinstitutions. The semantics of a heterogeneous specification then lives in the comorphism-based Grothendieck institution. Other types of translation between institutions (whose need is motivated by various examples) can be integrated as well, using spans of comorphisms. The advantage of the comorphism-based Grothendieck construction is its nice interaction with amalgamation properties. The latter are needed for obtaining structured and heterogeneous proof calculi.

Of course, a central prerequisite to make this work in practice is to show that a number of logics of varying nature and application domain can indeed be formalized as institutions in such a way that smooth translations are possible and an initial logic graph (which of course can be extended later on) emerges.

The central device for managing specifications and structured proofs are development graphs, which already have successfully been applied in industrial context. The central achievement of the present work is to extend development graphs with a hiding operation, consistency and conservativity considerations, and of course to make them heterogeneous. The central point here has been to obtain a soundness and completeness result for heterogeneous development graphs that can be applied to realistic logic graphs, such as the one studied here. Indeed, the conditions for soundness and completeness of our calculus are related to various forms of exactness and weak amalgamation conditions. These conditions are so mild that they hold in typical practical examples; in particular, they are considerably weaker than both the exactness conditions for Grothendieck institutions in [Dia02] and the Craig interpolation property needed for completeness of calculi for structured specification [Bor02]. We also have demonstrated that heterogeneous bridges (and hence truly heterogeneous proofs), as introduced in [BCL96, CBL99] in an *ad-hoc* manner, can be obtained in the semantic framework of Grothendieck institutions. Instead of using a global encoding into some "universal" logic, truly heterogeneous proofs have the advantage to better exploit specialized tool support.

Tool support for heterogeneous specifications and development graphs is provided in form of

the Heterogeneous Tool Set (HETS). The latter provides an abstract programming interface for the implementable part of institutions and comorphisms. This serves as a basis for heterogeneous analysis and proof tools that are based on corresponding tools for the individual logics, provided for an arbitrary but fixed graph of logics and logic transformations. Hence, we achieve not only genericity, but also true heterogeneity, involving different logics and their translations at the same time. While genericity has been achieved using multi-parameter type classes with functional dependencies (roughly corresponding to Standard ML functors), heterogeneity makes essential use of existential and dynamic types. We have provided a self-contained set of toy Haskell modules that show how the tool is implemented, as well as a realistic Haskell program of 60.000 lines. The effort needed to integrate a new logic remains moderate. HETS can thus seen as a framework for *integrating languages and tools*, aiming at software development in a heterogeneous setting, but also usable for mediating between different formal ontologies.

Compared with logic combination [MTP98, SSCM00, CMRS02, CGR03], heterogeneous specification has only weaker forms of feature interaction. Logic combination provides feature interaction between different logical connectives, quantifiers, modalities, and so on. By contrast, heterogeneous specification puts the involved logics side by side, with the only feature interaction provided by the logic translations (formalized as comorphisms).

The problematic point with logic combination is th resulting proliferation of even more logics, all of which need their own proof support (although calculi sometimes can be combined as well, they still need to be implemented and optimized individually). Here, heterogeneous specification and heterogeneous proofs are more flexible: they support better re-use of existing (sometimes highly specialized) proof tools for individual logic. Last but not least, heterogeneous specification is more widely applicable than logic combination: one needs just one metaformalism for all logics, while meta-frameworks for logic combination usually have to be fine-tuned for each new type of logic they are applied to.

Further comparison with related work is given in the bibliographical sections of each chapter.

8.1 Future Work

Future work will in the first place consist of the development of realistic case studies[1] of heterogeneous specifications with HETS, along with the further enhancement of HETS in order to provide the needed support for such case studies. The goal is to come up with a convincing reference application that will lead to more direct insight of the usefulness of heterogeneous specification and the heterogeneous tool set, both in academia and industry. We plan to extend HETS to an industrial-strength tool for formal software development using heterogeneous specifications. The approach presented here therefore needs to be extended in the directions described below.

As a general line, we are of course also interested in the integration of more logics, translations, provers and consistency and model checkers into HETS. If your favorite language or prover is not yet integrated, please contact us[2], or even better, join the HETS development team and integrate your language and/or prover. In Bremen, several people are working on this, but we need resources from other sites as well in order to turn HETS into a success.

Moreover, also, the general interfaces of HETS, like the general command-line interface and the graphical interface, as well as the interface for storing states of the development, can be enhanced.

Of course, also the heterogeneous language and the theory behind it need further development, see e.g. Sect. 8.1.5. However, given the body of theory developed in this thesis, we plan to concentrate the energies on turning HETS into a success.

8.1.1 Management Of Change

The development graph manager MAYA developed in Saarbrücken already provides a management of change for homogeneous development graphs [AHMS00, AM02]. The central motivation is the

[1] For example, the specification of safety properties of the Bremen autonomous wheel chair [RL00, LR01].

[2] See `http://www.tzi.de/cofi/hets`.

insight that not only implementations, but also specifications may be incorrect and therefore are subject to change. The central idea of [AHMS00] is to compute the difference between two different versions of a development graph and use heuristics to match the nodes and links of the two graphs. Based on this and the information about the axioms and theorems that have been used in the proofs, the change management can then determine which proofs remain valid without change and which proof goals have to be re-proved (either be re-running the respective tactic script, or by adapting it or even providing a completely new proof). This change management shall be extended to heterogeneous development graphs with hiding. In order to achieve logic independence, the only properties that will be used are those of an entailment system in the sense of [Mes89b]. Moreover, it is planned to actually realize a much more general change management, based on abstract objects and dependencies among them, and instantiate this with heterogeneous development graphs.

A related topic is version control and configuration management. Here, the idea is to integrate HETS with the UniForM workbench [KPO+99]. Also, it sounds promising to enhance the XML interface in such a way that the OMDOC infrastructure [Koh00] can be used for HETS.

8.1.2 Reduction Strategies

At at least two places, reduction strategies for rewriting will play a role in HETS.

Currently, the quotient on the Grothendieck signature category is not implemented yet. The implementation will represent the quotient category as a composition graph [Sch99]. A central question is the study of termination and confluence properties for the resulting term rewriting system.

Furthermore, the proof calculus for development graphs with hiding is non-deterministic as well. While the rules without hiding can easily be made deterministic, it is not clear how far this is possible also for the rules dealing with hiding. Also here, termination and confluence properties (this time for graph grammars) are the interesting things to study.

8.1.3 Flattening out Heterogeneity

The proof calculus for development graphs with hiding is of course ultimately based on entailment systems of the logics involved, using the rule **(Basic Inference)**, see Sect. 5.6. This rule is the interface to rule systems for the individual logics. The rule just flattens the theory in which a certain sentence has to be proved. In practice, the latter will be done using a theorem prover. However, flattening the theory usually is a bit too much: it is more advisable to keep the structure of the theory as much as possible and flatten out only those things that the prover cannot deal with. This means that the theory needs to be made homogeneous, i.e. any *inter-logic* signature morphism needs to be flattened away by translating the whole development graph along it. For most theorem provers, ordinary *intra-logic* signature morphisms that are not just inclusions must be flattened out as well, since most theorem provers do not support renaming. Note however that the hierarchical structuring along intra-logic inclusions can be kept. We thus have to add a flattening rule that allows to translate a flattenable node (i.e. without **derive**) along a comorphism.

A related topic is the logic-specific oracle for conservativity: in the proof of the completeness theorem (Theorem 6.26) the conservativity check is only used for a link whose source node is flattenable (Lemma 6.27) and thus easy to make homogeneous. However, the possibilities of homogenizing the target node need to be investigated.

8.1.4 Interface for Theorem Provers

Besides the goal of integrating more theorem provers (currently, SPASS, KRHyper and Racer are on the list), the general interface for theorem provers shall be improved. Actually, there are already a number of interface languages that are understood by more than one theorem provers: proof general, mathweb, the protegé language and an XML-RPC interface for theorem proving easing management of change [AM02]. HETS should learn these languages as well. If provers deliver proof trees (the

datastructure for this is already provided by HETS), then HETS can build heterogeneous proof trees and provide a small program that checks these for correctness.

8.1.5 Heterogeneous Architectural Specifications and Heterogeneous Refinement

Heterogeneous architectural specifications are just architectural specifications over the Grothendieck logic. Luckily, the semantics of architectural specifications given in [CoF04, II.5] neither existence of colimits of signatures nor exactness of the underlying institution (see Sect. 6.2 for a discussion why it is problematic to require this for Grothendieck institutions). Instead, the diagram semantics OF [CoF04, III.5.6] relies on the notion of a diagram ensuring amalgamability along a sink. This notion naturally applies also to Grothendieck institutions. An open problem is how to formulate (and implement, in the Heterogeneous Tool Set) the check for ensurance of amalgamability within the Grothendieck institution in terms of corresponding checks for the individual institutions. It might suffice to use weakly amalgamable cocones, but the question then is: how to find a factorization within the Grothendieck institutions (e.g. assuming factorizations for the individual logics)?

A related topic is making the refinement language for CASL developed in [MSA05] heterogeneous. Indeed, we expect that this should go through without any further technical problems. Of course, the real challenge is then the study of observational or behavioural refinement. This currently is being completed for the heterogeneous case.

Bibliography

[Abr03] S. Abramsky. Generic theories and theories of genericity. 2003. Invited talk at Fossacs 2003 (quotation from the talk, not the proceedings).

[AGM92] S. Abramsky, M. Gabbay, T. Maibaum. *Handbook of Logic in Computer Science Vol. 2 Computational Structures*. Clarendon Press, Oxford, 1992.

[AHS90] J. Adámek, H. Herrlich, G. Strecker. *Abstract and Concrete Categories*. Wiley, New York, 1990.

[AK95] Jiří Adámek, Václav Koubek. On the greatest fixed point of a set functor. *Theoretical Computer Science* **150**(1), 57–75, 16 October 1995.

[Ala02] Suad Alagi. Institutions: integrating objects, XML and databases. *Information and Software Technology* **44**, 207–216, 2002.

[All01] Michel Allem. Formal combination of the CCS process algebra with the CASL algebraic specification language, March 14 2001.

[AM82] M. Arbib, E. Manes. Parametrized data types do not need highly constrained parameters. *Inform. Control* **52**, 139–158, 1982.

[AF96] M. Arrais, J. L. Fiadeiro. Unifying theories in different institutions. In M. Haveraaen, O. Owe, O.-J. Dahl, eds., *Recent Trends in Data Type Specifications. 11th Workshop on Specification of Abstract Data Types*, *Lecture Notes in Computer Science* **1130**, 81–101. Springer Verlag, 1996.

[Asp97] D. Aspinall. *Type Systems for Modular Programming and Specification*. PhD thesis, Edinburgh, 1997.

[ABK+] E. Astesiano, M. Bidoit, H. Kirchner, B. Krieg-Brückner, P. D. Mosses, D. Sannella, A. Tarlecki. CASL: The common algebraic specification language. *Theoretical Computer Science* **this volume**.

[AC92] E. Astesiano, M. Cerioli. Relationships between logical frameworks. In M. Bidoit, C. Choppy, eds., *Proc. 8th ADT workshop*, *Lecture Notes in Computer Science* **655**, 126–143. Springer Verlag, 1992.

[AC94] E. Astesiano, M. Cerioli. Multiparadigm specification languages: a first attempt at foundations. In C.M.D.J. Andrews, J.F. Groote, eds., *Semantics of Specification Languages (SoSl 93)*, Workshops in Computing, 168–185. Springer Verlag, 1994.

[AC95] E. Astesiano, M. Cerioli. Free objects and equational deduction for partial conditional specifications. *Theoretical Computer Science* **152**, 91–138, 1995.

[AHMS00] S. Autexier, D. Hutter, H. Mantel, A. Schairer. Towards an evolutionary formal software-development using CASL. In C. Choppy, D. Bert, eds., *Recent Trends in Algebraic Development Techniques, 14th International Workshop, WADT'99, Bonas, France*, *Lecture Notes in Computer Science* **1827**, 73–88. Springer-Verlag, 2000.

[AHMS02] S. Autexier, D. Hutter, T. Mossakowski, A. Schairer. The development graph manager MAYA (system description). In H. Kirchner, C. Reingeissen, eds., *Algebraic Methodology and Software Technology, 2002*, *Lecture Notes in Computer Science* **2422**, 495–502. Springer-Verlag, 2002.

[AM02] S. Autexier, T. Mossakowski. Integrating HOL-CASL into the development graph manager MAYA. In A. Armando, ed., *Frontiers of Combining Systems, 4th International Workshop*, *Lecture Notes in Computer Science* **2309**, 2–17. Springer-Verlag, 2002.

[Bar74] Jon Barwise. Axioms for abstract model theory. *Annals of Mathematical Logic* **7**, 221–265, 1974.

[Bau98] H. Baumeister. *Relations between Abstract Datatypes modeled as Abstract Datatypes*. PhD thesis, Universität des Saarlandes, 1998.

[Bau01] H. Baumeister. An institution for SB-CASL. talk presented at the 15th International Workshop on Algebraic Development Techniques, Genova, 2001.

[BZ00] H. Baumeister, A. Zamulin. State-based extension of CASL. In *Proceedings IFM 2000*, *Lecture Notes in Computer Science* **1945**. Springer-Verlag, 2000.

[BHK90] J.A. Bergstra, J. Heering, P. Klint. Module algebra. *J. ACM* **37**(2), 335–372, 1990.

[BHK89] Jan A. Bergstra, Jan Heering, Paul Klint. The algebraic specification formalism ASF. In J. A. Bergstra, J. Heering, P. Klint, eds., *Algebraic Specification*, ACM Press Frontier Series. Addison-Wesley, 1989.

[BCL96] G. Bernot, S. Coudert, P. Le Gall. Towards heterogeneous formal specifications. In *AMAST 96*, *Lecture Notes in Computer Science* **1101**, 458–472. 1996.

[BHa] M. Bidoit, R. Hennicker. Constructor-based observational logic. Technical report, CNRS-LSV Cachan.

[BHb] M. Bidoit, R. Hennicker. Using an institution encoding for proving consequences of structured COL-specifications. Talk at the WADT 2002, Frauenchiemsee.

[BGM89] Michel Bidoit, Marie-Claude Gaudel, A. Mauboussin. How to make algebraic specifications more understandable? An experiment with the PLUSS specification language. *Science of Computer Programming* **12**(1), 1–38, 1989.

[BM04] Michel Bidoit, Peter D. Mosses. CASL *User Manual*. LNCS Vol. 2900 (IFIP Series). Springer, 2004. With chapters by Till Mossakowski, Donald Sannella, and Andrzej Tarlecki.

[BRV01] P. Blackburn, M. Rijke, Y. Venema. *Modal Logic*. Cambridge University Press, 2001.

[BDG+98] Jürgen Bohn, Werner Damm, Orna Grumberg, Hardi Hungar, Karen Laster. First-order-CTL model checking. In V. Arvind, R. Ramanujam, eds., *Proceedings of the 18th Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS'98 (Chennai, India, December 17-19, 1998)*, *LNCS* **1530**, 283–294. Springer-Verlag, Berlin-Heidelberg-New York-Barcelona-Hong Kong-London-Milan-Paris-Singapore-Tokyo, 1998.

[BBD+00] Eerke Boiten, Howard Bowman, John Derrick, Peter Linington, Maarten Steen. Viewpoint consistency in ODP. *Computer Networks (Amsterdam, Netherlands: 1999)* **34**(3), 503–537, September 2000.

[Bor94] F. Borceux. *Handbook of Categorical Algebra I – III*. Cambridge University Press, 1994.

[Bor99] T. Borzyszkowski. Moving specification structures between logical systems. In J. L. Fiadeiro, ed., *Recent Trends in Algebraic Development Techniques, 13th International Workshop, WADT'98, Lisbon, Portugal, April 1998, Selected Papers*, *Lecture Notes in Computer Science* **1589**, 16–30. Springer, 1999.

[Bor02] T. Borzyszkowski. Logical systems for structured specifications. *Theoretical Computer Science* **286**, 197–245, 2002.

[Bor00] Tomasz Borzyszkowski. Generalized interpolation in CASL. *Information Processing Letters* **76/1-2**, 19–24, 2000.

[BJKO00] Mark G. J. van den Brand, Hayco A. de Jong, Paul Klint, Pieter A. Olivier. Efficient annotated terms. *Software: Practice and Experience* **30**(3), 259–291, 2000.

[BTM85] V. Breazu-Tannen, A. R. Meyer. Lambda calculus with constrained types. In *Logic of Programs*, *LNCS* **193**, 23–40. Springer, 1985.

[BFG+93] M. Broy, C. Facchi, R. Grosu, R. Hettler, H. Hußmann, D. Nazareth, F. Regensburger, O. Slotosch, K. Stølen. The requirement and design specification language SPECTRUM: An informal introduction (v 1.0). Technical Report TUM-I9311, TUM-I9312, Institut für Informatik, Technische Universität München, 1993.

[Bur86] P. Burmeister. *A model theoretic approach to partial algebras*. Akademie Verlag, Berlin, 1986.

[Bur93] P. Burmeister. Partial algebras — an introductory survey. In *Algebras and Orders*, NATO ASI Series C, 1–70. Kluwer, 1993.

[BLR02] P. Burmeister, M. Llabrés, F. Rosselló. Pushout complements for partly total algebras. *Mathematical Structures in Computer Science* **12**(2), 177–201, 2002.

[BG80] R. M. Burstall, J. A. Goguen. The semantics of CLEAR, a specification language. In D. Bjørner, ed., *Abstract Software Specifications, 1979 Copenhagen Winter School, Proceedings*, LNCS Vol. 86, 292–332. Springer, 1980.

[CGR03] C. Caleiro, P. Gouveia, J. Ramos. Completeness results for fibred parchments: Beyond the propositional base. In M. Wirsing, D. Pattinson, R. Hennicker, eds., *Recent Trends in Algebraic Development Techniques - Selected Papers*, *Lecture Notes in Computer Science* **2755**, 185–200. Springer-Verlag, 2003.

[CMRS02] C. Caleiro, P. Mateus, J. Ramos, A. Sernadas. Combining logics: Parchments revisited. In M. Cerioli, G. Reggio, eds., *Recent Trends in Algebraic Development Techniques, 15th International Workshop, WADT'01, Genova, Italy*, *Lecture Notes in Computer Science* **2267**, 48–70. Springer-Verlag, 2002.

[Cer93] M. Cerioli. *Relationships between Logical Formalisms*. PhD thesis, TD-4/93, Università di Pisa-Genova-Udine, 1993.

[CM97] M. Cerioli, J. Meseguer. May I borrow your logic? (transporting logical structures along maps). *Theoretical Computer Science* **173**, 311–347, 1997.

[CMR99] M. Cerioli, T. Mossakowski, H. Reichel. From total equational to partial first order logic. In E. Astesiano, H.-J. Kreowski, B. Krieg-Brückner, eds., *Algebraic Foundations of Systems Specifications*, 31–104. Springer Verlag, 1999.

[CR98] M. Cerioli, G. Reggio. Very abstract specifications: a formalism independent approach. *Math. Struct. Comput. Sci.* **8**, 17–66, 1998.

[CHKBM97] Maura Cerioli, Anne Haxthausen, Bernd Krieg-Brückner, Till Mossakowski. Permissive subsorted partial logic in CASL. In Michael Johnson, ed., *Algebraic methodology and software technology: 6th international conference, AMAST 97*, *Lecture Notes in Computer Science* **1349**, 91–107. Springer-Verlag, 1997.

[Cîr02] C. Cîrstea. On specification logics for algebra-coalgebra structures: Reconciling reachability and observability. *LNCS* **2303**, 82–97, 2002.

[CEW93] I. Claßen, H. Ehrig, D. Wolz. *Algebraic Specification Techniques and Tools for Software Development*. AMAST Series in Computing Vol. 1. World Scientific, 1993.

[CGRW95] I. Claßen, M. Große-Rhode, U. Wolter. Categorical concepts for parameterized partial specification. *Mathematical Structures in Computer Science* **5**, 153–188, 1995.

[CF92] R. Cockett, T. Fukushima. About Charity. Yellow Series Report 92/480/18, Univ. of Calgary, Dept. of Comp. Sci., 1992.

[CoF] CoFI. The Common Framework Initiative for algebraic specification and development, electronic archives. Notes and Documents accessible from `http://www.cofi.info/`.

[CoF97] CoFI Semantics Task Group. CASL – The CoFI Algebraic Specification Language (version 0.97) – Semantics. Note S-6, in [CoF], July 1997.

[CoF99] CoFI Semantics Task Group. CASL – The CoFI Algebraic Specification Language – Semantics. Note S-9 (Documents/CASL/Semantics, version 0.96), in [CoF], July 1999.

[CoF04] CoFI (The Common Framework Initiative). CASL *Reference Manual*. LNCS Vol. 2960 (IFIP Series). Springer, 2004.

[Coq86] T. Coquand. An analysis of Girard's paradox. In *Logic in Computer Science*, 227–236. IEEE, 1986.

[CBL99] S. Coudert, G. Bernot, P. Le Gall. Hierarchical heterogeneous specifications. In J. L. Fiadeiro, ed., *Recent Trends in Algebraic Development Techniques, 13th International Workshop, WADT'98, Lisbon, Portugal, April 1998, Selected Papers*, *Lecture Notes in Computer Science* **1589**, 106–120. Springer, 1999.

[CH95] Max J. Cresswell, G. E. Hughes. *A New Introduction to Modal Logic*. Routledge, London, 1995.

[CO89] P.-L. Curien, A. Obtułowicz. Partiality, Cartesian closedness and toposes. *Inform. and Comput.* **80**, 50–95, 1989.

[Dia] R. Diaconescu. *Institution-independent model theory*. Manuscript, University of Bucharest.

[Dia98] R. Diaconescu. Extra theory morphisms for institutions: logical semantics for multi-paradigm languages. *J. Applied Categorical Structures* **6**, 427–453, 1998.

[Dia02] R. Diaconescu. Grothendieck institutions. *Applied categorical structures* **10**, 383–402, 2002.

[DF96] R. Diaconescu, K. Futatsugi. Logical semantics of CafeOBJ. Technical report, JAIST, 1996. IS-RR-96-0024S.

[DF02] R. Diaconescu, K. Futatsugi. Logical foundations of CafeOBJ. *Theoretical computer science* **285**, 289–318, 2002.

[DGS91] Răzvan Diaconescu, Joseph Goguen, Petros Stefaneas. Logical support for modularisation. In Gerard Huet, Gordon Plotkin, eds., *Proceedings of a Workshop on Logical Frameworks*, 1991.

[DBM99] T. Dimitrakos, J. Bicarregui, T. Maibaum. Integrating heterogeneous formalisms: Framework and application, 1999.

[DM99] Francisco Durán, José Meseguer. Structured theories and institutions. In Martin Hofmann, Giuseppe Rosolini, Dusko Pavlovic, eds., *CTCS '99 Conference on Category Theory and Computer Science*, *Electronic Notes in Theoretical Computer Science* **29**. 1999.

[EM90a] H. Ehrig, B. Mahr. *Fundamentals of Algebraic Specification 2*. Springer Verlag, Heidelberg, 1990.

[EM90b] H. Ehrig, B. Mahr. *Fundamentals of Algebraic Specification 2. Module Specifications and Constraints*. EATCS Monographs on Theoretical Computer Science, Vol. 21. Springer, 1990.

[FM91] J. Fiadeiro, T. Maibaum. Temporal reasoning over deontic specifications. *Journal of Logic and Computation* **1**(3), 357–395, May 1991.

[FC96] Jose Luiz Fiadeiro, Jose Felix Costa. Mirror, mirror in my hand: A duality between specifications and models of process behaviour. *Mathematical Structures in Computer Science* **6**(4), 353–373, 1996.

[GM] Dov Gabbay, Larisa Maksimova. *Interpolation and Definability Volume 1*. forthcoming, see`http://www.dcs.kcl.ac.uk/staff/dg/Interpolation.ps.gz`.

[GHH+92] C. George, P. Haff, K. Havelund, A. E. Haxthausen, R. Milne, C. Bendix Nielson, S. Prehn, K. R. Wagner. *The Raise Specification Language*. Prentice Hall, 1992.

[GMM97] Paul Gibson, Bruno Mermet, Dominique Mery. Feature interactions: A mixed semantic model approach. In *IWFM*, 1997.

[GM00] J. Goguen, G. Malcolm. A hidden agenda. *Theoret. Comput. Sci.* **245**, 55–101, 2000.

[Gog] J. A. Goguen. Information integration in institutions. to appear in a Jon Barwise memorial volume edited by Larry Moss.

[GB92] J. A. Goguen, R. Burstall. Institutions: Abstract model theory for specification and programming. *J. ACM* **39**, 95–146, 1992.

[GB84] J. A. Goguen, R. M. Burstall. Introducing institutions. *Lecture Notes in Computer Science* **164**, 221–256. Springer Verlag, 1984.

[GM86] J. A. Goguen, J. Meseguer. Eqlog: Equality, types, and generic modules for logic programming. In D. DeGroot, G. Lindstrom, eds., *Logic Programming. Functions, Relations and Equations*, 295–363. Prentice-Hall, Englewood Cliffs, New Jersey, 1986.

[GM92] J. A. Goguen, J. Meseguer. Order-sorted algebra I: equational deduction for multiple inheritance, overloading, exceptions and partial operations. *Theoretical Computer Science* **105**, 217–273, 1992.

[GTW78] J. A. Goguen, J. W. Thatcher, E. G. Wagner. An initial algebra approach to the specification, correctness and implementation of abstract data types. In R. Yeh, ed., *Current Trends in Programming Methodology*, 4, 80–144. Prentice Hall, 1978.

[GW88] J. A. Goguen, T. Winkler. Introducing OBJ3. Research report SRI-CSL-88-9, SRI International, 1988.

[GWM+92] J. A. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, J.-P. Jouannaud. Introducing OBJ3. In Joseph Goguen, Grant Malcolm, eds., *Software Engineering with OBJ: Algebraic Specification in Action*. Kluwer, 1992.

[GD94] Joseph Goguen, Răzvan Diaconescu. An Oxford survey of order sorted algebra. *Mathematical Structures in Computer Science* **4**(3), 363–392, September 1994.

[GT00] Joseph A. Goguen, Will Tracz. An implementation-oriented semantics for module composition. In Gary T. Leavens, Murali Sitaraman, eds., *Foundations of Component-Based Systems*, chapter 11, 231–263. Cambridge University Press, New York, NY, 2000.

[Gro63] A. Grothendieck. Catégories fibrées et descente. In *Revêtements Étales et Group Fondamental: Séminaire de Géométrie Algébrique du Bois Marie 1960/61 (SGA 1), Exposé VI*. Institut des Hautes Études Scientifiques, Paris, 1963. Reprinted in Lecture Notes in Mathematics No. 224, Springer-Verlag, 1971.

[GS] H. P. Gumm, T. Schröder. Coalgebras of bounded type. *Math. Struct. Comput. Sci.* to appear.

[GH93] J. V. Guttag, J. J. Horning. *Larch: Languages and Tools for Formal Specification*. Springer, 1993.

[Hal03] Thomas Hallgren. Haskell tools from the programatica project. In *Proceedings of the ACM SIGPLAN workshop on Haskell (HASKELL-03)*, 103–106, New York, August 28 2003. ACM Press.

[Har79] D. Harel. *First-Order Dynamic Logic (LNCS Volume 68)*. Springer-Verlag: Heidelberg, Germany, 1979.

[HS73] H. Herrlich, G. Strecker. *Category Theory*. Allyn and Bacon, Boston, 1973.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[HPF99] P. Hudak, J. Peterson, J. H. Fasel. A gentle introduction into Haskell 98. Available at `http://www.haskell.org`, 1999.

[HLS+96] Dieter Hutter, Bruno Langenstein, Claus Sengler, Jörg H. Siekmann, Werner Stephan, Wolpers Wolpers. Verification support environment (VSE). *High Integrity Systems* **1**(6), 523–530, 1996.

[IM05] Y. Isobe, M.Roggenbach. A generic theorem prover of csp refinement. Proceedings of TACAS 2005, to appear in LNCS, Springer, 2005.

[JR97] Jacobs, Rutten. A tutorial on (co)algebras and (co)induction. *BEATCS: Bulletin of the European Association for Theoretical Computer Science* **62**, 1997.

[Jac00] B. Jacobs. Towards a duality result in the modal logic of coalgebras. In *Coalgebraic Methods in Computer Science*, *Electron. Notes Theoret. Comput. Sci.* **33**. Elsevier, 2000.

[Jac01] B. Jacobs. Many-sorted coalgebraic modal logic: a model-theoretic study. *Theor. Inform. Appl.* **35**, 31–59, 2001.

[Jac02] Bart Jacobs. *Exercises in Coalgebraic Specification*, *Lecture Notes in Computer Science* **2297**, 237–280. 2002.

[Jon90] C. B. Jones. *Systematic Software Development Using VDM*. Prentice Hall, 1990.

[KST97] S. Kahrs, D. Sannella, A. Tarlecki. The definition of Extended ML: A gentle introduction. *Theoretical Computer Science* **173**, 445–484, 1997.

[KST94] Stefan Kahrs, Donald Sannella, Andrzej Tarlecki. Interfaces and Extended ML. In *Proc. ACM Workshop on Interface Definition Languages*, 111–118. ACM SIGPLAN Notices 29(8), 1994.

[Kie03] Richard B. Kieburtz. P-logic: property verification for haskell programs, April 23 2003.

[Koh00] Michael Kohlhase. OMDoc: An infrastructure for OpenMath content dictionary information. *SIGSAM Bulletin (ACM Special Interest Group on Symbolic and Algebraic Manipulation)* **34**(2), 43–48, June 2000.

[Koz83] D. Kozen. Results on the propositional μ-calculus. *Theoret. Comput. Sci.* **27**, 333–354, 1983.

[KM95] H.-J. Kreowski, T. Mossakowski. Equivalence and difference of institutions: Simulating Horn clause logic with based algebras. *Mathematical Structures in Computer Science* **5**, 189–215, 1995.

[KPO+99] B. Krieg-Brückner, J. Peleska, E.-R. Olderog, D. Balzer, A. Baer. The UniForM Workbench, a universal development environment for formal methods. In *FM99: World Congress on Formal Methods*, *Lecture Notes in Computer Science* **1709**, 1186–1205. Springer-Verlag, 1999.

[KBS88] R. Kubiak, A. Borzyszkowski, S. Sokolowski. A set-theoretic model for a typed polymorphic lambda calculus — a contribution to MetaSoft. In *VDM: The Way Ahead*, *LNCS* **328**, 267–298. Springer, 1988.

[Kur01a] A. Kurz. Coalgebras and modal logic. Course Notes for the European Summer School on Logic, Language and Information (CD-ROM), University of Helsinki, 2001.

[Kur01b] A. Kurz. Specifying coalgebras with modal logic. *Theoret. Comput. Sci.* **260**, 119–138, 2001.

[LS86] J. Lambek, P. J. Scott. *Introduction to Higher Order Categorical Logic*. Cambridge University Press, 1986.

[Lan66] P. J. Landin. The Next 700 Programming Languages. *Communications of the ACM* **9**(3), 157–166, 1966.

[Lan72] S. Mac Lane. *Categories for the working mathematician*. Springer, 1972.

[LR01] A. Lankenau, T. Röfer. The Bremen Autonomous Wheelchair – a versatile and safe mobility assistant. *IEEE Robotics and Automation Magazine, "Reinventing the Wheelchair"* **7**(1), 29 – 37, Mar. 2001.

[LM] Daan Leijen, Erik Meijer. Parsec: Direct style monadic parser combinators for the real world. Technical report. UU-CS-2001-35.

[Llo87] J.W. Lloyd. *Foundations of Logic Programming*. Springer Verlag, 1987.

[LF97] Antonia Lopes, Jose Luiz Fiadeiro. Preservation and reflection in specification. In *Algebraic Methodology and Software Technology*, 380–394, 1997.

[LTKKB99] C. Lüth, H. Tej, Kolyang, B. Krieg-Brückner. TAS and IsaWin: Tools for transformational program developkment and theorem proving. In J.-P. Finance, ed., *Fundamental Approaches to Software Engineering FASE'99. Joint European Conferences on Theory and Practice of Software ETAPS'99*, number 1577 in LNCS, 239–243. Springer Verlag, 1999.

[LW99] C. Lüth, B. Wolff. Functional design and implementation of graphical user interfaces for theorem provers. *Journal of Functional Programming* **9**(2), 167–189, March 1999.

[MSS90] V. Manca, A. Salibra, G. Scollo. Equational type logic. *Theoretical Computer Science* **77**, 131–159, 1990.

[MOM95] N. Martí-Oliet, J. Meseguer. From abstract data types to logical frameworks. *Lecture Notes in Computer Science* **906**, 48–80. Springer, 1995.

[Mes89a] J. Meseguer. General logics. In *Logic Colloquium 87*, 275–329. North Holland, 1989.

[Mes89b] J. Meseguer. General logics. In *Logic Colloquium 87*, 275–329. North Holland, 1989.

[Mes92] J. Meseguer. Conditional rewriting as a unified model of concurrency. *Theoretical Computer Science* **96**(1), 73–156, 1992.

[Mes98a] J. Meseguer. Membership algebra as a logical framework for equational specification. In F. Parisi Presicce, ed., *Recent trends in algebraic development techniques. Proc. 12th International Workshop*, *Lecture Notes in Computer Science* **1376**, 18–61. Springer, 1998.

[Mes98b] José Meseguer. Formal interoperability. In *Proceedings of the 1998 Conference on Mathematics in Artificial Intelligence*, Fort Laurerdale, Florida, January 1998. `http://rutcor.rutgers.edu/ amai/Proceedings.html`. Presented also at the *14th IMACS World Congress*, Atlanta, Georgia, July 1994.

[MS89] J. C. Mitchell, P. J. Scott. Typed lambda models and cartesian closed categories. In *Categories in Computer Science and Logic*, *Contemp. Math.* **92**, 301–316. Amer. Math. Soc., 1989.

[MP00] I. Moerdijk, E. Palmgren. Wellfounded trees in categories. *Ann. Pure Appl. Logic* **104**, 189–218, 2000.

[Mog86] E. Moggi. Categories of partial morphisms and the λ_p-calculus. In *Category Theory and Computer Programming*, *LNCS* **240**, 242–251. Springer, 1986.

[Mog88] E. Moggi. *The Partial Lambda Calculus*. PhD thesis, University of Edinburgh, 1988.

[Mos99] L. Moss. Coalgebraic logic. *Ann. Pure Appl. Logic* **96**, 277–317, 1999.

[Mos95] T. Mossakowski. A hierarchy of institutions separated by properties of parameterized abstract data types. In E. Astesiano, G. Reggio, A. Tarlecki, eds., *Recent Trends in Data Type Specification. Proceedings*, *Lecture Notes in Computer Science* **906**, 389–405. Springer Verlag, London, 1995.

[Mos96a] T. Mossakowski. Equivalences among various logical frameworks of partial algebras. In H. Kleine Büning, ed., *Computer Science Logic. 9th Workshop, CSL'95. Paderborn, Germany, September 1995, Selected Papers*, *Lecture Notes in Computer Science* **1092**, 403–433. Springer Verlag, 1996.

[Mos96b] T. Mossakowski. *Representations, hierarchies and graphs of institutions.* PhD thesis, Bremen University, 1996.

[Mos96c] T. Mossakowski. *Representations, hierarchies and graphs of institutions.* PhD thesis, Universität Bremen, 1996. also: Logos-Verlag, 2001.

[Mos96d] T. Mossakowski. Using limits of parchments to systematically construct institutions of partial algebras. In M. Haveraaen, O. Owe, O.-J. Dahl, eds., *Recent Trends in Data Type Specifications. 11th Workshop on Specification of Abstract Data Types*, *Lecture Notes in Computer Science* **1130**, 379–393. Springer Verlag, 1996.

[Mos98] T. Mossakowski. Colimits of order-sorted specifications. In F. Parisi Presicce, ed., *Recent trends in algebraic development techniques. Proc. 12th International Workshop*, *Lecture Notes in Computer Science* **1376**, 316–332. Springer, 1998.

[Mos00] T. Mossakowski. Specification in an arbitrary institution with symbols. In C. Choppy, D. Bert, P. Mosses, eds., *Recent Trends in Algebraic Development Techniques, 14th International Workshop, WADT'99, Bonas, France*, *Lecture Notes in Computer Science* **1827**, 252–270. Springer-Verlag, 2000.

[Mos02a] T. Mossakowski. Comorphism-based Grothendieck logics. In K. Diks, W. Rytter, eds., *Mathematical foundations of computer science*, *LNCS* **2420**, 593–604. Springer, 2002.

[Mos02b] T. Mossakowski. Heterogeneous development graphs and heterogeneous borrowing. In M. Nielsen, U. Engberg, eds., *Foundations of Software Science and Computation Structures*, *Lecture Notes in Computer Science* **2303**, 326–341. Springer-Verlag, 2002.

[Mos03] T. Mossakowski. Foundations of heterogeneous specification. In M. Wirsing, D. Pattinson, R. Hennicker, eds., *Recent Trends in Algebraic Development Techniques, 16th International Workshop, WADT 2002, Frauenchiemsee, Germany, 2002, Revised Selected Papers*, LNCS Vol. 2755, 359–375. Springer, 2003.

[MA] T. Mossakowski, A. Arnould. Testing and proving behavioural refinement in CASL. Submitted.

[MAH] T. Mossakowski, S. Autexier, D. Hutter. Extending development graphs with hiding. *Journal of Logic and Algebraic Programming.* to appear.

[MK02] T. Mossakowski, B. Klin. Institution independent static analysis for CASL. In M. Cerioli, G. Reggio, eds., *Recent Trends in Algebraic Development Techniques, 15th International Workshop, WADT'01, Genova, Italy*, *Lecture Notes in Computer Science* **2267**, 221–237. Springer-Verlag, 2002.

[MKK98] T. Mossakowski, Kolyang, B. Krieg-Brückner. Static semantic analysis and theorem proving for CASL. In F. Parisi Presicce, ed., *Recent trends in algebraic development techniques. Proc. 12th International Workshop*, *Lecture Notes in Computer Science* **1376**, 333–348. Springer, 1998.

[MSA05] T. Mossakowski, D. Sannella, A.Tarlecki. A simple refinement language for . In Jose Luiz Fiadeiro, ed., *WADT 2004*, *Lecture Notes in Computer Science* **3423**, 162–185. Springer; Berlin; http://www.springer.de, 2005.

[MTP98] T. Mossakowski, A. Tarlecki, W. Pawłowski. Combining and representing logical systems using model-theoretic parchments. In F. Parisi Presicce, ed., *Recent trends in algebraic development techniques. Proc. 12th International Workshop*, *Lecture Notes in Computer Science* **1376**, 349–364. Springer, 1998.

[Mos97] Till Mossakowski. Sublanguages of CASL. Note L-7, in [CoF], December 1997.

[Mos02] Till Mossakowski. Relating CASL with other specification languages: the institution level. *Theoretical Computer Science* **286**, 367–475, 2002.

[MAH01] Till Mossakowski, Serge Autexier, Dieter Hutter. Extending development graphs with hiding. In H. Hussmann, ed., *Fundamental Approaches to Software Engineering, 4th International Conference, FASE 2001, Genova, Italy, Proceedings*, LNCS Vol. 2029, 269–283. Springer, 2001.

[MHAH04] Till Mossakowski, Piotr Hoffman, Serge Autexier, Dieter Hutter. CASL logic. In Peter D. Mosses, ed., *CASL Reference Manual*, *Lecture Notes in Computer Science* **2960**, part ÏV. Springer Verlag, London, 2004. Edited by T. Mossakowski.

[MSRR] Till Mossakowski, Lutz Schröder, Markus Roggenbach, Horst Reichel. Algebraic-co-algebraic specification in CoCASL. *Journal of Logic and Algebraic Programming*. To appear.

[Mos89] P. D. Mosses. Unified algebras and institutions. Proceedings of the 4th Annual IEEE Symposium on Logic in Computer Science, 304–312. 1989.

[Mos97] Peter D. Mosses. CoFI: The Common Framework Initiative for Algebraic Specification and Development. In *TAPSOFT '97, Proc. Intl. Symp. on Theory and Practice of Software Development*, *LNCS* **1214**, 115–137. Springer-Verlag, 1997.

[NP86] Mogens Nielsen, Udo Pletat. Polymorphism in an institutional framework, 1986. Technical University of Denmark.

[NPW02] T. Nipkow, L. C. Paulson, M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, *LNCS* **2283**. Springer, 2002.

[NPS90] B. Nordström, K. Petersson, J. Smith. *Programming in Martin-Löf's Type Theory: An Introduction*. Oxford Univ. Press, 1990.

[OP02] Fernando Orejas, Elvira Pino. Heterogeneous modular systems. In *Integrated Design and Process Technology*. Sociey for Design and Process Science, 2002.

[Pad88] P. Padawitz. *Computing in Horn Clause Theories*. Springer Verlag, Heidelberg, 1988.

[Pat01] D. Pattinson. *Expressivity Results in the Modal Logic of Coalgebras*. PhD thesis, University of Munich, 2001.

[Pat02] D. Pattinson. Expressive logics for coalgebras via terminal sequence induction. Technical report, LMU München, 2002.

[Pau90] L. C. Paulson. Isabelle: The next 700 theorem provers. In P. Odifreddi, ed., *Logic and Computer Science*, 361–386. Academic Press, 1990.

[Pau94] L. C. Paulson. *Isabelle - A Generic Theorem Prover*. Number 828 in LNCS. Springer Verlag, 1994.

[PP99] D. Pavlovic, V. Pratt. On coalgebra of real numbers. In *Coalgebraic Methods in Computer Science*, *Electron. Notes Theoret. Comput. Sci.* **19**. Elsevier, 1999.

[Pep91] P. Pepper. Transforming algebraic specifications – lessons learnt from an example. In *Constructing Programs from Specifications*, 1–27. Elsevier, 1991.

[PJ03] S. Peyton-Jones, ed. *Haskell 98 Language and Libraries — The Revised Report*. Cambridge, 2003. also: J. Funct. Programming **13** (2003).

[PJM97] Simon Peyton Jones, Mark Jones, Erik Meijer. Type classes: exploring the design space. In *Haskell Workshop*. 1997.

[Pop94] Sally Popkorn. *First Steps in Modal Logic*. Cambridge University Press, Cambridge, England, 1994.

[Reg95] F. Regensburger. HOLCF: Higher order logic of computable functions. In *Theorem Proving in Higher Order Logics*, *LNCS* **971**, 293–307, 1995.

[RAC00] G. Reggio, E. Astesiano, C. Choppy. CASL-LTL - a CASL extension for dynamic reactive systems - summary. Technical Report of DISI - Università di Genova, DISI-TR-99-34, Italy, 2000.

[Rei87] H. Reichel. *Initial Computability, Algebraic Specifications and Partial Algebras*. Oxford Science Publications, 1987.

[Rei95] H. Reichel. An approach to object semantics based on terminal co-algebras. *Math. Struct. Comput. Sci.* **5**, 129–152, 1995.

[Rei00] H. Reichel. A uniform model theory for the specification of data and process types. In *Recent Developments in Algebraic Development Techniques, 14th International Workshop (WADT 99)*, *LNCS* **1827**, 348–365. Springer, 2000.

[RL00] T. Röfer, A. Lankenau. Architecture and applications of the Bremen Autonomous Wheelchair. *Information Sciences* **126**(1-4), 1 – 20, Jul. 2000.

[Rog] Markus Roggenbach. CSP-CASL – A new integration of process algebra and algebraic specification. Submitted for journal publication.

[Ros97] A.W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall, 1997.

[Ros86] G. Rosolini. *Continuity and effectiveness in topoi*. PhD thesis, University of Oxford, 1986.

[Rößi00] M. Rößiger. Coalgebras and modal logic. In *Coalgebraic Methods in Computer Science*, *Electron. Notes Theoret. Comput. Sci.* **33**. Elsevier, 2000.

[Roş00] G. Roşu. *Hidden Logic*. PhD thesis, Univ. of California at San Diego, 2000.

[RG04] Grigore Rosu, Joseph Goguen. Composing hidden information modules over inclusive institutions, 2004.

[RTJ01] J. Rothe, H. Tews, B. Jacobs. The Coalgebraic Class Specification Language CCSL. *J. Universal Comput. Sci.* **7**, 175–193, 2001.

[Rut00] J. Rutten. Universal coalgebra: A theory of systems. *Theoret. Comput. Sci.* **249**, 3–80, 2000.

[SS93] A. Salibra, G. Scollo. A soft stairway to institutions. In *Recent Trends in Data Type Specification, 8th International Workshop (WADT 91)*, *LNCS* **655**, 310–329. Springer, 1993.

[SS96] Antonino Salibra, Giuseppe Scollo. Interpolation and compactness in categories of pre-institutions. *Mathematical Structures in Computer Science* **6**(3), 261–286, June 1996.

[STa] D. Sannella, A. Tarlecki. *Foundations of Algebraic Specifications and Formal Program Development*. Cambridge University Press, to appear. See `http://zls.mimuw.edu.pl/~tarlecki/book/index.html`.

[STb] D. Sannella, A. Tarlecki. *Working with multiple logical systems, In: Foundations of Algebraic Specifications and Formal Program Development*, chapter 10. Cambridge University Press, to appear. See `http://zls.mimuw.edu.pl/~tarlecki/book/index.html`.

[ST88a] D. Sannella, A. Tarlecki. Specifications in an arbitrary institution. *Information and Computation* **76**, 165–210, 1988.

[ST88b] D. Sannella, A. Tarlecki. Specifications in an arbitrary institution. *Information and Computation* **76**, 165–210, 1988.

[ST88c] D. Sannella, A. Tarlecki. Toward formal development of programs from algebraic specifications: implementations revisited. *Acta Informatica* **25**, 233–281, 1988.

[ST88d] Donald Sannella, Andrzej Tarlecki. Toward formal development of programs from algebraic specifications: Implementations revisited. *Acta Informatica* **25**, 233–281, 1988.

[Sch04] Klaus Schneider. *Verification of Reactive Systems*. Springer Verlag, 2004.

[Sch] L. Schröder. The HASCASL prologue: categorical syntax and semantics of the partial λ-calculus. Available as `http://www. informatik.uni-bremen.de/~lschrode/hascasl/plam.ps`

[Sch99] L. Schröder. *Composition graphs and free extensions of categories*. PhD thesis, University of Bremen, 1999. in German; also: Logos, Berlin, 1999.

[Sch03] L. Schröder. Henkin models of the partial λ-calculus. In *Computer Science Logic*, *LNCS* **2803**, 498–512. Springer, 2003.

[SM] L. Schröder, T. Mossakowski. Monad-independent dynamic logic in HASCASL. *J. Logic Comput.* To appear. Earlier version in *Recent Developments in Algebraic Development Techniques, 16th International Workshop (WADT 02)*, volume 2755 of *LNCS*, pages 425-442. Springer, 2003.

[SM02] L. Schröder, T. Mossakowski. HASCASL: Towards integrated specification and development of functional programs. In *Algebraic Methodology and Software Technology*, *LNCS* **2422**, 99–116. Springer, 2002.

[SM03] L. Schröder, T. Mossakowski. Monad-independent Hoare logic in HASCASL. In *Fundamental Approaches to Software Engineering*, *LNCS* **2621**, 261–277. Springer, 2003.

[SM04] L. Schröder, T. Mossakowski. Type class polymorphism in an institutional framework, 2004. Available as `http://www. informatik.uni-bremen.de/~lschrode/papers/genpoly.ps`

[SMM] L. Schröder, T. Mossakowski, C. Maeder. HasCasl – Integrated functional specification and programming. Language summary. available under `http://www.informatik.uni-bremen.de/agbkb/forschung/formal_methods/CoFI/HasCASL`

[SMT01a] L. Schröder, T. Mossakowski, A. Tarlecki. Amalgamation via enriched CASL signatures. In *International Colloquium on Automata, Languages and Programming (ICALP 2001)*, *Lect. Notes Comp. Sci.* **2076**, 993–1004. Springer, 2001.

[SMT+01b] L. Schröder, T. Mossakowski, A. Tarlecki, P. Hoffman, B. Klin. Semantics of architectural specifications in Casl. In *Fundamental Approaches to Software Engineering*, *LNCS* **2029**, 253–268. Springer, 2001.

[SML05] Lutz Schröder, Till Mossakowski, Christoph Lüth. Type class polymorphism in an institutional framework. In José Fiadeiro, ed., *Recent Trends in Algebraic Development Techniques, 17th International Workshop (WADT 2004)*, *Lecture Notes in Computer Science* **3423**, 234–248. Springer; Berlin; http://www.springer.de, 2005.

[Sco93] G. Scollo. *On the engineering of logics*. PhD thesis, 1993. University of Twente, Enschede.

[SCS94] A. Sernadas, J. F. Costa, C. Sernadas. An institution of object behaviour. In H. Ehrig, F. Orejas, eds., *Recent Trends in Data Type Specification*, *Lecture Notes in Computer Science* **785**, 337–350. Springer-Verlag, 1994.

[SS93] A. Sernadas, C. Sernadas. Denotational semantics of object specification within an arbitrary temporal logic institution. Research report, Section of Computer Science, Department of Mathematics, Instituto Superior Técnico, 1049-001 Lisboa, Portugal, 1993. Presented at IS-CORE Workshop 93.

[SSCM00] A. Sernadas, C. Sernadas, C. Caleiro, T. Mossakowski. Categorical fibring of logics with terms and binding operators. In D. Gabbay, M. de Rijke, eds., *Frontiers of Combining Systems 2*, Studies in Logic and Computation, 295–316. Research Studies Press, 2000.

[Sho67] J.R. Shoenfield. *Mathematical Logic*. Addison-Wesley, Reading, Massachusetts, 1967.

[Tar86] A. Tarlecki. Quasi-varieties in abstract algebraic institutions. *J. Comput. System Sci.* **33**, 333–360, 1986.

[Tar87] A. Tarlecki. Institution representation. draft note, 1987.

[Tar96] A. Tarlecki. Moving between logical systems. In M. Haveraaen, O. Owe, O.-J. Dahl, eds., *Recent Trends in Data Type Specifications. 11th Workshop on Specification of Abstract Data Types*, *Lecture Notes in Computer Science* **1130**, 478–502. Springer Verlag, 1996.

[Tar00] A. Tarlecki. Towards heterogeneous specifications. In D. Gabbay, M. de Rijke, eds., *Frontiers of Combining Systems 2, 1998*, Studies in Logic and Computation, 337–360. Research Studies Press, 2000.

[Tar04] A. Tarlecki. Software specification and development in heterogeneous environments. In *Combination of Logics 2004. Workshop, Lisbon*. 2004.

[TBG91] A. Tarlecki, R. M. Burstall, J. A. Goguen. Some fundamentals algebraic tools for the semantics of computation. Part 3: Indexed categories. *Theoretical Computer Science* **91**, 239–264, 1991.

[Tew02] H. Tews. *Coalgebraic Methods for Object–Oriented Languages*. PhD thesis, Technical Univ. of Dresden, 2002.

[Tha00] Lars Thalmann. *Term-modal logic and quantifier-free dynamic assignment logic*. PhD thesis, Uppsala University, 2000.

[TWW81] J. W. Thatcher, E. G. Wagner, J. B. Wright. Specification of abstract data types using conditional axioms. Technical Report RC 6214, IBM Yorktown Heights, 1981.

[Uni] Univ. of Glasgow. The Glasgow Haskell Compiler. http://www.haskell.org/ghc.

[vdBJ01] J. van den Berg, B. Jacobs. The LOOP compiler for Java and JML. *LNCS* **2031**, 299–312, 2001.

[vL90] Jan van Leeuwen, ed. *Handbook of Theoretical Computer Science, Volume B*. Elsevier / MIT Press, 1990.

[Wen97] M. Wenzel. Type classes and overloading in higher-order logic. In *Theorem Proving in Higher Order Logics*, *LNCS* **1275**, 307–322. Springer, 1997.

[Wir86] M. Wirsing. Structured algebraic specifications: A kernel language. *Theoretical Computer Science* **42**, 123–249, 1986.

[WDC+95] U. Wolter, K. Didrich, F. Cornelius, M. Klar, R. Wessäly, H. Ehrig. How to cope with the spectrum of SPECTRUM. In Manfred Broy, Stefan Jähnichen, eds., *KORSO: Methods, Languages and Tools for the Construction of Correct Software*, *Lecture Notes in Computer Science* **1009**, 173–189. Springer-Verlag, New York, NY, 1995.

Appendix A

Heterogeneous CASL (HETCASL) Language Summary

Heterogeneous CASL (HETCASL) allows mixing specifications written in different logics (using translations between the logics). It extends CASL only at the level of structuring constructs, by adding constructs for choosing the logic and translating specifications among logics. HETCASL is needed when combining specifications written in CASL with specifications written in its sublanguages and extensions. HETCASL also allows the integration of logics that are completely different from the CASL logic.

This document provides a detailed definition of the HETCASL syntax and an informal description of the semantics, building on the existing CASL Summary [CoF04].

About this document

This document gives a detailed summary of the syntax and intended semantics of HETCASL. It is intended for readers already familiar with CASL, in particular with CASL structured specifications and libraries, see [CoF04]. Like the CASL Summary [CoF04], this document provides little or nothing in the way of discussion or motivation of design decisions; for such matters, see in particular [Mos03].

Structure

The document consists of a chapter explaining the semantic concepts needed for heterogeneous specification (Chap. A.1), and a chapter (Chap. A.2) describing the language constructs of HETCASL (which extend CASL structured specifications and libraries).

Like the CASL Summary [CoF04], this document provides appendices containing the abstract syntax (Appendices A.3 and A.4) and the concrete syntax (Appendix A.5) of HETCASL specifications.

A.1 Heterogeneous Concepts

A.1.1 Institutions

HETCASL exploits the fact that CASL structured and architectural specifications are defined independently of the underlying framework of basic specifications, formalized in terms of so-called ***institutions*** [GB92] (some category-theoretic details are omitted below) and *proof systems*.

A *basic specification framework* may be characterized by:

- a class **Sig** of ***signatures*** Σ, each determining the set of ***symbols*** $|\Sigma|$ whose intended interpretation is to be specified, with ***morphisms*** between signatures;

- a class $\mathbf{Mod}(\Sigma)$ of ***models***, with ***homomorphisms*** between them, for each signature Σ;
- a set $\mathbf{Sen}(\Sigma)$ of ***sentences*** (or ***axioms***), for each signature Σ;
- a relation $\models$ of ***satisfaction***, between models and sentences over the same signature; and
- (optionally) a ***proof system***, for inferring sentences from sets of sentences.

A ***signature morphism*** $\sigma : \Sigma \to \Sigma'$ determines a ***translation*** function $\mathbf{Sen}(\sigma)$ on sentences, mapping $\mathbf{Sen}(\Sigma)$ to $\mathbf{Sen}(\Sigma')$, and a ***reduct*** function $\mathbf{Mod}(\sigma)$ on models, mapping $\mathbf{Mod}(\Sigma')$ to $\mathbf{Mod}(\Sigma)$.[1] Satisfaction is required to be preserved by translation: for all $S \in \mathbf{Sen}(\Sigma), M' \in \mathbf{Mod}(\Sigma')$,

$$\mathbf{Mod}(\sigma)(M') \models S \iff M' \models \mathbf{Sen}(\sigma)(S).$$

If present, the proof system is required to be sound, i.e., sentences inferred from a specification are always consequences; moreover, inference is to be preserved by translation.

The semantics of a structured specification consists of a signature Σ together with a class of models in $\mathbf{Mod}(\Sigma)$. A specification is said to be ***consistent*** when there are some models that satisfy all the sentences, and ***inconsistent*** when there are no such models. A sentence is a ***consequence*** of a specification if it is satisfied in all the models of the specification.

A.1.2 Institution Morphisms and Comorphisms

Heterogeneous specifications involve several institutions, which are related by ***institution morphisms*** and ***comorphisms*** [RG04].

An *institution morphism* from an institution I to an institution J consists of the following components:

- a translation Φ of I-signatures to J-signatures,
- a translation α of J-sentences over $\Phi(\Sigma)$ to I-sentences over Σ,
- a translation β of I-models over Σ to J-models over $\Phi(\Sigma)$,

such that satisfaction is preserved by translation along the institution morphism: for all $\Sigma \in \mathbf{Sig}^I$, $M \in \mathbf{Mod}^I(\Sigma)$ and $\varphi' \in \mathbf{Sen}^J(\Phi(\Sigma))$,

$$M \models^I_\Sigma \alpha_\Sigma(\varphi') \iff \beta_\Sigma(M) \models^J_{\Phi(\Sigma)} \varphi$$

While institution morphisms often are projections expressing the fact that a "richer" institution is built over a "poorer" one, *institution comorphisms* often formalize inclusions or encodings between institution. An institution comorphism is similar to an institution morphism; only the directions of sentence and model translation change. It consists of the following components:

- a translation Φ of I-signatures to J-signatures,
- a translation α of I-sentences over Σ to J-sentences over $\Phi(\Sigma)$,
- a translation β of J-models over $\Phi(\Sigma)$ to I-models over Σ,

such that satisfaction is preserved by translation along the institution comorphism: for all $\Sigma \in \mathbf{Sig}^I$, $M' \in \mathbf{Mod}^J(\Phi(\Sigma))$ and $\varphi \in \mathbf{Sen}^I(\Sigma)$:

$$M' \models^J_{\Phi(\Sigma)} \alpha_\Sigma(\varphi) \iff \beta_\Sigma(M') \models^I_\Sigma \varphi.$$

Simple theoroidal institution morphisms and comorphisms [RG04] admit extra flexibility: signatures may be mapped to theories (where a theory consists of a signature and a set of sentences

[1] In fact **Sig** is a category, and **Sen**(.) and **Mod**(.) are functors. The categorical aspects of the semantics of CASL are emphasized in its formal semantics [CoF04].

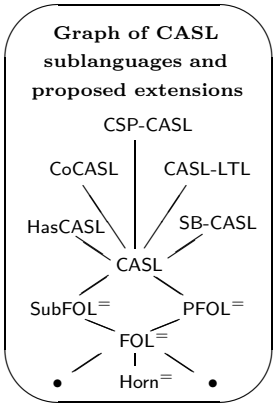


Figure A.1: A sample logic graph.

over that signature). In the sequel, we allow simple theoroidal (co)morphisms when we talk about (co)morphisms.

An institution comorphism is said to be a ***subinstitution comorphism***, if its signature and sentence translation components are embeddings, and its model translation component is an isomorphism. An institution I is said to be a ***subinstitution*** of an institution J if there is a subinstitution comorphism from I to J.[2]

Finally, a ***modification*** τ between two institution morphisms $(\Phi_1, \alpha_1, \beta_1) : I \to J$ and $(\Phi_2, \alpha_2, \beta_2)) : I \longrightarrow J$ consists of a family of signature morphisms $\tau_\Sigma : \Phi_1(\Sigma) \longrightarrow \Phi_2$, indexed by signature Σ in I, and satisfying some natural compatibility requirements. Modifications between comorphisms are defined similarly.

A.1.3 Logic Graphs

Heterogeneous specification is based on an arbitrary but fixed graph of institutions, morphisms, comorphisms and modifications, which we call the ***logic graph***. We will henceforth speak of logics when speaking about the institutions of the logic graph. Each logic, morphism and comorphism in the logic graph has a unique ***name***, which is needed for referring to it.[3]

We will assume that the logic graph comes with a ***default logic*** (which for the purposes of HetCasl is the institution underlying Casl). We also will assume that some of the subinstitution comorphisms in the logic graph are marked as (default) ***logic inclusions***. However, between a given pair of logics, at most one logic inclusion is allowed. The source logic of a logic inclusion is said to be a ***sublogic*** of the target logic. The logic inclusions are subject to a ***coherence condition***: given two paths of inclusions between two logics, there must be a comorphism modification between the composites of the paths.[4]

Similarly, we assume that some of the institution morphisms are marked as (default) ***logic projections***, again with the proviso that between a given pair of logics, at most one logic projection is allowed, and also with a coherence condition similar to that of logic inclusions.

A subset of the logics of the logic graph is marked as ***main logics***. Each main logic comes with an associated set of sublogics.

We further assume an (associative, symmetric, idempotent) partial ***union*** operation on the logics of the logic graph. If the union of two logics is defined, we require that both logics are included in

[2]The dual notion, subinstitution morphism, does not cover typical examples, e.g. the inclusion of equational algebra into first-order logic.

[3]The logic graph is implicitly extended with identities and compositions, yielding a 2-category of morphisms and a 2-category of comorphisms.

[4]This ensures that between two given logics in the 2-category of comorphisms, there is only one logic inclusion up to connectedness via 2-cells. Note that 2-cells are factorized out in the Grothendieck construction below.

the union via logic inclusions, and that the union is minimal (w.r.t. the sublogic relation) with this property. With the help of this binary union it is easy to define unions of finite lists of logics.

For proof-theoretic purposes, it is also required that each logic in the logic graph can be mapped (via some comorphism in the logic graph) to a logic equipped with a proof system, such that this mapping preserves and reflects semantic consequence.

The ***Grothendieck logic***[5] of the logic graph puts all signatures of all involved logics side by side (hence, Grothendieck signatures are pairs consisting of a logic and a signature in that logic). A signature morphism in this large realm of signatures consists of an intra-institution signature morphism plus an inter-institution translation (along some institution morphism or comorphism). Sentences, models and satisfaction for a signature of the Grothendieck logic are just the sentences, models and satisfaction of that signature in the respective logic. Translation of sentences and models is given by composing the intra-institution translation induced by the signature morphism with the inter-institution translation given by the institution morphism or comorphism.

The (co)morphism modifications in the logic graph lead to identification of certain signature morphism in the Grothendieck logic (this concerns signature morphisms that are conceptually "the same", and in particular are known to have identical induced sentence and model translations).

A ***signature inclusion*** in the Grothendieck logic is a signature morphism that consists of an intra-institution inclusion and a logic inclusion. The *union* of two signatures in the Grothendieck logic is constructed by translating the two signatures in the union of the underlying logics, and uniting them there (note that either of these steps may be undefined, leading to undefinedness of the signature union in the Grothendieck logic).

Some logics in the logic graph may be marked as ***process logics***. Each process logic has an associated ***data logic***, which is required to be included in the process logic by means of a logic inclusion.

A.2 Heterogeneous Constructs

This chapter indicates the abstract and concrete syntax of the constructs of heterogeneous specifications, extending those for CASL specifications. The semantics of a heterogeneous specification consists of a signature in the Grothendieck logic and a class of models over that signature. It is assumed that for any of the logics in the logic graph, there is an abstract syntax and semantics for basic specifications as well as for symbol lists and mappings.

For an introduction to the form of grammar used here to define the abstract syntax of language constructs, see Appendix A.3, which also provides the grammar defining the abstract syntax of the HETCASL specification language (as an extension of the CASL grammar).

The central slogan is: heterogeneous specification is just ordinary specification over the Grothendieck logic. The rest of this chapter details how this works.

A.2.1 The Current Logic

Within a homogeneous CASL structured specification, the ***current signature*** (also called local environment) may vary. Within a heterogeneous structured specification, also the ***current logic*** may vary. Since Grothendieck signatures consist of a logic and an ordinary signature, the current logic may be regarded as part of the local environment. However, there is also a current logic at the level of libraries, and a construct for changing the current logic. This is necessary in order to determine the logic in which the empty local environment (which is the empty signature in the current logic) is formed.

At some places, (implicit) ***coercions*** into the current logic may take place. More precisely, this happens for logic qualifications and data specifications as introduced below. A specification is coerced into the current logic by translating its logic into the current logic using the corresponding

[5] Technically, this construction corresponds to a quotient in the sense of [Mos02a] of a Bi-Grothendieck institution [Mos03] — the latter can be regarded as a Grothendieck institution in the sense of [Dia02] by regarding institution morphisms as spans of comorphisms.

logic inclusion. If there is no logic inclusion between the two logics, the construct involving the coercion is ill-formed.

Note that at other places, implicit logic coercions are induced by the definition of unions of Grothendieck signatures in Sect. A.1.3 above. E.g., the semantics of instantiations of generic specifications in CASL is such that the resulting signature of the instantiation is united with the local environment. When e.g. CASL specification downloaded from a CASL library is referenced in a library written in a CASL extension, this has the effect that the CASL specification is coerced to the logic of the CASL extension.[6]

The local environment of a heterogeneous specification may be translated only along logic inclusions, and may not be affected by other logic translations. This in particular means that translations and reductions involving non-inclusion (co)morphisms may not affect the local environment. Otherwise, the heterogeneous specification is ill-formed.

A.2.2 Heterogeneous Structured Specifications

```
SPEC ::= ... | LOGIC-QUALIFICATION | DATA-SPEC
```

A ***logic qualification*** selects a particular logic. A ***data specification*** is a concise notation for writing the data and process parts of a specification in a process logic. The syntax of CASL ***symbol lists*** and ***symbol mappings*** is extended in HETCASL in such a way that also inter-logic translations, reductions, fitting maps and views are allowed. The remaining CASL structuring constructs are available unchanged in HETCASL, but now with a heterogeneous meaning. Revealings and local specifications must be homogeneous, however. The semantics of basic specifications is determined by the semantics of basic specifications for the current logic.

Logic Qualifications

```
LOGIC-QUALIFICATION ::= logic-qual LOGIC SPEC
```

A logic qualification is written:

logic *L SP*

L must denote a logic in the logic graph. The specification *SP* gets the empty signature for that logic as local environment (this is similar to closed specifications). The result is then coerced into the enclosing current logic.

Logics

```
LOGIC        ::= SIMPLE-LOGIC | SUBLOGIC
SIMPLE-LOGIC ::= simple-logic LOGIC-NAME
SUBLOGIC     ::= sublogic LOGIC-NAME LOGIC-NAME
LOGIC-NAME   ::= SIMPLE-ID
```

A `SIMPLE-LOGIC` is written:

LN

LN must be the name of a main logic in the logic graph.

A `SUBLOGIC` is written

LN_1 . LN_2

[6] Still open is the question what happens with the actual parameters: Are they automatically parsed in the logic of the parameterized specification, or is it the responsibility of the user to ensure this?

LN_1 and LN_2 must be a logic names in the logic graph, such that LN_1 is a main logic and LN_2 is a sublogic of LN_1.

Data Specifications

```
DATA-SPEC ::= data-spec SPEC SPEC
```

A data specification is written:

data SP_1 SP_2

The current logic is required to be a process logic. SP_1 gets as local environment the empty signature in the data logic of the current logic. The resulting signature is then coerced into the current logic, and the result of this coercion is added to the local environment for SP_2.

Institution Morphisms and Comorphisms

The same syntax is used for both institution morphisms and comorphisms. It is determined by the context whether a morphism or a comorphism is needed. In the sequel, we will sometimes use 'morphism' when both a morphism or a comorphism can be meant.

```
MORPHISM           ::= NAMED-MORPHISM | QUALIFIED-MORPHISM
                     | ANONYMOUS-MORPHISM | DEFAULT-MORPHISM
NAMED-MORPHISM     ::= named-mor MORPHISM-NAME
QUALIFIED-MORPHISM ::= qual-mor MORPHISM-NAME LOGIC LOGIC
ANONYMOUS-MORPHISM ::= anonymous-mor LOGIC LOGIC
DEFAULT-MORPHISM   ::= default-mor LOGIC
MORPHISM-NAME      ::= SIMPLE-ID
```

A named morphism `NAMED-MORPHISM` is written

MN

MN must be the name of an institution morphism or comorphism in the logic graph.
A qualified morphism `QUALIFIED-MORPHISM` is written

$MN : LN_1 \to LN_2$

The sign '$\to$' is input as '`->`'.
LN_1 and LN_2 must be names of logics in the logic graph, and MN must be the name of an institution morphism or comorphism in the logic graph, with source LN_1 and target LN_2.
An anonymous morphism `ANONYMOUS-MORPHISM` is written

$LN_1 \to LN_2$

LN_1 and LN_2 must be names of logics in the logic graph, and there must be a unique institution morphism or comorphism in the logic graph having source LN_1 and target LN_2.
An default (inclusion or projection) morphism `DEFAULT-MORPHISM` is written

$\to L$

L must be the name of a logic in the logic graph. If the enclosing construct requires an institution comorphism, there must be a (necessarily unique) logic inclusion from the source logic (as determined by the enclosing construct) to L. If the enclosing construct requires an institution morphism, there must be a (necessarily unique) logic projection from the source logic (as determined by the enclosing construct) to L.

Symbol Lists

```
HET-SYMB-ITEMS  ::= HOM-SYMB-ITEMS | LOGIC-REDUCTION
HOM-SYMB-ITEMS  ::= hom-symb-items SYMB-ITEMS*
LOGIC-REDUCTION ::= logic-reduction MORPHISM
```

A heterogeneous symbol list `HET-SYMB-ITEMS*` denotes a signature morphism in the Grothendieck logic. Each `HET-SYMB-ITEMS` denotes such a signature morphism, and the signature morphism for a `HET-SYMB-ITEMS*` is obtained by composing all these signature morphisms. The composition may involve both homogeneous and heterogeneous components, e.g. as follows:

$$(L_1, \Sigma_1) \supseteq (L_1, \Sigma_2) \mapsto (L_2, \Phi(\Sigma_2)) \supseteq (L_2, \Sigma_3) \mapsto (L_3, \Phi'(\Sigma_3)) \ldots,$$

where the "$\mapsto$" components denote institution morphisms and the "$\supseteq$" components denote intra-institution signature inclusions. Each `HET-SYMB-ITEMS` gets a required target signature, which initially is the signature of the specification of the enclosing `REDUCTION`, and then is the source signature of the Grothendieck signature morphism constructed from the preceding list of `HET-SYMB-ITEMS`.

A logic reduction `LOGIC-REDUCTION` is written:

logic *MOR*

MOR must determine an institution morphism. The source logic of the institution morphism must match the required target signature as determined by the list of preceding `HET-SYMB-ITEMS`. The institution morphism contributes to the Grothendieck signature morphism denoted by the enclosing symbol list by mapping to its target logic. The resulting signature is the new required target signature.

Note that institution morphisms are defined in a way that models are mapped along their signature translation. The signature translation of the morphism is analogous to the signature reduction as determined by a homogeneous `SYMB-ITEMS*`, and the model translation of the morphism is analogous the model reduction as determined by a homogeneous `SYMB-ITEMS*`.

Reductions

The abstract syntax of reductions is changed as follows:

```
REDUCTION   ::= reduction SPEC RESTRICTION
RESTRICTION ::= HIDDEN | REVEALED
HIDDEN      ::= hidden HET-SYMB-ITEMS+
REVEALED    ::= revealed SYMB-MAP-ITEMS+
```

In this way, heterogeneous reductions can be formed. Heterogeneous symbol lists are not allowed within revealings (i.e. revealings are always required to be homogeneous).

Symbol Mappings

```
HET-SYMB-MAP-ITEMS ::= HOM-SYMB-ITEMS | LOGIC-TRANSLATION
HOM-SYMB-MAP-ITEMS ::= hom-symb-map-items SYMB-MAP-ITEMS*
LOGIC-TRANSLATION  ::= logic-translation MORPHISM
```

A heterogeneous symbol mapping `HET-SYMB-MAP-ITEMS*` denotes a signature morphism in the Grothendieck logic. Each `HET-SYMB-MAP-ITEMS` denotes such a signature morphism, and the signature morphism for a `HET-SYMB-MAP-ITEMS*` is obtained by composing all these signature morphisms. The composition may involve both homogeneous and heterogeneous components, e.g. as follows:

$$(L_1, \Sigma_1) \rightarrow (L_1, \Sigma_2) \mapsto (L_2, \Phi(\Sigma_2)) \rightarrow (L_2, \Sigma_3) \mapsto (L_3, \Phi'(\Sigma_3)) \ldots$$

where the "$\mapsto$" components denote institution morphisms and the "$\rightarrow$" components denote intra-institution signature inclusions. Each `HET-SYMB-MAP-ITEMS` gets a required source signature, which

initially is the signature of the (source) specification of the enclosing construct, and then is the source signature of the Grothendieck signature morphism constructed from the preceding list of HET-SYMB-MAP-ITEMS.

A logic translation LOGIC-TRANSLATION is written:

logic *MOR*

MOR must determine an institution comorphism. The source logic of the institution comorphism must match the required source logic as determined by the list of preceding HET-SYMB-MAP-ITEMS. The institution comorphism contributes to the Grothendieck signature morphism denoted by the enclosing symbol mapping by mapping to its target logic. The resulting signature is the new required source signature. Note that institution comorphisms are defined in a way that models are mapped against their signature translation. The signature translation of the comorphism is analogous to the signature translation as determined by a homogeneous SYMB-MAP-ITEMS*, and the model translation of the comorphism is analogous to the model reduction as determined (also in a contravariant way) by a homogeneous SYMB-MAP-ITEMS.

A.2.3 Translations

```
TRANSLATION ::= translation SPEC RENAMING
RENAMING    ::= renaming  HET-SYMB-MAP-ITEMS+
```

In this way, heterogeneous translations can be formed.

A.2.4 Fitting Arguments

```
FIT-SPEC ::= fit-spec SPEC HET-SYMB-MAP-ITEMS*
```

For heterogeneous (i.e. those involving logic translations) fitting maps, as well as for heterogeneous views, the rules determining a unique signature morphism between two given signatures (Sect. I:4.1.3 of the CASL Reference Manual [CoF04]) do not apply. Rather, each homogeneous sub-part of the symbol mapping has to explicitly map all the symbols of the appropriate source signature.

View Definitions

```
VIEW-DEFN ::= view-defn VIEW-NAME GENERICITY VIEW-TYPE HET-SYMB-MAP-ITEMS*
```

See the remark about heterogeneous fitting maps above.

A.2.5 Heterogeneous Architectural Specifications

The syntax and semantics of architectural specifications remains as for CASL, except that the underlying logic is the Grothendieck logic. Like for structured specifications above, the syntax and semantics of fitting maps has changed:

```
FIT-ARG-UNIT ::= fit-arg-unit UNIT-TERM HET-SYMB-MAP-ITEMS*
```

A.2.6 Heterogeneous Specification Libraries

There is one new construct at the level of specification libraries.

```
LIB-ITEM        ::= ... | LOGIC-SELECTION
LOGIC-SELECTION ::= logic-select LOGIC
```

A logic selection is written:

logic *L*

L must denote a logic in the logic graph, which is used as current logic for the subsequent library items (until the next `LOGIC-SELECTION`). The selection of the current logic does not affect downloads from other libraries. Vice versa, downloads (as well as other library items that are not logic selections) can change the current logic only locally. That is, the current logic remains unchanged for the next library item (until a logic selection occurs).

A.3 Abstract Syntax

The *abstract syntax* is central to the definition of a formal language. It stands between the concrete representations of documents, such as marks on paper or images on screens, and the abstract entities, semantic relations, and semantic functions used for defining their meaning.

The abstract syntax has the following objectives:

- to identify and separately name the abstract syntactic entities;
- to simplify and unify underlying concepts, putting like things with like, and reducing unnecessary duplication.

There are many possible ways of constructing an abstract syntax, and the choice of form is a matter of judgement, taking into account the somewhat conflicting aims of simplicity and economy of semantic definition.

The abstract syntax is presented as a set of production rules in which each sort of entity is defined in terms of its subsorts:

```
SOME-SORT       ::=  SUBSORT-1  |  ...  |  SUBSORT-n
```

or in terms of its constructor and components:

```
SOME-CONSTRUCT  ::=  some-construct COMPONENT-1 ... COMPONENT-n
```

The productions form a context-free grammar; algebraically, the nonterminal symbols of the grammar correspond to sorts (of trees), and the terminal symbols correspond to constructor operations. The notation `COMPONENT*` indicates repetition of `COMPONENT` any number of times; `COMPONENT+` indicates repetition at least once. (These repetitions could be replaced by auxiliary sorts and constructs, after which it would be straightforward to transform the grammar into a CASL `FREE-DATATYPE` specification.)

The context conditions for well-formedness of specifications are not determined by the grammar (these are considered as part of semantics).

The grammar here has the property that there is a sort for each construct (although an exception is made for constant constructs with no components). Appendix A.4 provides an abbreviated grammar defining the same abstract syntax. It was obtained by eliminating each sort that corresponds to a single construct, when this sort occurs only once as a subsort of another sort.

The following nonterminal symbol corresponds to the CASL syntax, and are left unspecified here: `SIMPLE-ID`. The grammars are given as extensions of the corresponding grammars for the CASL syntax, see part II of of the CASL Reference Manual [CoF04].

A.3.1 Structured Specifications

```
SPEC                 ::= ... | LOGIC-QUALIFICATION | DATA-SPEC

LOGIC-QUALIFICATION  ::= logic-qual LOGIC SPEC

LOGIC                ::= SIMPLE-LOGIC | SUBLOGIC
SIMPLE-LOGIC         ::= simple-logic LOGIC-NAME
SUBLOGIC             ::= sublogic LOGIC-NAME LOGIC-NAME
```

```
LOGIC-NAME            ::= SIMPLE-ID

DATA-SPEC             ::= data-spec SPEC SPEC

MORPHISM              ::= NAMED-MORPHISM | QUALIFIED-MORPHISM
                        | ANONYMOUS-MORPHISM | DEFAULT-MORPHISM
NAMED-MORPHISM        ::= named-mor MORPHISM-NAME
QUALIFIED-MORPHISM    ::= qual-mor MORPHISM-NAME LOGIC LOGIC
ANONYMOUS-MORPHISM    ::= anonymous-mor LOGIC LOGIC
DEFAULT-MORPHISM      ::= default-mor LOGIC

MORPHISM-NAME         ::= SIMPLE-ID

HET-SYMB-ITEMS        ::= HOM-SYMB-ITEMS | LOGIC-REDUCTION
HOM-SYMB-ITEMS        ::= hom-symb-items SYMB-ITEMS*
LOGIC-REDUCTION       ::= logic-reduction MORPHISM

HET-SYMB-MAP-ITEMS    ::= HOM-SYMB-ITEMS | LOGIC-TRANSLATION
HOM-SYMB-MAP-ITEMS    ::= hom-symb-map-items SYMB-MAP-ITEMS*
LOGIC-TRANSLATION     ::= logic-translation MORPHISM


RESTRICTION  ::= HIDDEN | REVEALED
HIDDEN       ::= hidden HET-SYMB-ITEMS+
REVEALED     ::= revealed SYMB-MAP-ITEMS+

RENAMING     ::= renaming  HET-SYMB-MAP-ITEMS+

FIT-ARG      ::= FIT-SPEC | FIT-VIEW
FIT-SPEC     ::= fit-spec SPEC HET-SYMB-MAP-ITEMS*
FIT-VIEW     ::= fit-view VIEW-NAME FIT-ARG*

VIEW-DEFN    ::= view-defn VIEW-NAME GENERICITY VIEW-TYPE HET-SYMB-MAP-ITEMS*

FIT-ARG-UNIT ::= fit-arg-unit UNIT-TERM HET-SYMB-MAP-ITEMS*
```

A.3.2 Specification Libraries

```
LIB-ITEM              ::= ... | LOGIC-SELECTION
LOGIC-SELECTION       ::= logic-select LOGIC
```

A.4 Abbreviated Abstract Syntax

A.4.1 Structured Specifications

```
SPEC              ::= ... | logic-qual LOGIC SPEC
                        | data-spec SPEC SPEC

LOGIC             ::= simple-logic LOGIC-NAME
                    | sublogic LOGIC-NAME LOGIC-NAME

LOGIC-NAME        ::= SIMPLE-ID

MORPHISM          ::= named-mor MORPHISM-NAME
                    | qual-mor MORPHISM-NAME LOGIC LOGIC
                    | anonymous-mor LOGIC LOGIC
```

```
                    | default-mor LOGIC

MORPHISM-NAME      ::= SIMPLE-ID

HET-SYMB-ITEMS     ::= hom-symb-items SYMB-ITEMS*
                     | logic-reduction MORPHISM

HET-SYMB-MAP-ITEMS ::= hom-symb-map-items SYMB-MAP-ITEMS*
                     | logic-translation MORPHISM

RESTRICTION  ::= hidden HET-SYMB-ITEMS+
               | revealed SYMB-MAP-ITEMS+

RENAMING     ::= renaming  HET-SYMB-MAP-ITEMS+

FIT-ARG      ::= fit-spec SPEC HET-SYMB-MAP-ITEMS*
               | fit-view VIEW-NAME FIT-ARG*

VIEW-DEFN    ::= view-defn VIEW-NAME GENERICITY VIEW-TYPE HET-SYMB-MAP-ITEMS*

FIT-ARG-UNIT ::= fit-arg-unit UNIT-TERM HET-SYMB-MAP-ITEMS*
```

A.4.2 Specification Libraries

```
LIB-ITEM     ::= ... | logic-select LOGIC
```

A.5 Concrete Syntax

The concrete syntax of HetCasl is based on concrete syntaxes for basic specifications, symbol lists and symbol mappings for each of the logics in the logic graph.

A parser for HetCasl is available via the Heterogeneous Tool Set (Hets) web page

`http://www.tzi.de/cofi/hets`

It is based upon parsers for basic specifications, symbol lists and symbol mappings for each of the logics in the logic graph.

Sect. A.5.1 below provides a context-free grammar for the HetCasl input syntax in terms of changes and additions to the context-free grammar of Casl (see Chap. II:3 of [CoF04]). It has been derived from the 'abbreviated' abstract syntax grammar in Appendix A.4.

The lexical syntax, comments and annotations, the literal syntax, and the display format is identical that of Casl, resp. that of the respective logic in the logic graph.

A.5.1 Context-Free Syntax

The following meta-notation for context-free grammars is used:

Nonterminal symbols are written as uppercase words, possibly hyphenated, e.g., `SORT`, `BASIC-SPEC`.

Terminal symbols are written as lowercase words, e.g. `free`, `assoc`.

Sequences of symbols are written with spaces between the symbols. The empty sequence is denoted by the reserved nonterminal symbol `EMPTY`.

Optional symbols are underlined, e.g. `end`, `;`. This is used also for the optional plural 's' at the end of some lowercase words used as terminal symbols, e.g. `sorts`.

Repetitions are indicated by ellipsis '`...`', e.g. `MIXFIX...MIXFIX` denotes one or more occurrences of `MIXFIX`, and `[SPEC]...[SPEC]` denotes one or more occurrences of `[SPEC]`. Repetitions often involve separators, e.g. `SORT,...,SORT` denotes one or more occurrences of `SORT` separated by '`,`'.

Alternative sequences are separated by vertical bars, e.g. `idem | unit TERM` where the alternatives are `idem` and `unit TERM`.

Production rules are written with the nonterminal symbol followed by '`::=`', followed by one or more alternatives. When a production extends a previously-given production for the same nonterminal symbol, this is indicated by writing '`...`' as its first alternative.

Start symbols are not specified.

A.5.2 Structured Specifications

```
SPEC                  ::= ... | logic LOGIC : GROUP-SPEC
                            | data GROUP-SPEC SPEC

LOGIC               ::= LOGIC-NAME
                      | LOGIC-NAME . LOGIC-NAME

LOGIC-NAME            ::= SIMPLE-ID

MORPHISM              ::= MORPHISM-NAME
                        | MORPHISM-NAME : LOGIC -> LOGIC
                        | LOGIC -> LOGIC
                        | -> LOGIC

MORPHISM-NAME         ::= SIMPLE-ID

HET-SYMB-ITEMS        ::= SYMB-ITEMS ,..., SYMB-ITEMS
                        | logic MORPHISM

HET-SYMB-MAP-ITEMS    ::= SYMB-MAP-ITEMS ,..., SYMB-MAP-ITEMS
                        | logic MORPHISM


RESTRICTION  ::= hide HET-SYMB-ITEMS ,..., HET-SYMB-ITEMS
               | reveal SYMB-MAP-ITEMS ,..., SYMB-MAP-ITEMS

RENAMING     ::= with HET-SYMB-MAP-ITEMS ,..., HET-SYMB-MAP-ITEMS

FIT-ARG      ::= SPEC fit HET-SYMB-MAP-ITEMS ,..., HET-SYMB-MAP-ITEMS
               | SPEC
               | view VIEW-NAME
               | view VIEW-NAME [ FIT-ARG ]...[ FIT-ARG ]

VIEW-DEFN    ::= view VIEW-NAME : VIEW-TYPE end
               | view VIEW-NAME : VIEW-TYPE =
                      HET-SYMB-MAP-ITEMS ,..., HET-SYMB-MAP-ITEMS end
               | view VIEW-NAME SOME-GENERICS : VIEW-TYPE end
               | view VIEW-NAME SOME-GENERICS : VIEW-TYPE =
                      HET-SYMB-MAP-ITEMS ,..., HET-SYMB-MAP-ITEMS end


FIT-ARG-UNIT ::= UNIT-TERM
               | UNIT-TERM fit HET-SYMB-MAP-ITEMS ,..., HET-SYMB-MAP-ITEMS
```

A.5.3 Specification Libraries

```
LIB-ITEM              ::= ... | logic LOGIC
```

A.5.4 Lexical Syntax

For parsing outside basic specifications, symbol lists and symbol mappings, the lexical syntax is almost identical to that of CASL. There are two additional keywords:

```
logic data
```

and the keywords specific to the CASL logic are removed from the list of keywords.

For parsing basic specifications, symbol lists and symbol mappings in a logic, the lexical syntax of that logic is used.

Appendix B

Selected Code of the Heterogeneous Tool Set HETS

B.1 Haskell Code of Type Class Logic

```
{-# OPTIONS -fallow-undecidable-instances #-}
{- |
Module      :  /repository/HetCATS/Logic/Logic.hs
Copyright   :  (c) Till Mossakowski, and Uni Bremen 2002-2003
Licence     :  similar to LGPL, see HetCATS/LICENCE.txt or LIZENZ.txt

Maintainer  :  till@tzi.de
Stability   :  provisional
Portability :  non-portable (various -fglasgow-exts extensions)

   Provides data structures for logics (with symbols). Logics are
class with an "identitiy" type (usually interpreted
   by a singleton set) which serves to treat logics as
   data. All the functions in the type class take the
   identity as first argument in order to determine the logic.

   For logic (co)morphisms see Comorphism.hs

   References:

   J. A. Goguen and R. M. Burstall
   Institutions: Abstract Model Theory for Specification and
     Programming
   JACM 39, p. 95--146, 1992
   (general notion of logic - model theory only)

   J. Meseguer
   General Logics
   Logic Colloquium 87, p. 275--329, North Holland, 1989
   (general notion of logic - also proof theory;
    notion of logic representation, called map there)

   T. Mossakowski:
```

```
   Specification in an arbitrary institution with symbols
   14th WADT 1999, LNCS 1827, p. 252--270
   (treatment of symbols and raw symbols, see also CASL semantics)

   T. Mossakowski, B. Klin:
   Institution Independent Static Analysis for CASL
   15h WADT 2001, LNCS 2267, p. 221-237, 2002.
   (what is needed for static anaylsis)

   S. Autexier and T. Mossakowski
   Integrating HOLCASL into the Development Graph Manager MAYA
   FroCoS 2002, to appear
   (interface to provers)

   Todo:
   ATerm, XML
   Weak amalgamability
   Metavars
   raw symbols are now symbols, symbols are now signature symbols
   provide both signature symbol set and symbol set of a signature
-}

module Logic.Logic (module Logic.Logic, module Logic.Languages) where

import Common.Id
import Common.GlobalAnnotations
import Common.Lib.Set
import Common.Lib.Map
import Common.Lib.Graph
import Common.Lib.Pretty
import Common.AnnoState
import Common.Result
import Common.AS_Annotation
import Common.Print_AS_Annotation
import Logic.Languages
import Logic.Prover -- for one half of class Sentences

import Common.PrettyPrint

import Data.Dynamic
import Common.DynamicUtils

-- for Conversion to ATerms
import Common.ATerm.Lib -- (ATermConvertible)

-- passed to ensures_amalgamability
import Common.Amalgamate

import Common.Taxonomy
import Taxonomy.MMiSSOntology (MMiSSOntology)


-- Categories are given by a quotient,
```

```
-- i.e. we need equality
-- Should we allow arbitrary composition graphs and build paths?

class (PrintLaTeX a, Typeable a, ATermConvertible a) => PrintTypeConv a
class (Eq a, PrintTypeConv a) => EqPrintTypeConv a

instance (PrintLaTeX a, Typeable a, ATermConvertible a) => PrintTypeConv a
instance (Eq a, PrintTypeConv a) => EqPrintTypeConv a

class (Language lid, Eq sign, Eq morphism)
    => Category lid sign morphism | lid -> sign, lid -> morphism where
         ide :: lid -> sign -> morphism
         comp :: lid -> morphism -> morphism -> Result morphism
           -- diagrammatic order
         dom, cod :: lid -> morphism -> sign
         legal_obj :: lid -> sign -> Bool
         legal_mor :: lid -> morphism -> Bool

-- abstract syntax, parsing and printing

class (Language lid, PrintTypeConv basic_spec,
       EqPrintTypeConv symb_items,
       EqPrintTypeConv symb_map_items)
    => Syntax lid basic_spec symb_items symb_map_items
        | lid -> basic_spec, lid -> symb_items,
          lid -> symb_map_items
      where
         -- parsing
         parse_basic_spec :: lid -> Maybe(AParser st basic_spec)
         parse_symb_items :: lid -> Maybe(AParser st symb_items)
         parse_symb_map_items :: lid -> Maybe(AParser st symb_map_items)
         -- default implementations
         parse_basic_spec _ = Nothing
         parse_symb_items _ = Nothing
         parse_symb_map_items _ = Nothing

-- sentences (plus prover stuff and "symbol" with "Ord" for efficient lookup)

class (Category lid sign morphism, Ord sentence,
       Ord symbol,
       PrintTypeConv sign, PrintTypeConv morphism,
       PrintTypeConv sentence, PrintTypeConv symbol,
       Eq proof_tree, Show proof_tree, ATermConvertible proof_tree,
       Typeable proof_tree)
    => Sentences lid sentence proof_tree sign morphism symbol
        | lid -> sentence, lid -> sign, lid -> morphism,
          lid -> symbol, lid -> proof_tree
      where
         -- sentence translation
      map_sen :: lid -> morphism -> sentence -> Result sentence
      map_sen l _ _ = statErr l "map_sen"
         -- simplification of sentences (leave out qualifications)
      simplify_sen :: lid -> sign -> sentence -> sentence
```

```
      simplify_sen _ _ = id  -- default implementation
         -- parsing of sentences
      parse_sentence :: lid -> Maybe (AParser st sentence)
      parse_sentence _ = Nothing
           -- print a sentence with comments
      print_named :: lid -> GlobalAnnos -> Named sentence -> Doc
      print_named _ = printText0
      sym_of :: lid -> sign -> Set symbol
      symmap_of :: lid -> morphism -> EndoMap symbol
      sym_name :: lid -> symbol -> Id
      provers :: lid -> [Prover sign sentence proof_tree symbol]
      provers _ = []
      cons_checkers :: lid -> [ConsChecker sign sentence morphism proof_tree]
      cons_checkers _ = []
      consCheck :: lid -> Theory sign sentence ->
                      morphism -> [Named sentence] -> Result (Maybe Bool)
      consCheck l _ _ _ = statErr l "consCheck"

-- static analysis
statErr :: (Language lid, Monad m) => lid -> String -> m a
statErr lid str = fail ("Logic." ++ str ++ " nyi for: " ++ language_name lid)

class ( Syntax lid basic_spec symb_items symb_map_items
      , Sentences lid sentence proof_tree sign morphism symbol
      , Ord raw_symbol, PrintLaTeX raw_symbol, Typeable raw_symbol)
    => StaticAnalysis lid
        basic_spec sentence proof_tree symb_items symb_map_items
        sign morphism symbol raw_symbol
        | lid -> basic_spec, lid -> sentence, lid -> symb_items,
          lid -> symb_map_items, lid -> proof_tree,
          lid -> sign, lid -> morphism, lid -> symbol, lid -> raw_symbol
      where
         -- static analysis of basic specifications and symbol maps
         basic_analysis :: lid ->
                           Maybe((basic_spec,  -- abstract syntax tree
                            sign,   -- efficient table for env signature
                            GlobalAnnos) ->   -- global annotations
                           Result (basic_spec,sign,sign,[Named sentence]))
                           -- the resulting bspec has analyzed axioms in it
                           -- sign's: sigma_local, sigma_complete, i.e.
                           -- the second output sign united with the input sign
                           -- should yield the first output sign
                           -- the second output sign is the accumulated sign
         -- default implementation
         basic_analysis _ = Nothing

         sign_to_basic_spec :: lid -> sign -> [Named sentence] -> basic_spec
         stat_symb_map_items ::
             lid -> [symb_map_items] -> Result (EndoMap raw_symbol)
         stat_symb_map_items _ _ = fail "Logic.stat_symb_map_items"
         stat_symb_items :: lid -> [symb_items] -> Result [raw_symbol]
         stat_symb_items l _ = statErr l "stat_symb_items"
         -- architectural sharing analysis
```

```
        ensures_amalgamability :: lid ->
             ([CASLAmalgOpt],        -- the program options
              Diagram sign morphism, -- the diagram to be analyzed
              [(Node, morphism)],    -- the sink
              Diagram String String) -- the descriptions of nodes and edges
                 -> Result Amalgamates
        ensures_amalgamability l _ = statErr l "ensures_amalgamability"
        -- symbols and symbol maps
        symbol_to_raw :: lid -> symbol -> raw_symbol
        id_to_raw :: lid -> Id -> raw_symbol
        matches :: lid -> symbol -> raw_symbol -> Bool

        -- operations on signatures and morphisms
        empty_signature :: lid -> sign
        signature_union :: lid -> sign -> sign -> Result sign
        signature_union l _ _ = statErr l "signature_union"
        morphism_union :: lid -> morphism -> morphism -> Result morphism
        morphism_union l _ _ = statErr l "morphism_union"
        final_union :: lid -> sign -> sign -> Result sign
        final_union l _ _ = statErr l "final_union"
          -- see CASL reference manual, III.4.1.2
        is_subsig :: lid -> sign -> sign -> Bool
        inclusion :: lid -> sign -> sign -> Result morphism
        inclusion l _ _ = statErr l "inclusion"
        generated_sign, cogenerated_sign ::
            lid -> Set symbol -> sign -> Result morphism
        generated_sign l _ _ = statErr l "generated_sign"
        cogenerated_sign l _ _ = statErr l "cogenerated_sign"
        induced_from_morphism ::
            lid -> EndoMap raw_symbol -> sign -> Result morphism
        induced_from_morphism l _ _ = statErr l "induced_from_morphism"
        induced_from_to_morphism ::
            lid -> EndoMap raw_symbol -> sign -> sign -> Result morphism
        induced_from_to_morphism l _ _ _ =
            statErr l "induced_from_to_morphism"
        -- generate taxonomy from theory
        theory_to_taxonomy :: lid
                           -> TaxoGraphKind
                           -> MMiSSOntology
                           -> sign -> [Named sentence]
                           -> Result MMiSSOntology
        theory_to_taxonomy l _ _ _ _ = statErr l "theory_to_taxonomy"

-- sublogics

class (Ord l, Show l) => LatticeWithTop l where
  meet, join :: l -> l -> l
  top :: l

-- a dummy instance
instance LatticeWithTop () where
  meet _ _ = ()
  join _ _ = ()
```

```
  top = ()

-- logics

class (StaticAnalysis lid
        basic_spec sentence proof_tree symb_items symb_map_items
        sign morphism symbol raw_symbol,
       LatticeWithTop sublogics, ATermConvertible sublogics,
       Typeable sublogics)
    => Logic lid sublogics
        basic_spec sentence symb_items symb_map_items
        sign morphism symbol raw_symbol proof_tree
        | lid -> sublogics, lid -> basic_spec, lid -> sentence,
          lid -> symb_items, lid -> symb_map_items, lid -> proof_tree,
          lid -> sign, lid -> morphism, lid ->symbol, lid -> raw_symbol
          where

        -- for a process logic, return its data logic
        data_logic :: lid -> Maybe AnyLogic
        data_logic _ = Nothing

        sublogic_names :: lid -> sublogics -> [String]
        sublogic_names lid _ = [language_name lid]
            -- the first name is the principal name
        all_sublogics :: lid -> [sublogics]
        all_sublogics _ = [top]

        is_in_basic_spec :: lid -> sublogics -> basic_spec -> Bool
        is_in_basic_spec _ _ _ = False
        is_in_sentence :: lid -> sublogics -> sentence -> Bool
        is_in_sentence _ _ _ = False
        is_in_symb_items :: lid -> sublogics -> symb_items -> Bool
        is_in_symb_items _ _ _ = False
        is_in_symb_map_items :: lid -> sublogics -> symb_map_items -> Bool
        is_in_symb_map_items _ _ _ = False
        is_in_sign :: lid -> sublogics -> sign -> Bool
        is_in_sign _ _ _ = False
        is_in_morphism :: lid -> sublogics -> morphism -> Bool
        is_in_morphism _ _ _ = False
        is_in_symbol :: lid -> sublogics -> symbol -> Bool
        is_in_symbol _ _ _ = False

        min_sublogic_basic_spec :: lid -> basic_spec -> sublogics
        min_sublogic_basic_spec _ _ = top
        min_sublogic_sentence :: lid -> sentence -> sublogics
        min_sublogic_sentence _ _ = top
        min_sublogic_symb_items :: lid -> symb_items -> sublogics
        min_sublogic_symb_items _ _ = top
        min_sublogic_symb_map_items :: lid -> symb_map_items -> sublogics
        min_sublogic_symb_map_items _ _ = top
        min_sublogic_sign :: lid -> sign -> sublogics
        min_sublogic_sign _ _ = top
        min_sublogic_morphism :: lid -> morphism -> sublogics
```

```
         min_sublogic_morphism _ _ = top
         min_sublogic_symbol :: lid -> symbol -> sublogics
         min_sublogic_symbol _ _ = top
         proj_sublogic_basic_spec :: lid -> sublogics
                                  -> basic_spec -> basic_spec
         proj_sublogic_basic_spec _ _ b = b
         proj_sublogic_symb_items :: lid -> sublogics
                                  -> symb_items -> Maybe symb_items
         proj_sublogic_symb_items _ _ _ = Nothing
         proj_sublogic_symb_map_items :: lid -> sublogics
                                      -> symb_map_items -> Maybe symb_map_items
         proj_sublogic_symb_map_items _ _ _ = Nothing
         proj_sublogic_sign :: lid -> sublogics -> sign -> sign
         proj_sublogic_sign _ _ s = s
         proj_sublogic_morphism :: lid -> sublogics -> morphism -> morphism
         proj_sublogic_morphism _ _ m = m
         proj_sublogic_epsilon :: lid -> sublogics -> sign -> morphism
         proj_sublogic_epsilon li _ s = ide li s
         proj_sublogic_symbol :: lid -> sublogics -> symbol -> Maybe symbol
         proj_sublogic_symbol _ _ _ = Nothing

         top_sublogic :: lid -> sublogics
         top_sublogic _ = top

------------------------------------------------------------------
-- Derived functions
------------------------------------------------------------------

empty_theory :: StaticAnalysis lid
        basic_spec sentence proof_tree symb_items symb_map_items
        sign morphism symbol raw_symbol =>
        lid -> Theory sign sentence
empty_theory lid = (empty_signature lid,[])

------------------------------------------------------------------
-- Existential type covering any logic
------------------------------------------------------------------

data AnyLogic = forall lid sublogics
        basic_spec sentence symb_items symb_map_items
        sign morphism symbol raw_symbol proof_tree .
        Logic lid sublogics
         basic_spec sentence symb_items symb_map_items
         sign morphism symbol raw_symbol proof_tree =>
        Logic lid

instance Show AnyLogic where
  show (Logic lid) = language_name lid
instance Eq AnyLogic where
  Logic lid1 == Logic lid2 = language_name lid1 == language_name lid2

tyconAnyLogic :: TyCon
tyconAnyLogic = mkTyCon "Logic.Logic.AnyLogic"
```

```
instance Typeable AnyLogic where
  typeOf _ = mkTyConApp tyconAnyLogic []

------------------------------------------------------------------
-- Typeable instances
------------------------------------------------------------------

namedTc :: TyCon
namedTc = mkTyCon "Common.AS_Annotation.Named"

instance Typeable s => Typeable (Named s) where
  typeOf s = mkTyConApp namedTc [typeOf ((undefined :: Named a -> a) s)]

setTc :: TyCon
setTc = mkTyCon "Common.Lib.Set.Set"

instance Typeable a => Typeable (Set a) where
  typeOf s = mkTyConApp setTc [typeOf ((undefined:: Set a -> a) s)]

mapTc :: TyCon
mapTc = mkTyCon "Common.Lib.Map.Map"

instance (Typeable a, Typeable b) => Typeable (Map a b) where
  typeOf m = mkTyConApp mapTc [typeOf ((undefined :: Map a b -> a) m),
                               typeOf ((undefined :: Map a b -> b) m)]

{- class hierarchy:
                               Language
                ----------/
   Category
      |                   /
   Sentences      Syntax
      \            /
      StaticAnalysis (no sublogics)
            \
             \
            Logic

-}
```

B.2 Haskell Code of Type Class Comorphism

```
{-|

Module      :  /repository/HetCATS/Logic.Comorphism.hs
Copyright   :  (c) Till Mossakowski, Uni Bremen 2002-2004
Licence     :  similar to LGPL, see HetCATS/LICENCE.txt or LIZENZ.txt

Maintainer  :  hets@tzi.de
Stability   :  provisional
Portability :  non-portable (via Logic)

   Provides data structures for institution comorphisms.
```

```
   These are just collections of
   functions between (some of) the types of logics.

-}

{-   References: see Logic.hs

   Todo:
   Weak amalgamability, also for comorphisms
   comorphism modifications
   comorphisms out of sublogic relationships
   restrictions of comorphisms to sublogics
   morphisms and other translations via spans
-}

module Logic.Comorphism where

import Logic.Logic
import Common.Lib.Set
import Common.Result
import Data.Maybe
import Data.Dynamic
import Common.DynamicUtils
import Common.AS_Annotation (Named, mapNamedM)

class (Language cid,
       Logic lid1 sublogics1
        basic_spec1 sentence1 symb_items1 symb_map_items1
        sign1 morphism1 symbol1 raw_symbol1 proof_tree1,
       Logic lid2 sublogics2
        basic_spec2 sentence2 symb_items2 symb_map_items2
        sign2 morphism2 symbol2 raw_symbol2 proof_tree2) =>
  Comorphism cid
            lid1 sublogics1 basic_spec1 sentence1 symb_items1 symb_map_items1
                sign1 morphism1 symbol1 raw_symbol1 proof_tree1
            lid2 sublogics2 basic_spec2 sentence2 symb_items2 symb_map_items2
                sign2 morphism2 symbol2 raw_symbol2 proof_tree2
             | cid -> lid1, cid -> lid2

  where
    -- source and target logic and sublogic
    -- the source sublogic is the maximal one for which the comorphism works
    -- the target sublogic is the resulting one
    sourceLogic :: cid -> lid1
    sourceSublogic :: cid -> sublogics1
    targetLogic :: cid -> lid2
    targetSublogic :: cid -> sublogics2
    -- finer information of target sublogics corresponding to source sublogics
    mapSublogic :: cid -> sublogics1 -> sublogics2
    -- default implementation
    mapSublogic cid _ = targetSublogic cid
    -- the translation functions are partial
    -- because the target may be a sublanguage
```

```
    -- map_basic_spec :: cid -> basic_spec1 -> Result basic_spec2
    -- cover theoroidal comorphisms as well
    map_sign :: cid -> sign1 -> Result (sign2,[Named sentence2])
    map_theory :: cid -> (sign1,[Named sentence1])
                      -> Result (sign2,[Named sentence2])
    --default implementations
    map_sign cid sign = map_theory cid (sign,[])
    map_theory cid (sign,sens) = do
       (sign',sens') <- map_sign cid sign
       sens'' <- mapM (mapNamedM $ map_sentence cid sign) sens
       return (sign',sens'++sens'')
    map_morphism :: cid -> morphism1 -> Result morphism2
    map_sentence :: cid -> sign1 -> sentence1 -> Result sentence2
          -- also covers semi-comorphisms
          -- with no sentence translation
          -- - but these are spans!
    map_symbol :: cid -> symbol1 -> Set symbol2
    constituents :: cid -> [String]
    -- default implementation
    constituents cid = [language_name cid]

data IdComorphism lid sublogics =
     IdComorphism lid sublogics deriving Show

idComorphismTc :: TyCon
idComorphismTc = mkTyCon "Logic.Comorphism.IdComorphism"

instance Typeable (IdComorphism lid sub) where
  typeOf _ = mkTyConApp idComorphismTc []

instance Logic lid sublogics
        basic_spec sentence symb_items symb_map_items
        sign morphism symbol raw_symbol proof_tree =>
         Language (IdComorphism lid sublogics) where
           language_name (IdComorphism lid sub) =
               case sublogic_names lid sub of
               [] -> error "language_name IdComorphism"
               h : _ -> "id_" ++ language_name lid ++ "." ++ h

instance Logic lid sublogics
        basic_spec sentence symb_items symb_map_items
        sign morphism symbol raw_symbol proof_tree =>
         Comorphism (IdComorphism lid sublogics)
          lid sublogics
          basic_spec sentence symb_items symb_map_items
          sign morphism symbol raw_symbol proof_tree
          lid sublogics
          basic_spec sentence symb_items symb_map_items
          sign morphism symbol raw_symbol proof_tree
         where
           sourceLogic (IdComorphism lid _sub) = lid
           targetLogic (IdComorphism lid _sub) = lid
           sourceSublogic (IdComorphism _lid sub) = sub
```

```
            targetSublogic (IdComorphism _lid sub) = sub
            map_sign _ = \sigma -> return (sigma,[])
            map_morphism _ = return
            map_sentence _ = \_ -> return
            map_symbol _ = single
            constituents _ = []

data CompComorphism cid1 cid2 = CompComorphism cid1 cid2 deriving Show

tyconCompComorphism :: TyCon
tyconCompComorphism = mkTyCon "Logic.Comorphism.CompComorphism"

instance Typeable (CompComorphism cid1 cid2) where
  typeOf _ = mkTyConApp tyconCompComorphism []

instance (Comorphism cid1
            lid1 sublogics1 basic_spec1 sentence1 symb_items1 symb_map_items1
                sign1 morphism1 symbol1 raw_symbol1 proof_tree1
            lid2 sublogics2 basic_spec2 sentence2 symb_items2 symb_map_items2
                sign2 morphism2 symbol2 raw_symbol2 proof_tree2,
          Comorphism cid2
            lid4 sublogics4 basic_spec4 sentence4 symb_items4 symb_map_items4
                sign4 morphism4 symbol4 raw_symbol4 proof_tree4
            lid3 sublogics3 basic_spec3 sentence3 symb_items3 symb_map_items3
                sign3 morphism3 symbol3 raw_symbol3 proof_tree3)
          => Language (CompComorphism cid1 cid2) where
   language_name (CompComorphism cid1 cid2) =
     language_name cid1++";"
     ++language_name cid2


instance (Comorphism cid1
            lid1 sublogics1 basic_spec1 sentence1 symb_items1 symb_map_items1
                sign1 morphism1 symbol1 raw_symbol1 proof_tree1
            lid2 sublogics2 basic_spec2 sentence2 symb_items2 symb_map_items2
                sign2 morphism2 symbol2 raw_symbol2 proof_tree2,
          Comorphism cid2
            lid4 sublogics4 basic_spec4 sentence4 symb_items4 symb_map_items4
                sign4 morphism4 symbol4 raw_symbol4 proof_tree4
            lid3 sublogics3 basic_spec3 sentence3 symb_items3 symb_map_items3
                sign3 morphism3 symbol3 raw_symbol3 proof_tree3)
          => Comorphism (CompComorphism cid1 cid2)
              lid1 sublogics1 basic_spec1 sentence1 symb_items1 symb_map_items1
              sign1 morphism1 symbol1 raw_symbol1 proof_tree1
              lid3 sublogics3 basic_spec3 sentence3 symb_items3 symb_map_items3
              sign3 morphism3 symbol3 raw_symbol3 proof_tree3 where
   sourceLogic (CompComorphism cid1 _) =
     sourceLogic cid1
   targetLogic (CompComorphism _ cid2) =
     targetLogic cid2
   sourceSublogic (CompComorphism cid1 _) =
     sourceSublogic cid1
   targetSublogic (CompComorphism _ cid2) =
```

```
  targetSublogic cid2
map_sentence (CompComorphism cid1 cid2) =
    \si1 se1 ->
      do (si2,_) <- map_sign cid1 si1
         se2 <- map_sentence cid1 si1 se1
         (si2', se2') <- mcoerce (targetLogic cid1) (sourceLogic cid2)
                         "Mapping sentence along comorphism" (si2, se2)
         map_sentence cid2 si2' se2'
map_sign (CompComorphism cid1 cid2) =
    \si1 ->
      do (si2, se2s) <- map_sign cid1 si1
         (si2', se2s') <- mcoerce (targetLogic cid1) (sourceLogic cid2)
                          "Mapping signature along comorphism"(si2, se2s)
         (si3, se3s) <- map_sign cid2 si2'
         se3s' <- mapM (mapNamedM $ map_sentence cid2 si2') se2s'
         return (si3, se3s ++ se3s')

map_theory (CompComorphism cid1 cid2) =
    \ti1 ->
      do ti2 <- map_theory cid1 ti1
         ti2' <- mcoerce (targetLogic cid1) (sourceLogic cid2)
                     "Mapping theory along comorphism" ti2
         map_theory cid2 ti2'

map_morphism (CompComorphism cid1 cid2) = \ m1 ->
    do m2 <- map_morphism cid1 m1
       m3 <- mcoerce (targetLogic cid1) (sourceLogic cid2)
               "Mapping signature morphism along comorphism"m2
       map_morphism cid2 m3

map_symbol (CompComorphism cid1 cid2) = \ s1 ->
      let mycast = fromJust . mcoerce (targetLogic cid1) (sourceLogic cid2)
                               "Mapping symbol along comorphism"
      in unions
             (map (map_symbol cid2 . mycast)
              (toList (map_symbol cid1 s1)))
constituents (CompComorphism cid1 cid2) =
   constituents cid1 ++ constituents cid2
```

B.3 Haskell Code of Grothendieck Logic

```
{-# OPTIONS -fallow-overlapping-instances -fallow-incoherent-instances #-}
{- |
Module      :  /repository/Logic/Grothendieck.hs
Copyright   :  (c) Till Mossakowski, and Uni Bremen 2002-2004
Licence     :  similar to LGPL, see HetCATS/LICENCE.txt or LIZENZ.txt

Maintainer  :  till@tzi.de
Stability   :  provisional
Portability :  non-portable (overlapping instances, dynamics, existentials)


   The Grothendieck logic is defined to be the
```

```
   heterogeneous logic over the logic graph.
   This will be the logic over which the data
   structures and algorithms for specification in-the-large
   are built.

   References:

   R. Diaconescu:
   Grothendieck institutions
   J. applied categorical structures 10, 2002, p. 383-402.

   T. Mossakowski:
   Heterogeneous development graphs and heterogeneous borrowing
   Fossacs 2002, LNCS 2303, p. 326-341

   T. Mossakowski: Foundations of heterogeneous specification
   Submitted

   T. Mossakowski:
   Relating CASL with Other Specification Languages:
        the Institution Level
   Theoretical Computer Science 286, 2002, p. 367-475

   Todo:

-}

module Logic.Grothendieck where

import Logic.Logic
import Logic.Prover
import Logic.Comorphism
import Common.PrettyPrint
import Common.Lib.Pretty
import Common.Lib.Graph
import qualified Common.Lib.Map as Map
import qualified Common.Lib.Set as Set
import Common.Result
import Common.AS_Annotation
import Common.ListUtils
import Data.Dynamic
import Common.DynamicUtils
import qualified Data.List as List
import Data.Maybe
import Control.Monad

------------------------------------------------------------------
--"Grothendieck" versions of the various parts of type class Logic
------------------------------------------------------------------

-- | Grothendieck basic specifications
data G_basic_spec = forall lid sublogics
        basic_spec sentence symb_items symb_map_items
```

```
         sign morphism symbol raw_symbol proof_tree .
        Logic lid sublogics
         basic_spec sentence symb_items symb_map_items
          sign morphism symbol raw_symbol proof_tree =>
  G_basic_spec lid basic_spec

instance Show G_basic_spec where
    show (G_basic_spec _ s) = show s

instance PrettyPrint G_basic_spec where
    printText0 ga (G_basic_spec _ s) = printText0 ga s

-- | Grothendieck sentences
data G_sentence = forall lid sublogics
        basic_spec sentence symb_items symb_map_items
         sign morphism symbol raw_symbol proof_tree .
        Logic lid sublogics
         basic_spec sentence symb_items symb_map_items
          sign morphism symbol raw_symbol proof_tree =>
  G_sentence lid sentence

instance Show G_sentence where
    show (G_sentence _ s) = show s

-- | Grothendieck sentence lists
data G_l_sentence_list = forall lid sublogics
        basic_spec sentence symb_items symb_map_items
         sign morphism symbol raw_symbol proof_tree .
        Logic lid sublogics
         basic_spec sentence symb_items symb_map_items
          sign morphism symbol raw_symbol proof_tree  =>
  G_l_sentence_list lid [Named sentence]

instance Show G_l_sentence_list where
    show (G_l_sentence_list _ s) = show s

instance Eq G_l_sentence_list where
    (G_l_sentence_list i1 nl1) == (G_l_sentence_list i2 nl2) =
      coerce i1 i2 nl1 == Just nl2

eq_G_l_sentence_set :: G_l_sentence_list -> G_l_sentence_list -> Bool
eq_G_l_sentence_set (G_l_sentence_list i1 nl1) (G_l_sentence_list i2 nl2) =
     case coerce i1 i2 nl1 of
       Just nl1' -> Set.fromList nl1' == Set.fromList nl2
       Nothing -> False

-- | Grothendieck signatures
data G_sign = forall lid sublogics
        basic_spec sentence symb_items symb_map_items
         sign morphism symbol raw_symbol proof_tree .
        Logic lid sublogics
         basic_spec sentence symb_items symb_map_items
          sign morphism symbol raw_symbol proof_tree =>
```

```
  G_sign lid sign

tyconG_sign :: TyCon
tyconG_sign = mkTyCon "Logic.Grothendieck.G_sign"
instance Typeable G_sign where
  typeOf _ = mkTyConApp tyconG_sign []

instance Eq G_sign where
  (G_sign i1 sigma1) == (G_sign i2 sigma2) =
     coerce i1 i2 sigma1 == Just sigma2

-- | prefer a faster subsignature test if possible
is_subgsign :: G_sign -> G_sign -> Bool
is_subgsign (G_sign i1 sigma1) (G_sign i2 sigma2) =
    maybe False (is_subsig i1 sigma1) $ coerce i2 i1 sigma2

instance Show G_sign where
    show (G_sign _ s) = show s

instance PrettyPrint G_sign where
    printText0 ga (G_sign _ s) = printText0 ga s

langNameSig :: G_sign -> String
langNameSig (G_sign lid _) = language_name lid

-- | Grothendieck signature lists
data G_sign_list = forall lid sublogics
        basic_spec sentence symb_items symb_map_items
         sign morphism symbol raw_symbol proof_tree .
        Logic lid sublogics
         basic_spec sentence symb_items symb_map_items
          sign morphism symbol raw_symbol proof_tree =>
  G_sign_list lid [sign]

-- | Grothendieck extended signatures
data G_ext_sign = forall lid sublogics
        basic_spec sentence symb_items symb_map_items
         sign morphism symbol raw_symbol proof_tree .
        Logic lid sublogics
         basic_spec sentence symb_items symb_map_items
          sign morphism symbol raw_symbol proof_tree =>
  G_ext_sign lid sign (Set.Set symbol)

tyconG_ext_sign :: TyCon
tyconG_ext_sign = mkTyCon "Logic.Grothendieck.G_ext_sign"
instance Typeable G_ext_sign where
  typeOf _ = mkTyConApp tyconG_ext_sign []

instance Eq G_ext_sign where
  (G_ext_sign i1 sigma1 sys1) == (G_ext_sign i2 sigma2 sys2) =
     coerce i1 i2 sigma1 == Just sigma2
     && coerce i1 i2 sys1 == Just sys2
```

```
instance Show G_ext_sign where
    show (G_ext_sign _ s _) = show s

instance PrettyPrint G_ext_sign where
    printText0 ga (G_ext_sign _ s _) = printText0 ga s

langNameExtSig :: G_ext_sign -> String
langNameExtSig (G_ext_sign lid _ _) = language_name lid

-- | Grothendieck theories
data G_theory = forall lid sublogics
        basic_spec sentence symb_items symb_map_items
         sign morphism symbol raw_symbol proof_tree .
        Logic lid sublogics
         basic_spec sentence symb_items symb_map_items
          sign morphism symbol raw_symbol proof_tree =>
  G_theory lid sign [Named sentence]

-- | compute sublogic of a theory
sublogicOfTh :: G_theory -> G_sublogics
sublogicOfTh (G_theory lid sigma sens) =
  let sub = foldr Logic.Logic.join
                  (min_sublogic_sign lid sigma)
                  (map (min_sublogic_sentence lid . sentence) sens)
   in G_sublogics lid sub

-- | simplify a theory (throw away qualifications)
simplifyTh :: G_theory -> G_theory
simplifyTh (G_theory lid sigma sens) =
  G_theory lid sigma (map (mapNamed (simplify_sen lid sigma)) sens)

-- | Grothendieck symbols
data G_symbol = forall lid sublogics
        basic_spec sentence symb_items symb_map_items
         sign morphism symbol raw_symbol proof_tree .
        Logic lid sublogics
         basic_spec sentence symb_items symb_map_items
          sign morphism symbol raw_symbol proof_tree  =>
  G_symbol lid symbol

instance Show G_symbol where
  show (G_symbol _ s) = show s

instance Eq G_symbol where
  (G_symbol i1 s1) == (G_symbol i2 s2) =
     coerce i1 i2 s1 == Just s2

-- | Grothendieck symbol lists
data G_symb_items_list = forall lid sublogics
        basic_spec sentence symb_items symb_map_items
         sign morphism symbol raw_symbol proof_tree .
        Logic lid sublogics
         basic_spec sentence symb_items symb_map_items
```

```
          sign morphism symbol raw_symbol proof_tree  =>
        G_symb_items_list lid [symb_items]

instance Show G_symb_items_list where
  show (G_symb_items_list _ l) = show l

instance PrettyPrint G_symb_items_list where
    printText0 ga (G_symb_items_list _ l) =
        fsep $ punctuate comma $ map (printText0 ga) l

instance Eq G_symb_items_list where
  (G_symb_items_list i1 s1) == (G_symb_items_list i2 s2) =
     coerce i1 i2 s1 == Just s2

-- | Grothendieck symbol maps
data G_symb_map_items_list = forall lid sublogics
        basic_spec sentence symb_items symb_map_items
         sign morphism symbol raw_symbol proof_tree .
        Logic lid sublogics
         basic_spec sentence symb_items symb_map_items
          sign morphism symbol raw_symbol proof_tree  =>
        G_symb_map_items_list lid [symb_map_items]

instance Show G_symb_map_items_list where
  show (G_symb_map_items_list _ l) = show l

instance PrettyPrint G_symb_map_items_list where
    printText0 ga (G_symb_map_items_list _ l) =
        fsep $ punctuate comma $ map (printText0 ga) l

instance Eq G_symb_map_items_list where
  (G_symb_map_items_list i1 s1) == (G_symb_map_items_list i2 s2) =
     coerce i1 i2 s1 == Just s2

-- | Grothendieck diagrams
data G_diagram = forall lid sublogics
        basic_spec sentence symb_items symb_map_items
         sign morphism symbol raw_symbol proof_tree .
        Logic lid sublogics
         basic_spec sentence symb_items symb_map_items
          sign morphism symbol raw_symbol proof_tree  =>
        G_diagram lid (Diagram sign morphism)

-- | Grothendieck sublogics
data G_sublogics = forall lid sublogics
        basic_spec sentence symb_items symb_map_items
         sign morphism symbol raw_symbol proof_tree .
        Logic lid sublogics
         basic_spec sentence symb_items symb_map_items
          sign morphism symbol raw_symbol proof_tree =>
        G_sublogics lid sublogics

tyconG_sublogics :: TyCon
```

```
tyconG_sublogics = mkTyCon "Logic.Grothendieck.G_sublogics"
instance Typeable G_sublogics where
  typeOf _ = mkTyConApp tyconG_sublogics []

instance Show G_sublogics where
    show (G_sublogics lid sub) = case sublogic_names lid sub of
      [] -> error "show G_sublogics"
      h : _ -> show lid ++ "." ++ h

instance Eq G_sublogics where
    (G_sublogics lid1 l1) == (G_sublogics lid2 l2) =
       coerce lid1 lid2 l1 == Just l2

instance Ord G_sublogics where
    compare (G_sublogics lid1 l1) (G_sublogics lid2 l2) =
     case coerce lid1 lid2 l2 of
       Just l2' -> compare l1 l2' {-if l1==l2' then EQ
                   else if l1 <<= l2' then LT
                   else GT-}
       Nothing -> error "Attempt to compare sublogics of different logics"

-- | Homogeneous Grothendieck signature morphisms
data G_morphism = forall lid sublogics
        basic_spec sentence symb_items symb_map_items
         sign morphism symbol raw_symbol proof_tree .
        Logic lid sublogics
         basic_spec sentence symb_items symb_map_items
          sign morphism symbol raw_symbol proof_tree =>
        G_morphism lid morphism

instance Show G_morphism where
    show (G_morphism _ l) = show l


------------------------------------------------------------------
-- Existential types for the logic graph
------------------------------------------------------------------

-- | Existential type for comorphisms
data AnyComorphism = forall cid lid1 sublogics1
        basic_spec1 sentence1 symb_items1 symb_map_items1
        sign1 morphism1 symbol1 raw_symbol1 proof_tree1
        lid2 sublogics2
        basic_spec2 sentence2 symb_items2 symb_map_items2
        sign2 morphism2 symbol2 raw_symbol2 proof_tree2 .
      Comorphism cid
                 lid1 sublogics1 basic_spec1 sentence1
                 symb_items1 symb_map_items1
                 sign1 morphism1 symbol1 raw_symbol1 proof_tree1
                 lid2 sublogics2 basic_spec2 sentence2
                 symb_items2 symb_map_items2
                 sign2 morphism2 symbol2 raw_symbol2 proof_tree2 =>
      Comorphism cid
```

```
instance Eq AnyComorphism where
  Comorphism cid1 == Comorphism cid2 =
     constituents cid1 == constituents cid2
  -- need to be refined, using comorphism translations !!!

instance Show AnyComorphism where
  show (Comorphism cid) =
    language_name cid
    ++" : "++language_name (sourceLogic cid)
    ++" -> "++language_name (targetLogic cid)

tyconAnyComorphism :: TyCon
tyconAnyComorphism = mkTyCon "Logic.Grothendieck.AnyComorphism"
instance Typeable AnyComorphism where
  typeOf _ = mkTyConApp tyconAnyComorphism []

-- | compute the identity comorphism for a logic
idComorphism :: AnyLogic -> AnyComorphism
idComorphism (Logic lid) = Comorphism (IdComorphism lid (top_sublogic lid))

-- | Test whether a comporphism is the identity
isIdComorphism :: AnyComorphism -> Bool
isIdComorphism (Comorphism cid) =
  constituents cid == []

-- | Compose comorphisms
compComorphism :: Monad m => AnyComorphism -> AnyComorphism -> m AnyComorphism
compComorphism cm1@(Comorphism cid1) cm2@(Comorphism cid2) =
   case coerce (targetLogic cid1) (sourceLogic cid2) (targetSublogic cid1) of
    Just sl1 ->
      if sl1 <= sourceSublogic cid2
       then case (isIdComorphism cm1,isIdComorphism cm2) of
         (True,_) -> return cm2
         (_,True) -> return cm1
         _ ->  return $ Comorphism (CompComorphism cid1 cid2)
       else fail ("Sublogic mismatch in composition of "++language_name cid1++
                  " and "++language_name cid2)
    Nothing -> fail ("Logic mismatch in composition of "++language_name cid1++
                     " and "++language_name cid2)

-- | Logic graph
data LogicGraph = LogicGraph {
                    logics :: Map.Map String AnyLogic,
                    comorphisms :: Map.Map String AnyComorphism,
                    inclusions :: Map.Map (String,String) AnyComorphism,
                    unions :: Map.Map (String,String) (AnyComorphism,AnyComorphism)
                  }

emptyLogicGraph :: LogicGraph
emptyLogicGraph = LogicGraph Map.empty Map.empty Map.empty Map.empty

-- | find a logic in a logic graph
```

```
lookupLogic :: Monad m => String -> String -> LogicGraph -> m AnyLogic
lookupLogic error_prefix logname logicGraph =
    case Map.lookup logname (logics logicGraph) of
    Nothing -> fail (error_prefix++" in LogicGraph logic \""
                     ++logname++"\" unknown")
    Just lid -> return lid

-- | union to two logics
logicUnion :: LogicGraph -> AnyLogic -> AnyLogic -> Result (AnyComorphism,AnyComorphism)
logicUnion lg l1@(Logic lid1) l2@(Logic lid2) =
  case logicInclusion lg l1 l2 of
    Result _ (Just c) -> return (c,idComorphism l2)
    _ -> case logicInclusion lg l2 l1 of
      Result _ (Just c) -> return (idComorphism l1,c)
      _ -> case Map.lookup (ln1,ln2) (unions lg) of
        Just u -> return u
        Nothing -> case Map.lookup (ln2,ln1) (unions lg) of
          Just (c2,c1) -> return (c1,c2)
          Nothing -> fail ("Union of logics "++ln1++" and "++ln2++" does not exist")
   where ln1 = language_name lid1
         ln2 = language_name lid2

-- | find a comorphism in a logic graph
lookupComorphism :: Monad m => String -> LogicGraph -> m AnyComorphism
lookupComorphism coname logicGraph = do
  let nameList = splitBy ';' coname
  cs <- sequence $ map lookupN nameList
  case cs of
    c:cs1 -> foldM compComorphism c cs1
    _ -> fail ("Illgegal comorphism name: "++coname)
  where
  lookupN name =
    case name of
      'i':'d':'_':logic -> do
         let mainLogic = takeWhile (/= '.') logic
         l <- maybe (fail ("Cannot find Logic "++mainLogic)) return
                 $ Map.lookup mainLogic (logics logicGraph)
         return $ idComorphism l
      _ -> maybe (fail ("Cannot find logic comorphism "++name)) return
             $ Map.lookup name (comorphisms logicGraph)

-- | auxiliary existential type needed for composition of comorphisms
data AnyComorphismAux lid1 sublogics1
        basic_spec1 sentence1 symb_items1 symb_map_items1
        sign1 morphism1 symbol1 raw_symbol1 proof_tree1
        lid2 sublogics2
        basic_spec2 sentence2 symb_items2 symb_map_items2
        sign2 morphism2 symbol2 raw_symbol2 proof_tree2 =
        forall cid .
      Comorphism cid
                 lid1 sublogics1 basic_spec1 sentence1
                 symb_items1 symb_map_items1
                 sign1 morphism1 symbol1 raw_symbol1 proof_tree1
```

```
                lid2 sublogics2 basic_spec2 sentence2
                symb_items2 symb_map_items2
                sign2 morphism2 symbol2 raw_symbol2 proof_tree2 =>
      ComorphismAux cid lid1 lid2 sign1 morphism2

tyconAnyComorphismAux :: TyCon
tyconAnyComorphismAux = mkTyCon "Logic.Grothendieck.AnyComorphismAux"

instance Typeable (AnyComorphismAux lid1 sublogics1
        basic_spec1 sentence1 symb_items1 symb_map_items1
        sign1 morphism1 symbol1 raw_symbol1 proof_tree1
        lid2 sublogics2
        basic_spec2 sentence2 symb_items2 symb_map_items2
        sign2 morphism2 symbol2 raw_symbol2 proof_tree2)
  where typeOf _ = mkTyConApp tyconG_sign []

instance Show (AnyComorphismAux lid1 sublogics1
        basic_spec1 sentence1 symb_items1 symb_map_items1
        sign1 morphism1 symbol1 raw_symbol1 proof_tree1
        lid2 sublogics2
        basic_spec2 sentence2 symb_items2 symb_map_items2
        sign2 morphism2 symbol2 raw_symbol2 proof_tree2)
  where show _ = "<AnyComorphismAux>"


------------------------------------------------------------------
-- The Grothendieck signature category
------------------------------------------------------------------

-- | Grothendieck signature morphisms
data GMorphism = forall cid lid1 sublogics1
        basic_spec1 sentence1 symb_items1 symb_map_items1
        sign1 morphism1 symbol1 raw_symbol1 proof_tree1
        lid2 sublogics2
        basic_spec2 sentence2 symb_items2 symb_map_items2
        sign2 morphism2 symbol2 raw_symbol2 proof_tree2 .
      Comorphism cid
                lid1 sublogics1 basic_spec1 sentence1
                symb_items1 symb_map_items1
                sign1 morphism1 symbol1 raw_symbol1 proof_tree1
                lid2 sublogics2 basic_spec2 sentence2
                symb_items2 symb_map_items2
                sign2 morphism2 symbol2 raw_symbol2 proof_tree2 =>
  GMorphism cid sign1 morphism2

instance Eq GMorphism where
  GMorphism cid1 sigma1 mor1 == GMorphism cid2 sigma2 mor2
     = Comorphism cid1 == Comorphism cid2 &&
       coerce cid1 cid1 (sigma1, mor1) == Just (sigma2, mor2)

hasIdComorphism :: GMorphism -> Bool
hasIdComorphism (GMorphism cid _ _) =
  isIdComorphism (Comorphism cid)
```

```
data Grothendieck = Grothendieck deriving Show

instance Language Grothendieck

instance Show GMorphism where
    show (GMorphism cid s m) = show cid ++ "(" ++ show s ++ ")" ++ show m

instance PrettyPrint GMorphism where
    printText0 ga (GMorphism cid s m) =
      ptext (show cid)
      <+> -- ptext ":" <+> ptext (show (sourceLogic cid)) <+>
      -- ptext "->" <+> ptext (show (targetLogic cid)) <+>
      ptext "(" <+> printText0 ga s <+> ptext ")"
      $$
      printText0 ga m

instance Category Grothendieck G_sign GMorphism where
  ide _ (G_sign lid sigma) =
    GMorphism (IdComorphism lid (top_sublogic lid)) sigma (ide lid sigma)
  comp _
       (GMorphism r1 sigma1 mor1)
       (GMorphism r2 _sigma2 mor2) =
    do let lid1 = sourceLogic r1
           lid2 = targetLogic r1
           lid3 = sourceLogic r2
           lid4 = targetLogic r2
       ComorphismAux r1' _ _ sigma1' mor1' <-
         (coerce lid2 lid3 $ ComorphismAux r1 lid1 lid2 sigma1 mor1)
       mor1'' <- map_morphism r2 mor1'
       mor <- comp lid4 mor1'' mor2
       return (GMorphism (CompComorphism r1' r2) sigma1' mor)
  dom _ (GMorphism r sigma _mor) =
    G_sign (sourceLogic r) sigma
  cod _ (GMorphism r _sigma mor) =
    G_sign lid2 (cod lid2 mor)
    where lid2 = targetLogic r
  legal_obj _ (G_sign lid sigma) = legal_obj lid sigma
  legal_mor _ (GMorphism r sigma mor) =
    legal_mor lid2 mor &&
    case maybeResult $ map_sign r sigma of
      Just (sigma',_) -> sigma' == cod lid2 mor
      Nothing -> False
    where lid2 = targetLogic r


-- | Embedding of homogeneous signature morphisms as Grothendieck sig mors
gEmbed :: G_morphism -> GMorphism
gEmbed (G_morphism lid mor) =
  GMorphism (IdComorphism lid (top_sublogic lid)) (dom lid mor) mor

-- | Embedding of comorphisms as Grothendieck sig mors
gEmbedComorphism :: AnyComorphism -> G_sign -> Result GMorphism
```

```
gEmbedComorphism (Comorphism cid) (G_sign lid sig) = do
  sig' <- mcoerce (sourceLogic cid) lid "gEmbedComorphism: logic mismatch" sig
  (sigTar,_) <- map_sign cid sig'
  let lidTar = targetLogic cid
  return (GMorphism cid sig' (ide lidTar sigTar))

-- | heterogeneous union of two Grothendieck signatures
gsigUnion :: LogicGraph -> G_sign -> G_sign -> Result G_sign
gsigUnion lg gsig1@(G_sign lid1 sigma1) gsig2@(G_sign lid2 sigma2) =
  if language_name lid1 == language_name lid2
     then homogeneousGsigUnion gsig1 gsig2
     else do
      (Comorphism cid1,Comorphism cid2) <- logicUnion lg (Logic lid1) (Logic lid2)
      let lidS1 = sourceLogic cid1
          lidS2 = sourceLogic cid2
          lidT1 = targetLogic cid1
          lidT2 = targetLogic cid2
      sigma1' <- mcoerce lid1 lidS1 "Union of signaturesa" sigma1
      sigma2' <- mcoerce lid2 lidS2 "Union of signaturesb" sigma2
      (sigma1'',_) <- map_sign cid1 sigma1'
      (sigma2'',_) <- map_sign cid2 sigma2'
      sigma2''' <- mcoerce lidT1 lidT2 "Union of signaturesc" sigma2''
      sigma3 <- signature_union lidT1 sigma1'' sigma2'''
      return (G_sign lidT1 sigma3)


-- | homogeneous Union of two Grothendieck signatures
homogeneousGsigUnion :: G_sign -> G_sign -> Result G_sign
homogeneousGsigUnion (G_sign lid1 sigma1) (G_sign lid2 sigma2) = do
  sigma2' <- mcoerce lid2 lid1 "Union of signaturesd" sigma2
  sigma3 <- signature_union lid1 sigma1 sigma2'
  return (G_sign lid1 sigma3)

-- | union of a list of Grothendieck signatures
gsigManyUnion :: LogicGraph -> [G_sign] -> Result G_sign
gsigManyUnion _ [] =
  fail "union of emtpy list of signatures"
gsigManyUnion lg (gsigma : gsigmas) =
  foldM (gsigUnion lg) gsigma gsigmas

-- | homogeneous Union of a list of morphisms
homogeneousMorManyUnion :: [G_morphism] -> Result G_morphism
homogeneousMorManyUnion [] =
  fail "homogeneous union of emtpy list of morphisms"
homogeneousMorManyUnion (G_morphism lid mor : gmors) = do
  mors <- let coerce_lid (G_morphism lid1 mor1) =
                    mcoerce lid lid1  "Union of signature morphisms" mor1
          in sequence (map coerce_lid gmors)
  bigMor <- let mor_union s1 s2 = do
                      s1' <- s1
                      morphism_union lid s1' s2
            in foldl mor_union (return mor) mors
  return (G_morphism lid bigMor)
```

```
-- | inclusion between two logics
logicInclusion :: LogicGraph -> AnyLogic -> AnyLogic -> Result AnyComorphism
logicInclusion logicGraph l1@(Logic lid1) (Logic lid2) =
     let ln1 = language_name lid1
         ln2 = language_name lid2 in
     if ln1==ln2 then
       return (idComorphism l1)
      else case Map.lookup (ln1,ln2) (inclusions logicGraph) of
           Just (Comorphism i) ->
               return (Comorphism i)
           Nothing ->
               fail ("No inclusion from "++ln1++" to "++ln2++" found")

-- | inclusion morphism between two Grothendieck signatures
ginclusion :: LogicGraph -> G_sign -> G_sign -> Result GMorphism
ginclusion logicGraph (G_sign lid1 sigma1) (G_sign lid2 sigma2) = do
    Comorphism i <- logicInclusion logicGraph (Logic lid1) (Logic lid2)
    sigma1' <- mcoerce lid1 (sourceLogic i) "Inclusion of signatures" sigma1
    (sigma1'',_) <- map_sign i sigma1'
    sigma2' <- mcoerce lid2 (targetLogic i) "Inclusion of signatures" sigma2
    mor <- inclusion (targetLogic i) sigma1'' sigma2'
    return (GMorphism i sigma1' mor)

-- | Composition of two Grothendieck signature morphisms
-- | with itermediate inclusion
compInclusion :: LogicGraph -> GMorphism -> GMorphism -> Result GMorphism
compInclusion lg mor1 mor2 = do
  incl <- ginclusion lg (cod Grothendieck mor1) (dom Grothendieck mor2)
  mor <- comp Grothendieck mor1 incl
  comp Grothendieck mor mor2

-- | Composition of two Grothendieck signature morphisms
-- | with itermediate homogeneous inclusion
compHomInclusion :: GMorphism -> GMorphism -> Result GMorphism
compHomInclusion mor1 mor2 = compInclusion emptyLogicGraph mor1 mor2

-- | Translation of a G_l_sentence_list along a GMorphism
translateG_l_sentence_list :: GMorphism -> G_l_sentence_list
                                  -> Result G_l_sentence_list
translateG_l_sentence_list (GMorphism cid sign1 morphism2)
                           (G_l_sentence_list lid sens) = do
  let tlid = targetLogic cid
  --(sigma2,_) <- map_sign cid sign1
  sens' <- mcoerce lid (sourceLogic cid) "Translation of sentence list" sens
  sens'' <- mapM (mapNamedM $ map_sentence cid sign1) sens'
  sens''' <- mapM (mapNamedM $ map_sen tlid morphism2) sens''
  return (G_l_sentence_list tlid sens''')

-- | Join two G_l_sentence_list's
joinG_l_sentence_list :: G_l_sentence_list -> G_l_sentence_list
                           -> Maybe G_l_sentence_list
joinG_l_sentence_list (G_l_sentence_list lid1 sens1)
```

```
                      (G_l_sentence_list lid2 sens2) = do
  sens2' <- mcoerce lid1 lid2 "Union of sentence lists" sens2
  return (G_l_sentence_list lid1 (sens1++sens2'))

-- | Flatten a list of G_l_sentence_list's
flatG_l_sentence_list :: [G_l_sentence_list] -> Maybe G_l_sentence_list
flatG_l_sentence_list [] = Nothing
flatG_l_sentence_list (gl:gls) = foldM joinG_l_sentence_list gl gls


-- | Find all (composites of) comorphisms starting from a given logic
findComorphismPaths :: LogicGraph ->  G_sublogics -> [AnyComorphism]
findComorphismPaths lg (G_sublogics lid sub) =
  List.nub $ map fst $ iterateComp (0::Int) [(idc,[idc])]
  where
  idc = Comorphism (IdComorphism lid sub)
  coMors = Map.elems $ comorphisms lg
  -- compute possible compositions, but only up to depth 5
  iterateComp n l = -- (l::[(AnyComorphism,[AnyComorphism])]) =
    if n>5 || l==newL then newL else iterateComp (n+1) newL
    where
    newL = List.nub (l ++ (concat (map extend l)))
    -- extend comorphism list in all directions, but no cylces
    extend (coMor,comps) =
       let addCoMor c =
            case compComorphism coMor c of
              Nothing -> Nothing
              Just c1 -> Just (c1,c:comps)
        in catMaybes $ map addCoMor $ filter (not . (`elem` comps)) $ coMors


------------------------------------------------------------------------
-- Provers
------------------------------------------------------------------------

-- | provers and consistency checkers for specific logics
data G_prover = forall lid sublogics
        basic_spec sentence symb_items symb_map_items
         sign morphism symbol raw_symbol proof_tree .
        Logic lid sublogics
         basic_spec sentence symb_items symb_map_items
          sign morphism symbol raw_symbol proof_tree =>
       G_prover lid (Prover sign sentence proof_tree symbol)
     | forall lid sublogics
        basic_spec sentence symb_items symb_map_items
         sign morphism symbol raw_symbol proof_tree .
        Logic lid sublogics
         basic_spec sentence symb_items symb_map_items
          sign morphism symbol raw_symbol proof_tree =>
       G_cons_checker lid (ConsChecker sign sentence morphism proof_tree)
```

```
------------------------------------------------------------------
-- Coercion
------------------------------------------------------------------

-- | coerce a theory into a "different" logic
coerceTheory :: forall lid sublogics
        basic_spec sentence symb_items symb_map_items
         sign morphism symbol raw_symbol proof_tree .
        Logic lid sublogics
         basic_spec sentence symb_items symb_map_items
          sign morphism symbol raw_symbol proof_tree =>
      lid -> G_theory -> Result (sign, [Named sentence])
coerceTheory lid (G_theory lid2 sign2 sens2)
  = mcoerce lid lid2 "Coercion of theories" (sign2,sens2)
```

B.4 Haskell Data Structure for Heterogeneous Development Graphs

```
{-|

Module      :  /repository/HetCATS/Static/DevGraph.hs
Copyright   :  (c) Till Mossakowski, Uni Bremen 2002-2004
Licence     :  similar to LGPL, see HetCATS/LICENCE.txt or LIZENZ.txt

Maintainer  :  hets@tzi.de
Stability   :  provisional
Portability :  non-portable(Logic)

   Central data structure for development graphs.
   Follows Sect. IV:4.2 of the CASL Reference Manual.
-}

{-
   References:

   T. Mossakowski, S. Autexier and D. Hutter:
   Extending Development Graphs With Hiding.
   H. Hussmann (ed.): Fundamental Approaches to Software Engineering 2001,
   Lecture Notes in Computer Science 2029, p. 269-283,
   Springer-Verlag 2001.

   T. Mossakowski, S. Autexier, D. Hutter, P. Hoffman:
   CASL Proof calculus. In: CASL reference manual, part IV.
   Available from http://www.cofi.info

todo:

Integrate stuff from Saarbrücken
Should AS be stored in GloblaContext as well?
-}
```

```
module Static.DevGraph where

import Logic.Logic
import Logic.Grothendieck
import Syntax.AS_Library
import Common.GlobalAnnotations

import Common.Lib.Graph as Graph
import qualified Common.Lib.Map as Map
import qualified Common.Lib.Set as Set
import Common.Id
import Common.PrettyPrint
import Common.PPUtils
import Common.Result
import Common.Lib.Pretty


-- * Types for structured specification analysis

data DGNodeLab = DGNode {
                 dgn_name :: Maybe SIMPLE_ID,
                 dgn_sign :: G_sign,
                 dgn_sens :: G_l_sentence_list,
                 dgn_origin :: DGOrigin
               }
             | DGRef {
                 dgn_renamed :: Maybe SIMPLE_ID,
                 dgn_libname :: LIB_NAME,
                 dgn_node :: Node
               } deriving (Show,Eq)

data DGLinkLab = DGLink {
               -- dgl_name :: String,
               -- dgl_src, dgl_tar :: DGNodeLab,  -- already in graph structure
               dgl_morphism :: GMorphism,
               dgl_type :: DGLinkType,
               dgl_origin :: DGOrigin }
               deriving (Eq,Show)

data ThmLinkStatus =  Open | Proven [DGLinkLab] deriving (Eq, Show)

data DGLinkType = LocalDef
             | GlobalDef
             | HidingDef
             | FreeDef NodeSig -- the "parameter" node
             | CofreeDef NodeSig -- the "parameter" node
             | LocalThm ThmLinkStatus Conservativity ThmLinkStatus
             | GlobalThm ThmLinkStatus Conservativity ThmLinkStatus
             | HidingThm GMorphism ThmLinkStatus
             | FreeThm GMorphism Bool
               -- DGLink S1 S2 m2 (DGLinkType m1 p) n
               -- corresponds to a span of morphisms
               -- S1 <--m1-- S --m2--> S2
```

```
              deriving (Eq,Show)

data Conservativity = None | Cons | Mono | Def
              deriving (Eq,Show)

data DGOrigin = DGBasic | DGExtension | DGTranslation | DGUnion | DGHiding
              | DGRevealing | DGRevealTranslation | DGFree | DGCofree
              | DGLocal | DGClosed | DGClosedLenv | DGLogicQual | DGLogicQualLenv
              | DGData
              | DGFormalParams | DGImports | DGSpecInst SIMPLE_ID | DGFitSpec
              | DGView SIMPLE_ID | DGFitView SIMPLE_ID | DGFitViewImp SIMPLE_ID
              | DGFitViewA SIMPLE_ID | DGFitViewAImp SIMPLE_ID | DGProof
              deriving (Eq,Show)

type DGraph = Graph DGNodeLab DGLinkLab

data NodeSig = NodeSig (Node,G_sign) | EmptyNode AnyLogic
               deriving (Eq,Show)
```

Appendix C

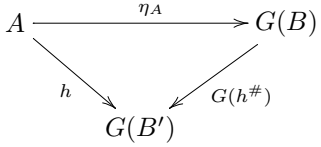
Free and Cofree Specifications

C.1 Free extensions and liberality

We have mostly left out the institution independent structuring construct of free extensions so far. We treat it separately because in many structuring languages, it is not included, and moreover, it is not preserved so well along institution comorphisms (see Sect. 2.6).

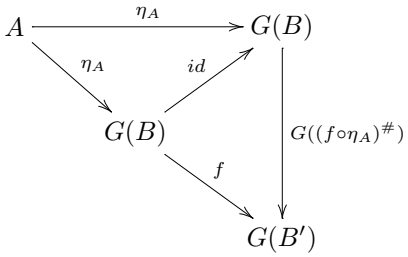
The structured **free** construct restricts the model class to *initial* or *free* models. That is, if Sp_1 is a specification with signature Σ_1, then the models of Sp_1 **then free** $\{Sp_2\}$ are those models M of Sp_1 **then** Sp_2 that are free over $M|_{\Sigma_1}$ w.r.t. the reduct functor $\_|_{\Sigma_1}$ associated to the inclusion of Σ_1 into the signature of Sp_1 **then** Sp_2. This allows for the specification of datatypes that are generated freely w.r.t. given axioms, as, for example, in the specification of finite sets over a state sort which is part of the specification of nondeterministic automata in Figure C.1. Here, the **assoc**, **comm**, **idem** and **unit** attributes specify the operation $\_ \cup \_$ to be associative, commutative, idempotent and have unit $\{\}$.

Free extensions can formally be defined w.r.t. an arbitrary functor: Given categories $\mathbf{A}$ and $\mathbf{B}$ and a functor $G\colon \mathbf{B} \longrightarrow \mathbf{A}$, an object $B \in \mathbf{B}$ is called *G-free (with unit $\eta_A\colon A \longrightarrow G(B)$) over* $A \in \mathbf{A}$, if for any object $B' \in \mathbf{B}$ and any morphism $h\colon A \longrightarrow G(B')$, there is a unique morphism $h^\#\colon B \longrightarrow B'$ such that $G(h^\#) \circ \eta_A = h$.



In this case, the unit η_A is called a *G-universal arrow*. We will mostly omit the specification of the unit. An object $B \in \mathbf{B}$ is called *persistently G-free*, if it is G-free over some $A \in \mathbf{A}$ with the unit being an isomorphism. It is called *strongly persistently G-free* if it is G-free with unit id over $G(B)$ (id denotes the identity).

Proposition C.1 Given a functor $G\colon \mathbf{B} \longrightarrow \mathbf{A}$, an object $B \in \mathbf{B}$ is persistently G-free if and only if it is strongly persistently G-free.



PROOF: The "if" direction is clear. For the "only if" direction, let $\eta_A: A \longrightarrow G(B)$ be a G-universal isomorphism. If $f: G(B) \longrightarrow G(B')$ is a morphism, $(f \circ \eta_A)^\#$ is the unique morphism $g: B \longrightarrow B'$ with $id \circ G(g) = f$. Hence, $id: G(B) \longrightarrow G(B)$ is G-universal as well. □

We now extend the kernel language of Sect. 5.1 as follows. For any signature morphism $\sigma: \Sigma \longrightarrow \Sigma'$ and Σ'-specification SP', **free** SP' **along** σ is a specification with:

$Sig(\textbf{free } SP' \textbf{ along } \sigma) = \Sigma'$

$\textbf{Mod}(\textbf{free } SP' \textbf{ along } \sigma) = \{M' \in \textbf{Mod}(SP') \mid$
M' is strongly persistently $(\textbf{Mod}(\sigma): \textbf{Mod}(SP') \longrightarrow \textbf{Mod}(\Sigma))$-free $\}$

These specifications are called **data** SP' **over** σ in [ST88b]. Since the unit has to be the identity, the freeness condition for M' means that for any model $N' \in \textbf{Mod}(SP')$ and any model morphism $h: M'|_\sigma \longrightarrow N'|_\sigma$, there is a unique morphism $h^\#: M' \longrightarrow N'$ such that $(h^\#)|_\sigma = h$. By Proposition C.1, we can equally use just persistent freeness instead of strongly persistent freeness.

Free extensions allow one to express certain inductive properties in a concise way. For example, the transitive closure of an arbitrary relation can be specified using the **free** construct in CASL as follows:

Example C.2 **spec** BINARYRELATION =
  sort *Elem*
  pred $\_ \sim \_ : Elem \times Elem$
end

spec TRANSITIVECLOSURE [BINARYRELATION] =
  free
  { **pred** $\_ \sim^* \_ : Elem \times Elem$
    $\forall\ x, y, z : Elem$
    • $x \sim y \Rightarrow x \sim^* y$
    • $x \sim^* y \wedge y \sim^* z \Rightarrow x \sim^* z$
  }
end □

The corresponding structured specification is constructed as follows: Let $\langle \Sigma', \Psi' \rangle$ be the presentation consisting of all sorts, predicates and axioms declared in either of BINARYRELATION and TRANSITIVECLOSURE, and let Σ be the signature of BINARYRELATION. Then as denotation of the above specification, we get

$$\textbf{free } \langle \Sigma', \Psi' \rangle \textbf{ along } \sigma$$

where σ is the inclusion of Σ into Σ'.

Another use of the free construct is in the generation of datatypes. For examples, consider the specification of finite sets over arbitrary elements in CASL:

Example C.3 **spec** GENERATEFINITESET [**sort** *Elem*] =
free
  { **type** $FinSet[Elem] ::= \{\} \mid \{\_\}(Elem) \mid \_\_ \cup \_\_(FinSet[Elem]; FinSet[Elem])$
    op $\_\_ \cup \_ : FinSet[Elem] \times FinSet[Elem] \to FinSet[Elem]$,
      assoc, **comm**, **idem**, **unit** $\{\}$
  }
end

This expands to the following:

spec GENERATEFINITESET [**sort** *Elem*] =
free

{ **sort** $FinSet[Elem]$
ops $\{\}: FinSet[Elem]$;
$\{\_\}: Elem \to FinSet[Elem]$;
$\_\_ \cup \_\_: FinSet[Elem] \times FinSet[Elem] \to FinSet[Elem]$
forall $x, y, z : Elem$
- $x \cup (y \cup z) = (x \cup y) \cup z$
- $x \cup y = y \cup x$
- $x \cup x = x$
- $x \cup \{\} = x$

}
end

The question whether free models actually exist leads to the notion of liberality [GB92]:

Definition C.4 Given an institution $I = (\mathbf{Sign}, \mathbf{Sen}, \mathbf{Mod}, \models)$,

- a theory morphism $\sigma: T \longrightarrow T'$ is said to be *liberal* if, for each T-model M, there is a T'-model M' that is $\mathbf{Mod}(\sigma)$-free over M; it is called *strongly persistently liberal* if, moreover, M' is strongly persistently $\mathbf{Mod}(\sigma)$-free;
- the institution I is called liberal if each of its theory morphisms is liberal;
- given a class $\mathcal{M}$ of theory morphisms in I, I is called $\mathcal{M}$-liberal if each theory morphism in $\mathcal{M}$ is liberal. □

The notion of (strongly persistent) liberality can easily be extended from theory morphisms to specification morphisms.

One could guess that in a liberal institution, **free** SP **along** σ is consistent whenever SP is. However, this is not the case, as the following counterexample shows:

Example C.5 The institution $Eq^=$ of equational logic is known to be liberal[1] (see [GB92] below). Let Σ consist of one sort s and one constant $c : s$, let Σ' be Σ plus one unary function $f: s \longrightarrow s$, and let $\sigma: \Sigma \longrightarrow \Sigma'$ be the inclusion. Clearly $\langle \Sigma', \emptyset \rangle$ is consistent. However, **free** $\langle \Sigma', \emptyset \rangle$ **along** σ is inconsistent: any Σ-model is freely extended by adding an ω-chain

$$\{ f(a),\ f(f(a)),\ f(f(f(a))),\ \ldots \}$$

over each of its elements a. Thus, a Σ'-model can never be free over its own σ-reduct. □

To show consistency of **free** SP **along** σ, we need strongly persistent liberality:

Proposition C.6 If $\sigma: \langle \Sigma, \emptyset \rangle \longrightarrow \langle \Sigma', \Psi' \rangle$ is strongly persistently liberal and $\langle \Sigma', \Psi' \rangle$ is consistent, then also

$$\textbf{free } \langle \Sigma', \Psi' \rangle \textbf{ along } \sigma$$

is consistent.

PROOF: Let $M'_1 \in \mathbf{Mod}(\langle \Sigma', \Psi' \rangle)$ by consistency, and let M' be $\mathbf{Mod}(\sigma)$-free over $M'_1|_\sigma$ with M' also being strongly persistently $\mathbf{Mod}(\sigma)$-free. Then $M' \in \mathbf{Mod}(\textbf{free } \langle \Sigma', \Psi' \rangle \textbf{ along } \sigma)$, and hence, **free** $\langle \Sigma', \Psi' \rangle$ **along** σ is consistent. □

For specifications containing **free**, it is not so easy to obtain a normal form. This is because in general **free** does not commute with the other specification building operations.

[1] One has to allow empty carrier sets or restrict oneself to strict theory morphisms to get this result, see [Mos02].

C.2 Borrowing For Structured Specifications (Including free)

It is easy to show borrowing for structured specifications including **free** under the very strong assumption of a *subinstitution comorphism*:

Theorem C.7 Let $\mu = (\Phi, \alpha, \beta): I \longrightarrow J$ be a subinstitution comorphism. Then

- For any specification SP,

$$\beta^{-1}_{Sig(SP)}(\mathbf{Mod}^I(SP)) = \mathbf{Mod}^J(\hat{\mu}(SP)).$$

- μ admits borrowing of entailment and refinement for all specifications.

PROOF: (1) is straightforward. (2) follows with Proposition 5.2(2). □

Thus

Subinstitution comorphisms admit borrowing of entailment and refinement for all structured specifications.

We here use the term liberal (in accordance with [Dia98]) since it stresses the connection with liberality of institutions. Meseguer [Mes98a] has introduced persistently liberal comorphisms under the name of *extensions*. He additionally requires that the isomorphism $id \cong \beta_\Sigma \circ \gamma_\Sigma$ is the unit of the adjunction; however, in the light of Proposition C.10, this requirement is superfluous.

Let us now study how persistently liberal institution comorphisms interact with liberality, strengthening a result of [KM95, Mos96b] (we can now drop the assumption of the existence of $\mathbf{Mod}^I(\sigma)$-free models).

Theorem C.8 Let $(\mu, \gamma) = ((\Phi, \alpha, \beta), \gamma): I \longrightarrow J$ be a persistently liberal institution comorphism such that additionally either satisfaction in I is closed under isomorphism or (μ, γ) is even strongly persistently liberal. Then free constructions can be lifted against (μ, γ) in the following sense:

1. If $\sigma: \langle \Sigma_1, \Psi_1 \rangle \longrightarrow \langle \Sigma_2, \Psi_2 \rangle$ is a theory morphism in I, $M_1 \in \mathbf{Mod}^I(\langle \Sigma_1, \Psi_1 \rangle)$, and if $M_2' \in \mathbf{Mod}^J(\hat{\mu}(\langle \Sigma_2, \Psi_2 \rangle))$ is $\mathbf{Mod}^J(\Phi(\sigma))$-free over $\gamma_{\Sigma_1}(M_1)$, then $\beta_{\Sigma_2}(M_2')$ is $\mathbf{Mod}^I(\sigma)$-free over M_1.
2. If $\sigma: \langle \Sigma_1, \Psi_1 \rangle \longrightarrow \langle \Sigma_2, \Psi_2 \rangle$ is a theory morphism in I, then σ is liberal if $\Phi(\sigma)$ is liberal.
3. I is liberal if J is liberal.
4. Let $\mathcal{M}$ be a class of signature morphisms in $\mathbf{Sign}^I$. I is $\mathcal{M}$-liberal if J is $\Phi(\mathcal{M})$-liberal.

In a word:

Free constructions can be lifted against persistently liberal institution comorphisms.

PROOF:

$$\begin{array}{ccc}
\mathbf{Mod}^I(\langle \Sigma_2, \Psi_2 \rangle) & \underset{\beta_{\Sigma_2}}{\overset{\gamma_{\Sigma_2}}{\rightleftarrows}} & \mathbf{Mod}^J(\hat{\mu}(\langle \Sigma_2, \Psi_2 \rangle)) \\
\Big\downarrow{\scriptstyle \mathbf{Mod}^I(\sigma)} & & \Big\downarrow{\scriptstyle \mathbf{Mod}^J(\Phi(\sigma))} \\
\mathbf{Mod}^I(\langle \Sigma_1, \Psi_1 \rangle) & \underset{\beta_{\Sigma_1}}{\overset{\gamma_{\Sigma_1}}{\rightleftarrows}} & \mathbf{Mod}^J(\hat{\mu}(\langle \Sigma_1, \Psi_1 \rangle))
\end{array}$$

(1) By the satisfaction condition, β_{Σ_i} restricts to $\beta_{\Sigma_i}: \mathbf{Mod}^J(\hat{\mu}(\langle \Sigma_i, \Psi_i \rangle)) \longrightarrow \mathbf{Mod}^I(\langle \Sigma_i, \Psi_i \rangle)$ for $i = 1, 2$. We now show that similarly, γ_{Σ_i} restricts to $\gamma_{\Sigma_i}: \mathbf{Mod}^I(\langle \Sigma_i, \Psi_i \rangle) \longrightarrow \mathbf{Mod}^J(\hat{\mu}(\langle \Sigma_i, \Psi_i \rangle))$:

For $M \in \mathbf{Mod}^I(\langle \Sigma_i, \Psi_i \rangle)$, either $\beta_{\Sigma_i}(\gamma_{\Sigma_i}(M)) = M$, or $\beta_{\Sigma_i}(\gamma_{\Sigma_i}(M)) \cong M$ and satisfaction in I is closed under isomorphism. In both cases, $\beta_{\Sigma_i}(\gamma_{\Sigma_i}(M)) \models_{\Sigma_i} \Psi_i$, and by the satisfaction condition, $\gamma_{\Sigma_i}(M) \models \alpha_{\Sigma_i}(\Psi_i)$, hence $\gamma_{\Sigma_i}(M) \in \mathbf{Mod}^J(\hat{\mu}(\langle \Sigma_i, \Psi_i \rangle))$. We now can apply Proposition C.11 (1) with $U = \mathbf{Mod}^I(\sigma)$, $V = \mathbf{Mod}^J(\Phi(\sigma))$, $R = \beta_{\Sigma_2}$, $L = \gamma_{\Sigma_2}$, $R' = \beta_{\Sigma_1}$, and $L' = \gamma_{\Sigma_1}$, $B = M_2'$ and $X = M_1$.

(2), (3) and (4) directly follow from (1). □

Example 2.32 below shows that the assumption of persistent liberality is needed to get this result.

We now come to borrowing for specifications containing **free**.

Proposition C.9 Let $(\mu, \gamma, \delta) \colon I \longrightarrow J$ be a persistently bi-liberal institution comorphism such that γ and δ are natural transformations, let SP be a structured Σ-specification and let φ be a Σ-sentence. Then

1. $\mathbf{Mod}^I(SP) = (\gamma_\Sigma)^{-1}(\mathbf{Mod}^J(\hat{\mu}(SP)))$,
2. $\beta_\Sigma(\mathbf{Mod}^J(\hat{\mu}(SP)) \subseteq \mathbf{Mod}^I(SP))$,
3. $\beta_\Sigma(\mathbf{Mod}^J(\hat{\mu}(SP)) \supseteq \mathbf{Mod}^I(SP))$,
4. $SP \models^I_\Sigma \varphi$ iff $\hat{\mu}(SP) \models^J_{\Phi(\Sigma)} \alpha_\Sigma(\varphi)$.

Proof: We prove (1) and (2) simultaneously by induction over SP.

- $SP = \langle \Sigma, \Psi \rangle$:
  (1): $M \in \mathbf{Mod}^I(SP)$ iff $M \models_\Sigma \Psi$ iff $\beta_\Sigma(\gamma_\Sigma(M)) \models_\Sigma \Psi$ iff (by the satisfaction condition) $\gamma_\Sigma(M) \models_{\Phi(\Sigma)} \alpha_\Sigma(\Psi)$ iff $\gamma_\Sigma(M) \in \mathbf{Mod}^J(\hat{\mu}(SP))$. (2) follows by the same inductive argument as used in the proof of Lemma 8.6 in [Bor02].

- $SP = SP_1 \cup SP_2$:
  (1): $M \in \mathbf{Mod}^I(SP)$ iff $M \in \mathbf{Mod}^I(SP_1) \cap \mathbf{Mod}^I(SP_2)$ iff (by the induction hypothesis) $\gamma_\Sigma(M) \in \mathbf{Mod}^J(\hat{\mu}(SP_1)) \cap \mathbf{Mod}^J(\hat{\mu}(SP_2))$ iff $\gamma(M) \in \mathbf{Mod}^J(\hat{\mu}(SP))$. (2) follows in the same way as above.

- $SP =$ **translate** SP_1 **by** $\sigma \colon \Sigma_1 \longrightarrow \Sigma$:
  (1): $M \in \mathbf{Mod}^I(SP)$ iff $M|_\sigma \in \mathbf{Mod}^I(SP_1)$ iff (by the induction hypothesis) $\gamma_{\Sigma_1}(M|_\sigma) \in \mathbf{Mod}^J(\hat{\mu}(SP_1))$ iff (since γ is $\mathcal{D}$-natural and $\sigma \in \mathcal{D}$) $(\gamma_\Sigma(M))|_{\Phi(\sigma)} \in \mathbf{Mod}^J(\hat{\mu}(SP_1))$ iff $\gamma_\Sigma(M) \in \mathbf{Mod}^J(\hat{\mu}(SP))$. (2) follows in the same way as above.

- $SP =\vdash SP'\sigma \colon \Sigma \longrightarrow \Sigma_1$:
  (1): $M \in \mathbf{Mod}^I(SP)$ implies that there is some $M' \in \mathbf{Mod}^I(SP')$ with $M'|_\sigma = M$. Then, by the induction hypothesis, $\gamma_{\Sigma_1}(M') \in \mathbf{Mod}^J(\hat{\mu}(SP'))$, and since γ is $\mathcal{D}$-natural and $\sigma \in \mathcal{D}$, $(\gamma_{\Sigma_1}(M'))|_{\Phi(\sigma)} = \gamma_\Sigma(M'|_\sigma) = \gamma_\Sigma(M)$. Thus, $\gamma_\Sigma(M) \in \mathbf{Mod}^J(\hat{\mu}(SP))$.

  Conversely, assume that $\gamma_\Sigma(M) \in \mathbf{Mod}^J(\hat{\mu}(SP))$. Then there is some $M_1 \in \mathbf{Mod}^J(\hat{\mu}(SP))$ with $M_1|_{\Phi(\sigma)} = \gamma_\Sigma(M)$. Hence, $\beta_{\Sigma_1}(M_1)|_\sigma = \beta_\Sigma(M_1|_{\Phi(\sigma)}) = \beta_\Sigma(\gamma_\Sigma(M)) = M$. Therefore, $M \in \mathbf{Mod}^I(SP)$.

  (2) follows in the same way as above.

- $SP =$ **free** SP' **along** $\sigma \colon \Sigma_1 \longrightarrow \Sigma$:
  By the induction hypothesis, (1) $\mathbf{Mod}^I(SP') = (\gamma_\Sigma)^{-1}(\mathbf{Mod}^J(\hat{\mu}(SP')))$, which implies $\gamma_\Sigma(\mathbf{Mod}^I(SP')) \subseteq \mathbf{Mod}^J(\hat{\mu}(SP'))$, and (2) $\beta_\Sigma(\mathbf{Mod}^J(\hat{\mu}(SP'))) \subseteq \mathbf{Mod}^I(SP')$. Thus, we can restrict γ_Σ to

$\gamma_\Sigma\colon \mathbf{Mod}^I(SP') \longrightarrow \mathbf{Mod}^J(\hat{\mu}(SP'))$, and β_Σ to $\beta_\Sigma\colon \mathbf{Mod}^J(\hat{\mu}(SP')) \longrightarrow \mathbf{Mod}^I(SP')$:

$$\begin{array}{ccc}
\mathbf{Mod}^I(SP') & \underset{\beta_\Sigma}{\overset{\gamma_\Sigma}{\rightleftarrows}} & \mathbf{Mod}^J(\hat{\mu}(SP')) \\
\Big\downarrow{\scriptstyle \mathbf{Mod}^I(\sigma)} & & \Big\downarrow{\scriptstyle \mathbf{Mod}^J(\Phi(\sigma))} \\
\mathbf{Mod}^I(\Sigma_1) & \underset{\beta_{\Sigma_1}}{\overset{\gamma_{\Sigma_1}}{\rightleftarrows}} & \mathbf{Mod}^J(\Phi(\Sigma_1))
\end{array}$$

Since the subcategories determined by specifications are full, the thus restricted functor β_Σ is still right adjoint to the restricted γ_Σ. Concerning (1), $M \in \mathbf{Mod}^I(SP)$ iff M is strongly persistently $\mathbf{Mod}^I(\sigma)$-free. Since γ is $\mathcal{D}$-natural and $\sigma \in \mathcal{D}$, we can apply Proposition A.2(3) of [Mos02] with $U = \mathbf{Mod}^I(\sigma)$, $V = \mathbf{Mod}^J(\Phi(\sigma))$, $R = \beta_\Sigma$, $L = \gamma_\Sigma$, $R' = \beta_{\Sigma_1}$ and $L' = \gamma_{\Sigma_1}$. From this, we obtain that the above holds iff $\gamma_\Sigma(M)$ is strongly persistently $\mathbf{Mod}^J(\Phi(\sigma))$-free. This in turn holds iff $\gamma_\Sigma(M) \in \mathbf{Mod}^J(\hat{\mu}(SP))$.

Concerning (2), let $M' \in \mathbf{Mod}^J(\hat{\mu}(SP))$. Then M' is strongly persistently $\mathbf{Mod}^J(\Phi(\sigma))$-free. By Proposition 2.28, $\beta_\Sigma \circ \delta_\Sigma \cong id$ and $\beta_{\Sigma_1} \circ \delta_{\Sigma_1} \cong id$. Since δ is $\mathcal{D}$-natural and $\sigma \in \mathcal{D}$, we can apply Proposition A.2(3) of [Mos02] with $U = \mathbf{Mod}^J(\Phi(\sigma))$, $V = \mathbf{Mod}^I(\sigma)$, $R = \delta_\Sigma$, $L = \beta_\Sigma$, $R' = \delta_{\Sigma_1}$ and $L' = \beta_{\Sigma_1}$. Thus, $\beta_\Sigma(M')$ is strongly persistently $\mathbf{Mod}^I(\sigma)$-free. Hence, $\beta_\Sigma(M') \in \mathbf{Mod}^I(SP)$.

(3) Let $M \in \mathbf{Mod}^I(SP)$. By (1), $\gamma_\Sigma(M) \in \mathbf{Mod}^J(\hat{\mu}(SP))$. By $\beta_\Sigma(\gamma_\Sigma(M)) = M$, $M \in \beta_\Sigma(\mathbf{Mod}^J(\hat{\mu}(SP)))$.
(4) $SP \models^I_\Sigma \varphi$ iff (by definition) $M \in \mathbf{Mod}^I(SP)$ implies $M \models^I_\Sigma \varphi$ iff (by (3)) $M' \in \mathbf{Mod}^J(\hat{\mu}(SP))$ implies $\beta_\Sigma(M') \models^I_\Sigma \varphi$ iff (by the satisfaction condition) $M' \in \mathbf{Mod}^J(\hat{\mu}(SP))$ implies $M' \models^J_{\Phi(\Sigma)} \alpha_\Sigma(\varphi)$ iff (by definition) $\hat{\mu}(SP) \models^J_{\Phi(\Sigma)} \alpha_\Sigma(\varphi)$. □

C.3 Preservation of Freeness

The following propositions are important for proving properties about persistently liberal institution comorphisms. We will use the following terminology: Given a functor F left adjoint to G, η^G and ε^F (or just η and ε, if no confusion can arise) will denote the unit and counit of some corresponding adjoint situation.

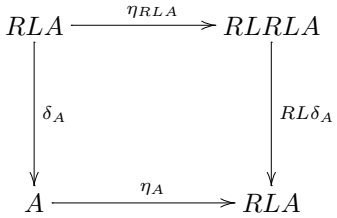
Proposition C.10 Given $R\colon \mathbf{B} \longrightarrow \mathbf{A}$ with left adjoint L such that $R \circ L \cong id$, then η, ε_L, and $R\varepsilon$ are isomorphisms. Moreover, L is full.

Proof: Let $\delta\colon R \circ L \longrightarrow id$ be a natural isomorphism, and let A be an object in $\mathbf{A}$.

$$\begin{array}{ccc}
RLA & \xrightarrow{(\delta\circ\eta)_{RLA}} & RLA \\
\Big\downarrow{\scriptstyle R\varepsilon_{LA}\circ\delta^{-1}_{RLA}} & \searrow^{id} & \Big\downarrow{\scriptstyle R\varepsilon_{LA}\circ\delta^{-1}_{RLA}} \\
RLA & \xrightarrow{(\delta\circ\eta)_{RLA}} & RLA
\end{array}$$

Since $R\varepsilon \circ \eta_R = id$ for any adjunction, $R\varepsilon_{LA} \circ \delta^{-1}_{RLA} \circ \delta_{RLA} \circ \eta_{RLA} = id$. Hence, the upper triangle commutes. By naturality of $\delta \circ \eta$, also the square commutes. Thus, also the lower triangle commutes. But this shows $(\delta \circ \eta)_{RLA}$ to be an isomorphism. Since δ is an isomorphism, η_{RLA} is an

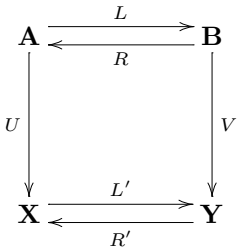
isomorphism as well. By naturality of η,



commutes. Hence, η_A is an isomorphism. Since $\varepsilon_L \circ L\eta = id$ and $R\varepsilon \circ \eta_R = id$ for any adjunction, ε_L and $R\varepsilon$ are isomorphisms as well.

Fullness of L follows with the dual of [AHS90], 19.14. □

Proposition C.11 Let the following diagram of categories and functors be given.



Assume further that

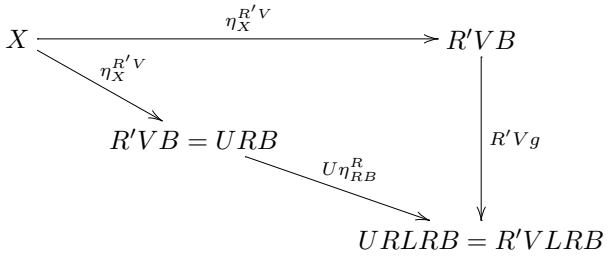
- $U \circ R = R' \circ V$,
- $R \circ L \cong id$ and $R' \circ L' \cong id$,
- L is left adjoint to R, and
- L' is left adjoint to R'.

Then

1. If $B \in \mathbf{B}$ is V-free over $L'X$ for some $X \in \mathbf{X}$, then RB is U-free over X.
2. $A \in \mathbf{A}$ is U-free over $X \in \mathbf{X}$ iff LA is V-free over $L'X$.
3. If $B \in \mathbf{B}$ is strongly persistently V-free and VB is in the image of L', then RB is strongly persistently U-free.
4. Further assume that $L' \circ U = V \circ L$. Then $A \in \mathbf{A}$ is strongly persistently U-free iff LA is strongly persistently V-free.

Proof: (1)

By Proposition C.10, η^R and ε^L_L are isomorphisms, and L is full. Assume that $B \in \mathbf{B}$ is V-free over $L'X$, i.e. there is a V-universal arrow $\eta^V_{L'X} : L'X \longrightarrow VB$. By composition with the R'-universal arrow $\eta^{R'}_X : X \longrightarrow R'L'X$ we get an $R'V$-universal arrow $\eta^{R'V}_X = R'\eta^V_{L'X} \circ \eta^{R'}_X : X \longrightarrow R'VB$. By its universality, there is a morphism $g : B \longrightarrow LRB$ with $R'Vg \circ \eta^{R'V}_X = U\eta^R_{RB} \circ \eta^{R'V}_X$.



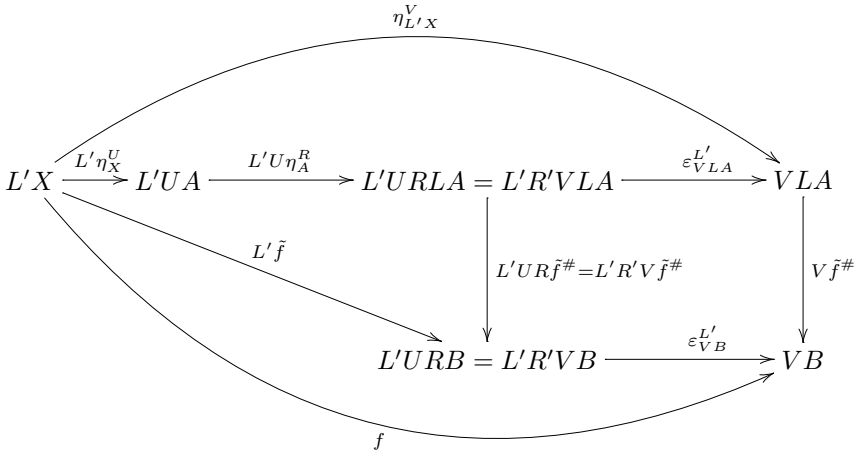
Now $R'V\varepsilon^L_B \circ R'Vg \circ \eta^{R'V}_X = UR\varepsilon^L_B \circ U\eta^R_{RB} \circ \eta^{R'V}_X = \eta^{R'V}_X$. By universality of $\eta^{R'V}_X$, we get $\varepsilon^L_B \circ g = id$, and thus also $\varepsilon^L_B \circ g \circ \varepsilon^L_B = \varepsilon^L_B$. Since L is full, $g \circ \varepsilon^L_B$ is the L-image of an $\mathbf{A}$-morphism. By co-universality of ε^L_B, we get $g \circ \varepsilon^L_B = id$. Thus, $g\colon B \longrightarrow LRB$ is an isomorphism. But then, $\eta^U_X := \eta^{R'V}_X\colon X \longrightarrow URB$ can be shown to be a U-universal arrow as follows: If $f\colon X \longrightarrow UA$ is an $\mathbf{X}$-morphism, by universality of $\eta^{R'V}_X$, there is a unique morphism $f^\#\colon B \longrightarrow LA$ satisfying $R'Vf^\# \circ \eta^{R'V}_X = U\eta^R_A \circ f$. Now $(\eta^R_A)^{-1} \circ R(f^\# \circ g^{-1}) \circ \eta_{RB}$ is a morphism from RB to A with $U((\eta^R_A)^{-1} \circ R(f^\# \circ g^{-1}) \circ \eta_{RB}) \circ \eta^{R'V}_X = U(\eta^R_A)^{-1} \circ UR(f^\#) \circ UR(g^{-1}) \circ U\eta^R_{RB} \circ UR\varepsilon^L_B \circ URg \circ \eta^{R'V}_X = U(\eta^R_A)^{-1} \circ UR(f^\#) \circ UR(g^{-1}) \circ URg \circ \eta^{R'V}_X = U(\eta^R_A)^{-1} \circ UR(f^\#) \circ \eta^{R'V}_X = U(\eta^R_A)^{-1} \circ U\eta^R_A \circ f = f$. Since g is an isomorphism, uniqueness follows from that of $f^\#$. □

Proof of (2), "$\Longrightarrow$":

Let $\eta^U_X\colon X \longrightarrow UA$ be U-universal, and $\eta^R_A\colon A \longrightarrow RLA$ be R-universal. Then

$$\eta^V_{L'X} := \; L'X \xrightarrow{L'\eta^U_X} L'UA \xrightarrow{L'U\eta^R_A} L'URLA = L'R'VLA \xrightarrow{\varepsilon^{L'}_{VLA}} VLA$$

is a V-universal arrow: Let $f\colon L'X \longrightarrow VB$ be an $\mathbf{X}$-morphism. By co-universality of $\varepsilon^{L'}_{VB}$, there is some unique $\tilde{f}\colon X \longrightarrow R'VB = URB$ with $\varepsilon^{L'}_{VB} \circ L'\tilde{f} = f$. By UR-universality of $U\eta^R_A \circ \eta^U_X$, there is some unique $\tilde{f}^\#\colon LA \longrightarrow B$ with $UR\tilde{f}^\# \circ U\eta^R_A \circ \eta^U_X = \tilde{f}$. By also considering the commutativity of the square (due to naturality of $\varepsilon^{L'}$), $\tilde{f}^\#$ is the unique morphism from LA to B with $V\tilde{f}^\# \circ \eta^V_{L'X} = f$.



□

Proof of (2), "$\Longleftarrow$":

Follows from (1) with $B = LA$, since $RLA \cong A$. □

Proof of (3): Follows from (1) with X such that $L'X = VB$, where Proposition C.10 ensures that the unit constructed in the proof is an isomorphism, which leads to strongly persistent freeness by Proposition C.1. □

Proof of (4):

Follows from (2) with $X = UA$, since by assumption, $L'UA = VLA$, where $\varepsilon^{L'}_{VL} = \varepsilon^{L'}_{L'U}$ and proposition C.10 ensure that the unit constructed in the proof is an isomorphism, which leads to strongly persistent freeness by Proposition C.1. □

C.4 Rules for Free Theorem Links in Development Graphs

Although there is no general approach to verify that an extension of a specification is conservative (or monomorphic, or definitional), several schemes for extending specifications have been developed in the past which guarantee these properties by construction. We only very informally list some possible rules here:

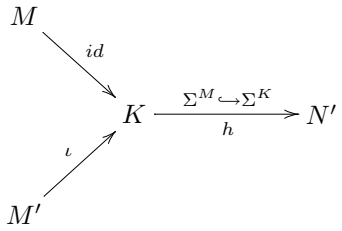
- extensions declaring new signature elements are conservative, provided the new symbols are not constrained in any way (by axioms, by requirements on the subsort and overloading relations, etc.) to be related to old symbols, and the new symbols themselves are constrained by a positive theory (i.e. not involving negation),
- free datatypes are monomorphic extensions of the local environment in which they are introduced,
- structured free Horn theories are monomorphic extensions,
- subsort definitions are definitional extensions, and
- inductive definitions over free datatypes are definitional extensions.

There is no hope to tackle freeness in an institution independent way either. But is is possible to define a CASL-specific elimination oracle for free definition links. It basically introduces a new node that is used to mimic the quotient term algebra construction.

We cannot expect to get proof support for free specifications that is independent of the underlying logical system; hence, this section assumes that we are working in the CASL logic.

Elimination Rule for Free Definition Links

Given a free definition link $M \xRightarrow[free]{\sigma} N$ with $\sigma: \Sigma \longrightarrow \Sigma^M$ such that M is flattenable and $Th_{\mathcal{S}}(M)$ is in Horn form, replace it with $M \xRightarrow{id} K$, where K is constructed as follows:



Let $\Sigma^{M'}$ be a copy of Σ^M with all function symbols made total, and ι be a renaming of $\Sigma^{M'}$ that makes it disjoint from Σ^M. Then Σ^K is Σ^M united with $\iota(\Sigma^{M'})$ and augmented by new operations

$$\begin{array}{l} make_s: s \longrightarrow \iota(s) \quad (s \in Sorts(\Sigma)) \\ h: \iota(s) \longrightarrow ? s \quad (s \in Sorts(\Sigma^M)) \end{array}$$

Ψ^K consists of

- an axiomatization of the sorts in $\iota(\Sigma^{M'})$ as free types with all operations as constructors (i.e. including the $make_s$, if sort $\iota(s)$ is axiomatized),
- the following homomorphism equations:

$$\begin{array}{l} h\langle \iota(f_{w,s})\langle x^1_{s_1}, \ldots, x^n_{s_n}\rangle\rangle = f_{w,s}\langle h\langle x^1_{s_1}\rangle, \ldots, h\langle x^n_{s_n}\rangle\rangle \\ \qquad\qquad \text{for } f: w \longrightarrow s \in \Sigma^M, w = s_1 \ldots s_n \\ \iota(p_w)\langle x^1_{s_1}, \ldots, x^n_{s_n}\rangle \Leftrightarrow p_w\langle h\langle x^1_{s_1}\rangle, \ldots, h\langle x^n_{s_n}\rangle\rangle \\ \qquad\qquad \text{for } p: w \in \Sigma^M, w = s_1 \ldots s_n \end{array}$$

- surjectivity of the homomorphism functions:

$$\forall y : s. \exists x : \iota(s). h(x) \stackrel{e}{=} y \quad (s \in Sorts(\Sigma^M))$$

- the kernel of h is the least partial predicative congruence[2] satisfying $Th_{\mathcal{S}}(M)$. This is expressed with the following family of axioms. For any family of Σ^K-formulas $(\Phi_s(x_{\iota(s)}, y_{\iota(s)}))_{s\in Sorts(\Sigma^M)}$ in two free variables (coding the binary relations for the sorts) and family of Σ^K-formulas $(\Phi_{p:w}(x^1_{\iota(s_1)}, \ldots, x^n_{\iota(s_n)}))_{p:w\in\Sigma^M, w=s_1\ldots s_n}$ in n free variables (coding the n-ary relations for the predicate symbols), we take the axiom

$$symmetry \wedge transitivity \wedge congruence \wedge satThM \Rightarrow largerThenKerH$$

Here, *symmetry* stands for

$$\bigwedge_{s\in Sorts(\Sigma^M)} \forall x : \iota(s), y_{\iota(s)}.\Phi_s(x_{\iota(s)}, y_{\iota(s)}) \Rightarrow \Phi_s(y_{\iota(s)}, x_{\iota(s)}),$$

transitivity stands for

$$\bigwedge_{s\in Sorts(\Sigma^M)} \forall x : \iota(s), y : \iota(s), z : \iota(s).\Phi_s(x_{\iota(s)}, y_{\iota(s)}) \wedge \Phi_s(y_{\iota(s)}, z_{\iota(s)}) \Rightarrow \Phi_s(x_{\iota(s)}, z_{\iota(s)}),$$

congruence stands for

$$\begin{array}{l}\bigwedge_{f:w\longrightarrow s\in\Sigma^M} \forall \bar{x} : \iota(w), \bar{y} : \iota(x)\,. \\ \qquad D(\iota(f_{w,s})\langle \bar{x}_{\iota(w)}\rangle) \wedge D(\iota(f_{w,s})\langle \bar{y}_{\iota(w)}\rangle) \wedge \Phi_w(\bar{x}_{\iota(w)}, \bar{y}_{\iota(w)}) \\ \qquad \Rightarrow \Phi_s(\iota(f_{w,s})\langle \bar{x}_{\iota(w)}\rangle, \iota(f_{w,s})\langle \bar{y}_{\iota(w)}\rangle)\end{array}$$

and *satThM* stands for

$$Th_{\mathcal{S}}(M)[\stackrel{e}{=}/\Phi_s;\ p : w/\Psi_{p:w};\ D(t)/\Phi_s(t,t);\ t = u/\Phi_s(t,u) \vee (\neg\Phi_s(t,t) \wedge \neg\Phi_s(u,u))]$$

where for a set of formulas Ψ, $\Psi[sy_1/sy'_1;\ \ldots sy_n, sy'_n]$ denotes the simultaneous substitution of sy'_i for sy_i in all formulas of Ψ (while possibly instantiating the meta-variables t and u).

Finally *largerThenKerH* stands for

$$\begin{array}{l}\bigwedge_{s\in Sorts(\Sigma^M)} \forall x : \iota(s), y : \iota(s).h(x) \stackrel{e}{=} h(y) \Rightarrow \Phi_s(x_{\iota(s)}, y_{\iota(s)}) \\ \wedge \bigwedge_{p:w\longrightarrow\in\Sigma^M} \forall \bar{x} : \iota(w).\iota(p : w)\langle \bar{x}_{\iota(w)}\rangle \Rightarrow \Psi_{p:w}(\bar{x}_{\iota(w)})\end{array}$$

Should we axiomatize things in Σ^M to be term generated as well? This should already follow from generatedness of things in $\iota(\Sigma^{M'})$ and surjectivity of the h's ...
Aha, and we have to add a definedness condition for f... And also definedness of ops for congruence axioms!

Sketch of soundness proof:

Basically, K mimics the quotient term algebra construction. The term algebra is held in $\iota(\Sigma^{M'})$ (axiomatized to be an absolutely free algebra), and the (partial!) quotient homomorphism is held in the operation h.

Simplifications of the rule are possible if we know something about sufficient completeness (cf. the last rule for definitional extensions) – say something more here!!! In particular, state simpler versions of this rule for free datatypes (1) without axioms; (2) with just new predicates (3) with new operations introduced in a sufficiently complete way

Introduction Rule for Free Theorem Links

$$\frac{K = \stackrel{\theta}{=} \Rightarrow N}{M = \underset{free}{\stackrel{\theta}{=}} \underset{\sigma}{\Rightarrow} N}$$

[2] A partial predicative congruence consists of a symmetric and transitive binary relation for each sort and a relation of appropriate type for each predicate symbol.

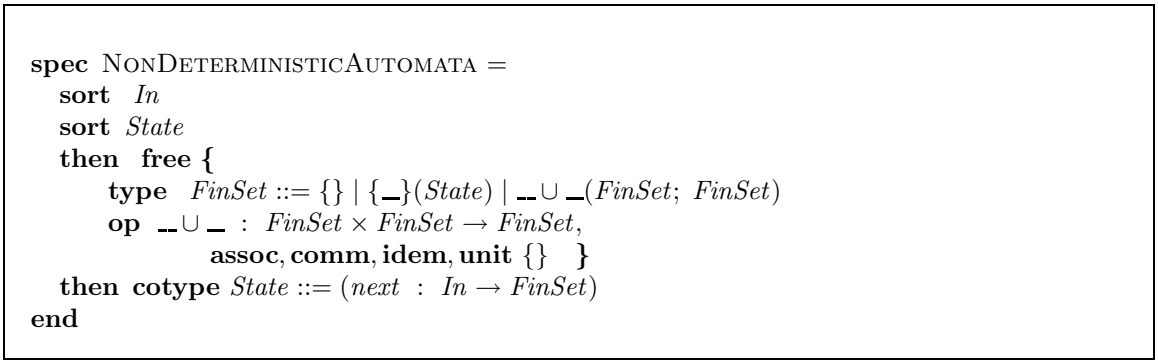


Figure C.1: Specification of non-deterministic automata.

where K is constructed from M and σ similarly as in the elimination rule for free definition links — the formulas Φ_s and $\Phi_{p:w}$ now have to be replaced by new predicate symbols. This is in much the same way related to the above rule as the rule **Sortgen-Intro** is related to the rule **Induction** in the calculus for many-sorted specifications (see Chap. IV.2 of [CoF04]).

C.5 Cofree specifications

CoCasl's **cofree { ... }** construct dualizes the **free { ... }** construct by restricting the model class of a specification to the cofree, i.e. final ones. This generalizes the **cofree cotypes** construct to arbitrary specifications; in particular, final models may be restricted by axioms (e.g. as in Figure C.3 below).

More precisely, the semantics of **cofree** is defined as follows:

Definition C.12 If Sp_1 is a specification with signature Σ_1, then the models of Sp_1 **then cofree** $\{Sp_2\}$ are those models M of Sp_1 **then** Sp_2 that are *fibre-final* over $M|_{\Sigma_1}$ w.r.t. the reduct functor $\_|_{\Sigma_1}$. Here, fibre-finality means that M is the final object in the fibre over $M|_{\Sigma_1}$. The fibre over $M|_{\Sigma_1}$ is the full subcategory of $Mod(Sp_1$ **then** $Sp_2)$ consisting of those models whose Σ_1-reduct is $M|_{\Sigma_1}$.

This definition deviates somewhat from the semantics of **free** in that the latter postulates initiality, i.e. that M is free over $M|_{\Sigma_1}$ with $\_|_{\Sigma_1}$-universal arrow $id : M|_{\Sigma_1} \to M|_{\Sigma_1}$, which is stronger than fibre-initiality of M. We will see shortly that the more liberal semantics for **cofree** is essential in cases where sorts from the local environment occur as argument sorts of selectors. Call a sort from the local environment an *output sort* if it occurs only as a result type of selectors. In the cases of interest, a more general co-universal property concerning, in the notation of the above definition, morphisms of Σ_1-models that are the identity on all sorts except possibly the output sorts, follows from fibre-finality.

The **cofree cotypes** construct is equivalent to **cofree { cotypes ... }**:

Proposition C.13 If DD is a sequence of cotype declarations, then

cofree **{** ***cotypes*** DD **}** and ***cofree cotypes*** DD

have the same semantics.

Proof: Thanks to the fact that the semantics of the **cofree** construct is defined via *fibre*-finality, the interpretations of additional parameters for observers are fixed (in a given fibre). Hence, we can apply currying as in Def. 3.22. The result then follows from Props. 3.23 and 3.27. □

By contrast, the use of **cofree { types ... }** should be avoided:

spec FINALMOORE2 =
  sorts *In*, *Out*
  then **cofree** {
  cotype *State* ::= (*next* : *In* → *State*; *observe* : *Out*)
  }
end

Figure C.2: Structured cofree specification of the final Moore automaton.

spec BITSTREAM3 =
  free **type** *Bit* ::= *0* | *1*
  then **cofree** {
    cotype *BitStream* ::= (*hd* : *Bit*; *tl* : *BitStream*)
    ∀*s* : *BitStream*
    • $hd(s) = 0 \wedge hd(tl(s)) = 0 \Rightarrow hd(tl(tl(s))) = 1$ }
end

Figure C.3: Structured cofree specification of bit streams in COCASL.

Example C.14 The specification

free type *Bool* ::= *false* | *true*
then
cofree { **type** *T* ::= *c1*(*s1* :?*Bool*) | *c2*(*s2* :?*Bool*) }

is inconsistent. Indeed, by applying the uniqueness part of finality to a model of the unrestricted type where T has an element on which both selectors are undefined (this is allowed for types but not for cotypes), one obtains that any model of the cofree type would be a singleton; however, singleton models fail to satisfy the finality property e.g. for the model of the unrestricted type where T is $Bool \times Bool$ and the selectors are the projections.

As an example for the significance of the relaxation of the cofreeness condition, consider the specification of Moore automata as given in Figure C.2. Here, the observer *next* depends not only on the state, but additionally on an input letter.

In the standard theory of coalgebra, *next* would become a higher-order operation $next\colon State \longrightarrow State^{In}$, and the cofree coalgebra indeed yields the final automaton showing all possible behaviours - but only for a *fixed* carrier for *In* (the inputs). The carrier for *Out* is also regarded as fixed; however, one can show that the co-universal property holds also for morphisms that act non-trivially on *Out*. If the semantics of **cofree** required actual cofreeness, i.e. a couniversal property also for morphisms that act non-trivially on *In*, the specification would be inconsistent!

Let us now come to a further modification of the stream example. If the axiom were omitted in the specification in Figure C.3, the model class would be the same as that in Figure 3.9, instantiated to the case of bits as elements. *With* the axiom, the streams are restricted to those where two 0's are always followed by a 1. Again, this is unique up to isomorphism.

It is straightforward to specify iterated free/cofree constructions, similarly as in [Rei00]. Consider e.g. the specification of lists of streams of trees in Figure C.4. Alternatively, one could have used structured free and cofree constructs as well:

$$SP \textbf{ then free } \{SP_1\} \textbf{ then cofree } \{SP_2\} \textbf{ then free } \ldots$$

```
spec ListStreamTree [sort Elem] =
  free type
      Tree ::= EmtpyTree
             | Tree(left :?Tree; elem :?Elem; right :?Tree)
  cofree cotype
      Stream ::= (hd : Tree; tl : Stream)
  free type
      List ::= Nil | Cons(head :?Stream; tail :?List)
end
```

Figure C.4: Nested free and cofree (co)types.

```
spec FinalNonDeterministicAutomaton =
  sort In
  then cofree {
      sort State
      then free {
          type Set ::= {} | {_}(State) | __ ∪ __(Set; Set)
          op _ ∪ __ : Set × Set → Set,
                 assoc, comm, idem, unit {}  }
      then cotype State ::= (next : In → Set)  }
end
```

Figure C.5: A free type *within* a cofree specification.

Note that also in the latter case, there won't be any **free** *within* a **cofree** or vice versa. An example for **free** *within* **cofree** is shown in Figure C.5. This specification extends the specification of non-deterministic automata of Figure C.1 by an outer (structured) cofreeness constraint, so that its model class now consists only of models where the cotype *State* is 'the' final non-deterministic automaton (determined uniquely up to isomorphism) over the interpretation of *In* rather than the class of all non-deterministic automata. Here, like in Figure C.1, the inner **free** has to be a structured one, since finite sets cannot be specified as **free type** directly. In principle, **free** and **cofree** can be nested arbitrarily; however, care must be taken to ensure that this does not lead to inconsistencies. A general consistency criterion that covers nestings of the type used in Figure C.5 is given in [MSRR].

Proof support for specifications of the form **cofree** $\{SP\}$ is not at all trivial. If SP is flattenable and axiomatized within the modal logic, we can proceed similarly to the case of **cofree types**: the model of **cofree** SP is a subcoalgebra of the coalgebra of *all* behaviours (as specified by the corresponding **cofree type**), namely the largest subcoalgebra that satisfies the modal axioms.

More complex examples, such as nondeterministic automata or trees with unbounded branching, involve a free specification of output sorts of selectors (like lists or sets) within a **cofree** {...}. Here, in a first step, we proceed as above and encode the cofree type over the absolutely free type (only the branching may now be infinite, being determined by a datatype). Then the cofree type over the relatively free type is obtained as the quotient modulo the largest congruence [GS]. In terms of tool support, this means the following:

- Equality of elements in the cofree datatype is obtained as before by coinductive reasoning (or via terminal sequence induction [Pat02]), the difference with the absolutely free case being

that the formulas in the free specification (e.g. associativity, commutativity, and idempotence in the case of finite sets) are now available for such proofs.

- Distinctness of elements is shown, again as before, by establishing that the behaviours are different. Here, the encoding of free specifications comes in: distinctness of two elements of a relatively free type is shown by separating the two elements by a congruence.

Remark C.15 Above, we have seen two cases where free specifications within cofree specifications allow good technical handling:

- the output sorts of selectors for a cofree datatype may be given by a free specification, which is handled as described above;
- the modal formulas that restrict the elements of the cofree type may involve freely (or cofreely) specified predicates, which are dealt with in Isabelle by means of least and greatest fixed points.

Beyond these two cases, the situation remains somewhat unclear. E.g., the following specification is inconsistent:

spec FinalElement = Bool **then**
cofree {
  free type *Unit* ::= *1*
  op *el* : *Unit* → *Bool* **}**

This seems to indicate that input sorts should not be restricted by equational axioms (the freeness constraint can be replaced by an equation here), or in fact by anything else except modal formulas; this is in agreement with suggestions made in [Kur01b]. On the other hand, observe that constraining output sorts is more or less mandatory, e.g. when specifications using freely specified finite sets as observations.

Also, a proof principle for free specifications containing cofree specifications seems to be much harder to obtain. Here, we propose to avoid the outer free specification and use a generation axiom plus some characterization of equality by suitably chosen observers instead.