Proof Support for RAISE

—

by a Reuse Approach based on Institutions

Morten P. Lindegaard

Kongens Lyngby 2004
IMM-PHD-2004-132

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk

IMM-PHD: ISSN 0909-3192

Denne afhandling er et led i opfyldelse af betingelserne for erhvervelse af ph.d.-graden ved Danmarks Tekniske Universitet.

Studierne blev udført på Informatik og Matematisk Modellering med lektor Anne E. Haxthausen som hovedvejleder og lektor Michael R. Hansen som vejleder under hovedvejleders orlov – begge fra Informatik og Matematisk Modellering.

Undertegnede var ansat som lønnet ph.d.-stipendiat (DTU stipendium).

This thesis is submitted in partial fulfillment of the requirements for the Ph.D. degree at Technical University of Denmark.

The studies were conducted at Informatics and Mathematical Modelling with Associate Professor Anne E. Haxthausen as main supervisor and Associate Professor Michael R. Hansen as supervisor during main supervisor's leave – both from Informatics and Mathematical Modelling.

The author was holder of a scholarship from Technical University of Denmark.

Kgs. Lyngby, 28 November 2003

Morten P. Lindegaard

Resumé

Formelle metoder er matematisk baserede metoder til udvikling af programmel. Sådanne metoder involverer som regel, at programmel og krav beskrives i et formelt specifikationssprog, hvorefter det verificeres, at programmellet opfylder de ønskede krav.

RAISE er en formel metode med det tilhørende specifikationssprog RSL og et bevissystem. Der findes datamatbaserede bevisværktøjer til RAISE, men en højere grad af automation er ønskværdig.

Isabelle/HOL er en bevisfører for højereordens logik (HOL). Bevisføreren er en instantiering af den generiske bevisfører Isabelle, der tilbyder en passende grad af automation og fleksibilitet.

Med henblik på at benytte bevisføreren Isabelle/HOL i forbindelse med RAISE metoden, betragtes oversættelse fra RSL til HOL. Oversættelsen baseres på institutioner, der formaliserer det uformelle begreb "logisk system".

Institutioner og morfier mellem institutioner gennemgås sammen med specifikationer over institutioner og modelteoretisk semantik af specifikationer. Begrebet "let institutionscomorfi" introduceres som en modifikation af velkendte institutionscomorfier, og det bevises, at en let institutionscomorfi muliggør sund genbrug af bevisførere, når den har visse egenskaber. Derudover introduceres begrebet "løsere semantik af specifikationer" som en modelteoretisk beskrivelse af semantikken af RSL-specifikationer, og et ækvivalensresultat bevises.

Der defineres en institution for en applikativ, deterministisk delmængde af RSL, der benævnes "mRSL". Herefter præsenteres en velkendt institution for HOL, og Isabelle/HOL gennemgås kort. Der defineres en let institutionscomorfi fra mRSL-institutionen til HOL-institutionen, hvorved der fås en oversættelse fra mRSL til Isabelle/HOL, og det bevises, at den definerede lette institutionscomorfi har de egenskaber, der muliggør sund genbrug af bevisføreren Isabelle/HOL.

Brugen af den definerede oversættelse beskrives i forbindelse med tre eksempler: logiske kredsløb, en jernbaneoverskæring og en indkodning af varighedskalkylen i RSL.

Abstract

Formal methods are mathematically based methods for developing software. Such methods usually involve that software and requirements are specified in a formal specification language, after which it is verified that the software meets the requirements.

RAISE is a formal method with the associated specification language RSL and a proof system. Computer-based proof tools are available for RAISE, but a higher degree of automation is desired.

Isabelle/HOL is a proof assistant for higher-order logic (HOL). It is an instantiation of the generic proof assistant Isabelle which offers a suitable degree of automation and flexibility.

In order to use the Isabelle/HOL proof assistant for the RAISE method, translation from RSL to HOL is considered. The translation is based on institutions which formalize the informal notion of a "logical system".

Institutions and morphisms between institutions are presented together with specifications over institutions and model-theoretic semantics of specifications. The concept of "light institution comorphisms" is introduced as a modification of well-known institution comorphisms, and it is proved that a light institution comorphism enables sound reuse of proof assistants when it has certain properties. Moreover, the concept of "looser semantics of specifications" is introduced as a model-theoretic description of the semantics of RSL specifications, and an equivalence result is proved.

An institution for an applicative, deterministic subset of RSL, referred to as "mRSL", is defined. Then, a well-known institution for HOL is presented, and Isabelle/HOL is briefly described. An institution comorphism from the mRSL institution to the HOL institution is defined, providing a translation from mRSL to Isabelle/HOL, and it is proved that the light institution comorphism has the properties that enable sound reuse of the Isabelle/HOL proof assistant.

The use of the translation is described in connection with three examples: logical circuits, a generalized railway crossing, and an encoding of Duration Calculus in RSL.

Acknowledgements

First of all, I would like to thank my main supervisor, Anne E. Haxthausen, for her support, comments on my work, and encouragement. I appreciate it very much.

I would also like to thank Michael R. Hansen, who was my supervisor during Anne's leave, and Hanne Riis Nielson and Paul Fischer, who were members of my "follow-up-group".

In connection with course activities, I would especially like to thank Lars Birkedal, Sebastian Skalberg, Noah Torp-Smith, and Nikolaj Oldager.

My average day would not have been the same without my office mates – first Thomas Marthedal Rasmussen and later Henrik Pilegaard. I would like to thank both of them for discussions and creating a pleasant atmosphere.

I would also like to thank the rest of the people at Informatics and Mathematical Modelling, who has helped me during the years or in other ways has left a positive impression.

I had an enjoyable time visiting University of Bremen. Especially, I would like to thank Till Mossakowski, Bettina Buth, Markus Roggenbach, Lutz Schröder, and Christoph Lüth.

Finally, I would like to thank Jan Storbank Pedersen, Xia Yong, and Paul Taylor for providing various documents and a package for drawing commuting diagrams in LaTeX.

Looking back, I have fond memories of meeting a lot of wonderful people during my studies. Only some of them have been mentioned here.

Contents

CHAPTER 1

Introduction

Software is everywhere. Not only do we use software on our personal computer, we also rely on embedded software in consumer electronics, traffic control systems, and communication devices every day. Thus, reliability of software is important, especially for safety-critical systems such as train control systems for which failures may have severe consequences.

Formal methods are mathematically based development methods that ensure a high degree of confidence in the software. A formal method usually encompasses a formal specification language in which software as well as desired properties of software (requirements) can be described (specified). It may then be verified that the software – as it is specified – has the desired properties. There are two main approaches to verification: *model checking* and *theorem proving*.

In model checking, a system is modelled as a (finite) state transition graph and requirements are expressed in a suitable logic. Verifying that the modelled system meets the requirements is done by exploring the state space of the system. An advantage of this approach is that the task is fully automated, but a disadvantage is that some systems may have state spaces that are practically too large to handle – this situation is known as *state-space explosion*.

In theorem proving, a proof system consisting of a collection of proof rules is used to deduce that the software has the desired properties. This task is supported by computer-based tools to avoid erroneous applications of rules that may occur in pen-and-paper proofs. The main advantages of using theorem proving rather than model checking is that there are no state-space explosion problems and properties of generic systems can be verified. An example is the generic specification of railway interlocking systems in [40]. A disadvantage of the theorem proving approach is that the task is not fully automated in general.

RAISE [54, 55] is an acronym for *Rigorous Approach to Industrial Software Engineering.* It is a mathematically based method which has shown to be useful in the industrial development of various kinds of software systems. RAISE has an associated specification language – the RAISE Specification Language (RSL) – and an important part of the RAISE method is to verify desired properties of specifications by means of theorem proving. This is currently a labour expensive task and proof support with a higher degree of automation is desired.

Higher-order logic (HOL) was originally introduced as *a simple theory of types* by Church [15]. A thorough account of higher-order logic is given in [1], and a theorem proving environment is presented in [27]. Higher-order logic has especially shown to be useful for hardware verification, but it has also shown to be applicable in many other areas of computer science and mathematics. Based on [27], the generic proof assistant *Isabelle* [49, 50, 51] has been instantiated to provide proof support for higher-order logic. This instantiation is referred to as *Isabelle/HOL* [46].

In this thesis, an approach to providing increased proof support for the RAISE development method by translating RSL specifications and proof obligations to higher-order logic and making proofs with the Isabelle/HOL proof assistant is investigated. The translation is based on a formal foundation, ensuring that results obtained using the Isabelle/HOL proof assistant on translated specifications and proof obligations hold for the original RSL specifications and proof obligations.

In the remaining part of this chapter, we briefly touch upon specification languages, theorem proving systems, and the notion of logical systems. Moreover, the aims of this thesis and the main results are summarized, and an out-line of the remaining chapters is given.

1.1 Specification Languages

A specification language is a formal language in which software (algorithms and datatypes) and requirements may be described.

There is a vast number of specification languages with different features available. Some languages are tailored towards *algebraic specification* in which software modules are viewed as algebras. Algebraic specification is especially applicable for specifying abstract datatypes with data and operations. An example of an algebraic specification language is CASL [4, 16], which is an acronym of *Common Algebraic Specification Language.*

Model-oriented specifications rely on well-known datatypes such as integers and Booleans. An example of a model-oriented specification language is VDM-SL (also known as Meta-IV) which is used in the Vienna Development Method (VDM) [36].

The RAISE Specification Language is a wide-spectrum language featuring constructs known from mathematics, logic, process algebra, and programming languages. Consequently, it features both algebraic and model-oriented specifications, and it may be used for writing abstract specifications as well as specifications that are very close to being actual programs.

1.2 Theorem Proving Systems

By *theorem proving systems* we mean computer-based tools that support theorem proving for formal languages.

In some theorem proving systems, proofs are developed manually and the tools checks that rules are not applied erroneously. Such tools are referred to as *proof checkers*. For some logics, however, proofs may be found automatically, in which case we can talk about *automated theorem provers*. Unfortunately, this is not the case for higher-order logics for which theorem provers need human guidance to find proofs. Consequently, we talk about *interactive theorem provers* that are not fully automatic but do provide a higher degree of mechanization than mere proof checkers.

The `eden` *justification editor* is a theorem proving system for RAISE. Proofs (or *justifications*) are developed primarily by manual application of proof rules, in which the editor provides the user with a list of applicable proof rules and the user selects the rules to be applied. Some simplification (automatic rewriting of expressions) is available, though.

Isabelle is a generic theorem proving system, i.e., Isabelle has a meta-logic in which object-logics may be described. Instantiating Isabelle to an object-logic amounts to describing the logic in the meta-logic, and this has been done for a series of logics including first-order logic, set theory, and higher-order logic. Instantiations of Isabelle are described in [52, 53, 47, 46].

Isabelle is one of many interactive theorem proving systems. The HOL system [27] is an interactive theorem proving system for classical higher-order logic, and it is a direct descendant of the LCF system [26]. The approach to theorem proving is to represent theorems by values of an abstract type and represent inference rules as ML functions such that new theorems are created by applying ML functions to axioms or previously generated theorems. The higher-order logic instance of Isabelle [46] is based on the HOL system. Another example of an interactive theorem proving system is PVS [48] which consists of a specification language and tool support including a theorem prover.

1.3 Logical Systems

The term "logical system" is often used in connection with formal specification languages and logics, and the concept of *institutions* formalizes the informal notion of a logical system. People may not agree on what is meant by "logical system", but we can give a clear definition of what is meant by "institution".

Institutions and various kinds of mappings between institutions provide a framework for comparing and relating logical systems. Furthermore, mappings between institutions provide formal foundations for translating specifications of one institution to specifications of another institution.

The concept of institutions is, of course, not the only way of formalizing a logical system. Institutions represent a model-oriented view in the sense that an institution contains a satisfaction relation between models and sentences. The concept of *general logics* [42] also includes an *entailment relation* $\Gamma \vdash \varphi$ between a set of sentences Γ and a sentence φ. Such a relation is typically defined by a collection of inference rules (also called proof rules) that also gives rise to a notion of *proof*.

1.4 Aims of Thesis

As earlier mentioned, the possibility of providing increased proof support for the RAISE development method by translating RSL specifications and proof obligations to higher-order logic and making proofs with the Isabelle/HOL proof assistant is considered in this thesis.

To base the translation on a formal foundation, the approach is to define an institution for RSL, and define a mapping from the institution of RSL into the institution of HOL. This mapping is then used to translate RSL specifications and proof obligations to HOL specifications and proof obligations such that the Isabelle/HOL proof assistant can be used.

The use of Isabelle/HOL is required to be *sound* – if a translated proof obligation can be proved for a translated specification, then the original proof obligation must hold for the original specification. Consequently, the mapping from the institution of RSL into the institution of HOL needs certain properties.

1.5 Main Results

The main contributions in this thesis are the definition of an institution for a subset of RSL and a mapping from this institution to an institution of higher-order logic. The mapping is a *light institution comorphism* and it enables translation of basic RSL specifications and proof obligations to specifications and proof obligations in higher-order logic such that the Isabelle/HOL proof assistant can be reused. The concept of *light institution comorphisms* is a modification of well-known *institution comorphisms* [24] that are also known as *plain maps* [42] or *simple institution representations* [45].

Another contribution is a formalization of the concept of *looser semantics* of specifications which is used to give an abstract description of the semantics of RSL specifications. Looser semantics is a generalization of the well-known *loose semantics* [18, 41], and we prove that loose and looser semantics are equivalent with respect to semantic consequence.

Furthermore, a series of RSL proof rules has been translated to Isabelle/HOL and proved as theorems using Isabelle. This not only provides helpful theorems but also formally proves that these RSL proof rules are sound with respect to the denotational semantics.

Finally, the use of the proposed translation for practical applications has been investigated in example applications and the experiences are reported.

1.6 Organization

The remaining chapters of this thesis are briefly described below.

Chapter 2 introduces the formal background. Basic category theory, institutions, specifications and their semantics, mappings between institutions, and a result about reusing theorem provers are reviewed.

Chapter 3 provides an overview of the RAISE Specification Language and the RAISE Development Method.

Chapter 4 defines an institution for an applicative, deterministic subset of RSL referred to as *mRSL*.

Chapter 5 describes higher-order logic (HOL). The description includes the definition of an institution for HOL and an introduction to the proof assistant Isabelle/HOL.

Chapter 6 defines a light institution comorphism from the institution of mRSL to the institution of HOL. It is also explained how the light institution comorphism is used for translating specifications.

Chapter 7 discusses experiences from using the light institution comorphism and Isabelle/HOL. General considerations and three applications are discussed.

Chapter 8 concludes the thesis by summarizing and evaluating the results.

CHAPTER 2

Institutions as Framework for Formal Specifications

This chapter introduces the theoretical framework of institutions together with basic notions of specifications and semantics of specifications.

The presentation assumes some knowledge about basic category theory. However, basic definitions from category theory are presented to make the presentation self-contained.

Institutions and mappings between institutions are described in a variety of papers. The concept was originally introduced by Goguen and Burstall in [23, 25], and has later been considered in, e.g., [60]. For a contemporary account of institutions and especially mappings between institutions, see [24].

We introduce yet another kind of mapping between institutions, a light institution comorphism. As the name suggests, this kind of mapping strongly resembles an institution comorphism. However, it does not have all the properties required for it to be an institution comorphism.

We present some well-known definitions concerning semantic consequence and loose semantics of specifications. They are, e.g., found in [41][1]. In addition, we introduce the concept of looser semantics, the purpose of which is to provide a theoretical description of the semantics of RSL specifications which is later considered in Chapter 3. Loose and looser semantics are briefly compared, and we prove a theorem stating that loose and looser semantics are equivalent with respect to semantic consequence.

[1]Algebraic specification and semantics of specifications is introduced in [41] without using category theory.

The chapter is concluded with an important theorem that enables reuse of theorem provers by using a mapping between institutions to translate specifications and proof obligations. We present and prove a version of the theorem for the light institution comorphisms. The theorem is found in its original form in [25].

2.1 Basic Definitions and Notational Conventions

In this section, we present basic definitions from category theory and notational conventions. The definitions are from [62].

2.1.1 Objects, Morphisms, Categories

Categories contain objects and morphisms.

Definition 2.1 A *category* consists of a collection of *objects* and a collection of *morphisms* such that:

- Each morphism has a *domain* and a *codomain* that are objects. We write $f : X \longrightarrow Y$ or $X \xrightarrow{f} Y$ when f is a morphism with the domain X and the codomain Y, and we say that f is a morphism *from* X *to* Y.
- For each pair of morphisms $f : X \longrightarrow Y$ and $g : Y \longrightarrow Z$, the *composition* of f and g is defined as a morphism $(g \circ f) : X \longrightarrow Z$.
- The composition operator $\circ$ is *associative*, i.e., for morphisms $f : X \longrightarrow Y$, $g : Y \longrightarrow Z$, and $h : Z \longrightarrow V$, we have $(h \circ g) \circ f = h \circ (g \circ f)$.
- For every object X, there is an *identity* which is a morphism $id_X : X \longrightarrow X$ satisfying $f \circ id_X = f$ for all $f : X \longrightarrow Y$ and $id_X \circ g = g$ for all $g : Y \longrightarrow X$.

If C is a category, we use the notation C_0 or $|C|$ for the collection of objects, and C_1 for the collection of morphisms. A *discrete category* is a category in which the only morphisms are the identities on the objects.

The terms "arrow" and "map" may be used instead of "morphism".

Some morphisms and objects may have special properties.

Definition 2.2 A morphism $f : X \longrightarrow Y$ is an *isomorphism* if there exists a morphism $g : Y \longrightarrow X$ such that $(g \circ f) = id_X$ and $(f \circ g) = id_Y$.

Definition 2.3 The objects X and Y are *isomorphic* if there exists an isomorphism between them.

Definition 2.4 An object X is an *initial object* if there exists exactly one morphism $X \longrightarrow Y$ for all objects Y.

Furthermore, we say that a morphism $f : X \longrightarrow X$ with the same domain and codomain is an *endomorphism*.

Special categories are later used in the definition of institutions. One of them is the category of sets.

Definition 2.5 *Set* is the category in which the collection of objects is the class of all sets and the collection of morphisms is the class of all functions between sets.

Categories may be constructed from other categories. We only use a fairly simple construction of the *opposite category*.

Definition 2.6 Let C be a category. The category C^{op} has the same objects as C. The morphisms of C^{op} are the morphisms of C with their directions reversed: if $f : X \longrightarrow Y$ is in C, then $\overline{f} : Y \longrightarrow X$ is in C^{op}.

2.1.2 Functors

As we have morphisms between objects within a category, we may also have morphisms between categories.

Definition 2.7 Let C and D be categories. A *functor* $F : C \longrightarrow D$ is a pair of functions $F_0 : C_0 \longrightarrow D_0$ and $F_1 : C_1 \longrightarrow D_1$ such that

- $F_1(f) : F_0(X) \longrightarrow F_0(Y)$ for each $f : X \longrightarrow Y$ in C_1
- $F_1(g \circ f) = F_1(g) \circ F_1(f)$ for each $f : X \longrightarrow Y$ and $g : Y \longrightarrow Z$ in C_1
- $F_1(id_X) = id_{F_0(X)}$ for each X in C_0.

Consequently, a functor $F : C \longrightarrow D$ maps objects and morphisms of C to objects and morphisms of D such that their structuring is preserved.

We often omit the subscripts and write $F(X)$ and $F(f)$ instead of $F_0(X)$ and $F_1(f)$, respectively.

Functors are composed by function composition of the components.

Definition 2.8 Let $F : A \longrightarrow B$ and $G : B \longrightarrow C$ be functors. The composition $G \circ F$ is a functor $A \longrightarrow C$ with $(G \circ F)_0 = G_0 \circ F_0$ and $(G \circ F)_1 = G_1 \circ F_1$.

Categories and functors form a category.

Definition 2.9 *Cat* is the category in which the objects are categories and the morphisms are functors.

As remarked in [45], *Cat* is strictly speaking not a category but a so-called quasicategory.

2.1.3 Natural Transformations

We often need to consider a collection of similar entities.

Definition 2.10 A *family* is a collection (set) of enities *indexed* by another collection (set).

We write $U = (U_i)_{i \in S}$ for a family U indexed by S. Moreover, an *S-sorted set* is a family of sets indexed by S.

When U is an S-sorted set, $x \in U$ means that $x \in U_s$ for some $s \in S$, and $x \notin U$ means that $x \notin U_s$ for all $s \in S$.

Definition 2.11 Let C and D be categories, and let $F : C \longrightarrow D$ and $G : C \longrightarrow D$ be functors. A *natural transformation* $\mu : F \Longrightarrow G$ is a family of morphisms $(\mu_X : F(X) \longrightarrow G(X))_{X \in C_0}$ in D, indexed by the collection of objects in C, such that the diagram

$$\begin{tikzcd} F(X) \arrow[r, "\mu_X"] \arrow[d, "F(f)"'] & G(X) \arrow[d, "G(f)"] \\ F(Y) \arrow[r, "\mu_Y"] & G(Y) \end{tikzcd}$$

commutes for all morphisms $f : X \longrightarrow Y$ in C, i.e., such that $(\mu_Y \circ F(f)) = (G(f) \circ \mu_X)$.

A morphism μ_X is called a *component* of $\mu = (\mu_X)_{X \in C_0}$.

2.1.4 Dependent Types

For *dependent types*, we use the notation

$$\Pi\, X \in S \bullet t(X)$$

where X is an element of the set S, and $t(X)$ denotes a (composite) type that depends on X. An element of the dependent type is essentially a function which takes an argument X and returns an element in $t(X)$, i.e., the type of the returned element depends on X.

2.2 Institutions

The concept of an institution formalizes the informal notion of a logical system. An institution consists of signatures, sentences, models, and a satisfaction relation such that certain conditions are met.

A signature may be thought of as a vocabulary for writing sentences; it typically contains types (sorts) of data and operators/constants that are available for forming sentences. A model over a signature is an interpretation of the entities in the signature – each identifier in the signature is given a meaning in a semantic domain. There is a satisfaction relation between models and sentences. Intuitively, a model satisfies a sentence if and only if the sentence is true when evaluated with the interpretation given by the model.

The concept of institutions is formally defined as follows:

Definition 2.12 An *institution* is a quadruple $\langle Sign, Sen, Mod, \models \rangle$ where

- $Sign$ is a category of signatures
- $Sen : Sign \longrightarrow Set$ is a functor that maps each signature to the set of sentences over the signature
- $Mod : Sign \longrightarrow Cat^{op}$ is a functor that maps each signature to the category of models over the signature
- $\models = (\models_\Sigma)_{\Sigma \in |Sign|}$ where $\models_\Sigma \subseteq |Mod(\Sigma)| \times Sen(\Sigma)$ is a satisfaction relation for each $\Sigma \in |Sign|$

so that

$$m' \models_{\Sigma'} Sen(\sigma)(e) \text{ iff } Mod(\sigma)(m') \models_\Sigma e$$

for each $m' \in |Mod(\Sigma')|$, $\sigma : \Sigma \longrightarrow \Sigma'$ in $Sign_1$, and $e \in Sen(\Sigma)$. This requirement is known as *the satisfaction condition*.

The notation $m \models_\Sigma e$ is used for $(m, e) \in \models_\Sigma$, and we say that the model m *satisfies* the sentence e. Moreover, the notation $\sigma(e)$ is often used for $Sen(\sigma)(e)$, and the notation $m|_\sigma$ is often used for $Mod(\sigma)(m)$. Finally, we say that $m|_\sigma$ is the *σ-reduct* of m.

Sentences over a signature Σ, i.e., sentences in $Sen(\Sigma)$, are called *Σ-sentences*. Similarly, models over a signature Σ, i.e., models in $|Mod(\Sigma)|$, are called *Σ-models*.

The satisfaction relation is lifted to relations between models and sentences. If m is a Σ-model and E is a set of Σ-sentences, then $m \models_\Sigma E$ if and only if $m \models_\Sigma e$ for all $e \in E$. And if M is a set of Σ-models and e is a Σ-sentence, then $M \models_\Sigma e$ if and only if $m \models_\Sigma e$ for all $m \in M$.

The concept of institutions (and the various kinds of mappings between institutions) provides a formal framework for comparing and relating logical systems. Mappings between institutions are considered in Section 2.7.

2.2.1 Related Concepts

The concept of institutions is one way of formalizing and describing logical systems; other concepts similar to institutions occur in the literature.

The concept of institutions takes a model-oriented stand. A central part of an institution is the satisfaction relation between models and sentences, and the definition of an institution does not explicitly include the notion of a *proof system* or *entailment relation* which describes when a sentence is a consequence of other sentences. The concept of a *general logic* [42], introduced by Meseguer, encompasses both an *entailment relation* $\Gamma \vdash \varphi$ between a set of sentences Γ and a sentence φ, and an institution.

While the concept of general logics adds an entailment relation to an institution, so to speak, the concept of *pre-institutions* [57], introduced by Salibra and Scollo, removes the satisfaction condition. A pre-institution is a weak version of an institution – it is defined as an institution for which the satisfaction condition is not required to hold.

Finally, there are other definitions of institutions than that of Definition 2.12. In [25], various equivalent definitions of institutions are given; one of which defines an institution as a functor from a category of signatures to a special category of *twisted relations* that contains information about sentences, models, and the satisfaction relation. In this definition, $Mod(\Sigma)$ denotes a set of models instead of a category. In other variations of definitions, $Sen(\Sigma)$ may denote a category.

2.3 Semantic Closure

Using the notation of [25], we let $*$ denote two functions:

$$* : \mathcal{P}(Mod(\Sigma)) \longrightarrow \mathcal{P}(Sen(\Sigma))$$

$$* : \mathcal{P}(Sen(\Sigma)) \longrightarrow \mathcal{P}(Mod(\Sigma))$$

where $Mod(\Sigma)$ denotes the collection of all Σ-models, $Sen(\Sigma)$ denotes the collection of all sentences over Σ (i.e., all Σ-sentences), and $\mathcal{P}(\_)$ is the power set operator.

Given a collection of Σ-models $M \subseteq Mod(\Sigma)$, the meaning of M^* is the collection of sentences that are satisfied by all the Σ-models in M:

$$M^* = \{e \in Sen(\Sigma) \mid \forall m \in M : m \models e\} \qquad \text{for } M \subseteq Mod(\Sigma).$$

Similarly, given a collection of Σ-sentences $E \subseteq Sen(\Sigma)$, the meaning of E^* is the collection of models that satisfy all the Σ-sentences in E:

$$E^* = \{m \in Mod(\Sigma) \mid \forall e \in E : m \models e\} \qquad \text{for } E \subseteq Sen(\Sigma).$$

Moreover, we abbreviate $**$ by $\bullet$ so that the *semantic closure* of a set E of Σ-sentences is written $E^\bullet$. Likewise, we may take the semantic closure of a set of models.

2.4 Specifications

When using a specification language, a specification is written in a concrete syntax. Although we shall use the term *specification* for a piece of text, we also understand a specification as the abstract syntax of this.

There are two kinds of specifications: *Flat* specifications and *structured* specifications.

Definition 2.13 A *flat specification* is a pair $\langle \Sigma, E \rangle$ where Σ is a signature and E is a finite set of sentences over Σ.

A flat specification is also known as a *basic specification*, and given a specification $SP = \langle \Sigma, E \rangle$, we may use the notation $Sig(SP) = \Sigma$ and $Ax(SP) = E$.

A structured specification is a specification constructed from other specifications by means of specification-building operators. Different specification-building operators are found in the literature (e.g., [32, 8]) and their definitions depend on the way a semantics is given to specifications. In this thesis, however, we concentrate on basic specifications.

The notion of a *presentation* is identical to our understanding of a flat specification.

Definition 2.14 A *presentation* is a pair $\langle \Sigma, E \rangle$ where Σ is a signature and E is a finite set of sentences over Σ.

The notion of a *theory* is similar to that of a presentation.

Definition 2.15 A *theory* is a pair $\langle \Sigma, E \rangle$ where Σ is a signature and E is a set of sentences over Σ, closed under semantic entailment (i.e., $E = E^\bullet$).

A theory may contain an infinite number of sentences, and a presentation may be seen as a finite representation of a theory.

Presentations and theories are used in the discussion of semantics of specifications.

2.5 Semantics of Specifications

We may assign semantics to specifications at three different levels:

- The presentation level: the semantics of a specification $\langle \Sigma, E \rangle$ is the presentation $\langle \Sigma, E \rangle$.
- The theory level: the semantics of a specification $\langle \Sigma, E \rangle$ is the theory $\langle \Sigma, E^\bullet \rangle$.
- The model level: the semantics of a specification is a collection of models that satisfy the axioms of the specification.

In the following, we concentrate on the model level and consider three different ways of defining the collection of models: *loose semantics*, *looser semantics*, and *initial semantics*.

2.5.1 Loose Semantics

Loose semantics of specifications is described in [18, 41].

Definition 2.16 The *loose semantics* of a flat specification $SP = \langle \Sigma, E \rangle$ is the class of all models that satisfy the set of sentences E:

$$\mathcal{M}_{loose}(SP) = \{m \in |Mod(\Sigma)| \mid m \models_\Sigma E\}.$$

In loose semantics, it is tacitly assumed that we only consider models over a given signature (the signature of the specification).

2.5.2 Looser Semantics

Let $Models$ be the class of *all* models:

$$Models = \bigcup_{\Sigma \in |Sign|} |Mod(\Sigma)|.$$

Definition 2.17 The *looser semantics* of a flat specification $SP = \langle \Sigma, E \rangle$ is:

$$\mathcal{M}_{looser}(SP) = \{m \in Models \mid \exists \Theta \in |Sign| : m \in |Mod(\Theta)| \wedge \Sigma \subseteq \Theta \wedge m \models_\Theta \sigma(E)\}$$

where $\sigma : \Sigma \hookrightarrow \Theta$ is a signature morphism that *includes* Σ in Θ. Such a signature morphism exists when Θ is an extension of Σ.

The looser semantics of a specification SP is the class of models that interpret *at least* the entities in the signature of SP and satisfy the sentences of SP.

Exactly what it means that a signature morphism $\sigma : \Sigma \hookrightarrow \Theta$ includes Σ in Θ depends on the definition of the institution and $Sign$.

2.5.3 Initial Semantics

It may be convenient to think of a specification as a description of a single model, or a class of isomorphic models.

Definition 2.18 The *initial semantics* of a specification $SP = \langle \Sigma, E \rangle$ is the class of all models in $\mathcal{M}_{loose}(SP)$ that are isomorphic to a model which is initial (i.e., has a unique morphism to all models) in $\mathcal{M}_{loose}(SP)$.

If $Mod(\Sigma)$ is a discrete category, it does not make sense to talk about initial semantics. Hence initial semantics cannot be defined for specifications built over arbitrary institutions as it is the case for loose and looser semantics.

2.5.4 Differences of Semantics

The difference between loose and looser semantics of a specification is especially noticable when the *empty specification* is considered. The signature of the empty specification is empty, i.e., no identifiers of entities are present. Moreover, the set of axioms is empty.

The loose semantics of the empty specification is a singleton set containing the *empty model*. The looser semantics of the empty specification, on the other hand, is the class of all models over all signatures!

2.6 Semantic Consequence

Let $SP = \langle \Sigma, E \rangle$ be a flat specification. We say that a sentence $\varphi \in Sen(\Sigma)$ is a *consequence* of the specification SP and write $SP \models \varphi$ if the sentence φ is satisfied by all the models of SP. The exact definition depends on the semantics assigned to the specification.

2.6.1 Loose Semantics

Definition 2.19 We say that φ is a consequence of $SP = \langle \Sigma, E \rangle$ *with respect to loose semantics* if

$$\forall m \in |Mod(\Sigma)|.[(\forall e \in E.m \models_\Sigma e) \Rightarrow m \models_\Sigma \varphi].$$

This is due to the definition of loose semantics and the usual way of lifting the satisfaction relation to sets:

$$\begin{aligned} & \mathcal{M}_{loose}(SP) \models_{Sig(SP)} \varphi \\ \text{iff } & \mathcal{M}_{loose}(\langle \Sigma, E \rangle) \models_\Sigma \varphi \\ \text{iff } & \{n \in |Mod(\Sigma)| \mid n \models_\Sigma E\} \models_\Sigma \varphi \\ \text{iff } & \{n \in |Mod(\Sigma)| \mid \forall e \in E.n \models_\Sigma e\} \models_\Sigma \varphi \\ \text{iff } & \forall m \in \{n \in |Mod(\Sigma)| \mid \forall e \in E.n \models_\Sigma e\}.[m \models_\Sigma \varphi] \\ \text{iff } & \forall m \in |Mod(\Sigma)|.[(\forall e \in E.m \models_\Sigma e) \Rightarrow m \models_\Sigma \varphi]. \end{aligned}$$

2.6.2 Looser Semantics

The essence of a precise definition of consequence of $SP = \langle \Sigma, E \rangle$ with respect to looser semantics is that

$$m \models_\Theta \sigma(E) \Rightarrow m \models_\Theta \sigma(\varphi)$$

for all signatures Θ, inclusions $\sigma : \Sigma \hookrightarrow \Theta$, and Θ-models m.

Definition 2.20 We say that φ is a consequence of $SP = \langle \Sigma, E\rangle$ *with respect to looser semantics* if

$$\forall \Theta \in |Sign|.[\forall \sigma : \Sigma \hookrightarrow \Theta.[\forall m \in |Mod(\Theta)|.[(\forall e \in E.m \models_\Theta \sigma(e)) \Rightarrow m \models_\Theta \sigma(\varphi)]]].$$

2.6.3 An Equivalence Result

The following theorem states that semantic consequence with respect to loose semantics is equivalent to semantic consequence with respect to looser semantics.

Theorem 2.21 *For all $SP = \langle \Sigma, E\rangle$ and $\varphi \in Sen(Sig(SP))$:*

$$\mathcal{M}_{loose}(SP) \models_{Sig(SP)} \varphi \quad \textit{iff} \quad \mathcal{M}_{looser}(SP) \models_{Sig(SP)} \varphi.$$

Proof. Assume that $\varphi \in Sen(\Sigma)$ is a consequence of the specification $\langle \Sigma, E\rangle$ with respect to loose semantics:

$$\forall m \in |Mod(\Sigma)|.[(\forall e \in E.m \models_\Sigma e) \Rightarrow m \models_\Sigma \varphi].$$

Then we shall show that φ is a consequence of $\langle \Sigma, E\rangle$ with respect to looser semantics:

$$\forall \Theta \in |Sign|.[\forall \sigma : \Sigma \hookrightarrow \Theta.[\forall m \in |Mod(\Theta)|.[(\forall e \in E.m \models_\Theta \sigma(e)) \Rightarrow m \models_\Theta \sigma(\varphi)]]].$$

Due to the satisfaction condition of institutions, this is equivalent to

$$\forall \Theta \in |Sign|.[\forall \sigma : \Sigma \hookrightarrow \Theta.[\forall m \in |Mod(\Theta)|.[(\forall e \in E.m|_\sigma \models_\Sigma e) \Rightarrow m|_\sigma \models_\Sigma \varphi]]].$$

Since all models $m|_\sigma$ are Σ-models, the assumption of $\langle \Sigma, E\rangle \models \varphi$ with respect to loose semantics implies that

$$\forall m \in |Mod(\Theta)|.(\forall e \in E.m|_\sigma \models_\Sigma e) \Rightarrow m|_\sigma \models_\Sigma \varphi$$

is true. Consequently, $\langle \Sigma, E\rangle \models \varphi$ with respect to loose semantics implies that $\langle \Sigma, E\rangle \models \varphi$ with respect to looser semantics.

Assume that $\varphi \in Sen(\Sigma)$ is a consequence of a specification $\langle \Sigma, E\rangle$ with respect to looser semantics:

$$\forall \Theta \in |Sign|.[\forall \sigma : \Sigma \hookrightarrow \Theta.[\forall m \in |Mod(\Theta)|.[(\forall e \in E.m \models_\Theta \sigma(e)) \Rightarrow m \models_\Theta \sigma(\varphi)]]].$$

For $\Theta = \Sigma$ and $\sigma = id_\Sigma : \Sigma \hookrightarrow \Sigma$ (the identities in the category $Sign$ are inclusions):

$$\forall m \in |Mod(\Sigma)|.[(\forall e \in E.m \models_\Sigma id_\Sigma(e)) \Rightarrow m \models_\Sigma id_\Sigma(\varphi)].$$

Since $id_\Sigma(\psi) = \psi$ for all $\psi \in Sen(\Sigma)$, this is the same as:

$$\forall m \in |Mod(\Sigma)|.[(\forall e \in E.m \models_\Sigma e) \Rightarrow m \models_\Sigma \varphi].$$

Consequently, $\langle \Sigma, E\rangle \models \varphi$ with respect to looser semantics implies that $\langle \Sigma, E\rangle \models \varphi$ with respect to loose semantics.

We can conclude that loose and looser semantics are equivalent with respect to semantic consequence. $\square$

The result means that the reusing theorem to be presented in Section 2.8.2 also may be used when specifications are given looser semantics.

2.7 Mappings between Institutions

Having defined institutions, we may define mappings between them. Many different kinds of mappings between institutions have been suggested, and [24] tries to bring order into this chaos.

We present institution morphisms and institution comorphisms[2], and introduce a new concept of light institution comorphisms that are weak versions of institution comorphisms.

The introduction of light institution comorphisms calls for the inevitable question: why introduce another kind of mapping? The answer is that when working with a mapping from an institution of (a subset of) RSL to an institution of HOL, it turned out that the requirements to an institution comorphism made matters unnecessarily complicated. Focus was therefore put on a kind of mapping that only contains what is needed for our purpose.

2.7.1 Institution Morphisms

Following the terminology of [24], the concept of an institution morphism is defined.

Definition 2.22 Let $I = \langle Sign, Sen, Mod, \models\rangle$ and $I' = \langle Sign', Sen', Mod', \models'\rangle$ be institutions. An *institution morphism* $\phi : I \longrightarrow I'$ is a triple $\langle \phi, \alpha, \beta\rangle$ where

- $\phi : Sign \longrightarrow Sign'$ is a functor,
- $\alpha : (Sen' \circ \phi) \Longrightarrow Sen$ is a natural transformation,
- $\beta : Mod \Longrightarrow (Mod' \circ \phi)$ is a natural transformation

so that

$$m \models_\Sigma \alpha_\Sigma(e') \text{ iff } \beta_\Sigma(m) \models'_{\phi(\Sigma)} e'$$

for any signature $\Sigma \in |Sign|$, model $m \in Mod(\Sigma)$, and sentence $e' \in Sen'(\phi(\Sigma))$.

[2]Institution comorphisms were formerly known as institution representations [59, 45] or maps of institutions [42] – the new name is due to [24]. However, the definition of an institution comorphism in [24] is slightly different from that of a simple institution representation in [45]. In our definition of a light institution comorphism, we lean towards [45].

In some formulations ([25, 59]), β may be given as $\beta : Mod \Longrightarrow (Mod' \circ \phi^{op})$ where $\phi^{op} : Sign^{op} \longrightarrow Sign'^{op}$ is the same as ϕ but on the opposite categories. Using ϕ^{op} is consistent with a definition of an institution where Mod is a functor from $Sign^{op}$ to Cat. Since we use $Mod : Sign \longrightarrow Cat^{op}$, we shall use $\beta : Mod \Longrightarrow (Mod' \circ \phi)$ to make the arrows in the diagrams fit.

When the question comes to using institution morphisms for translating specifications and proof obligations from one logical system to another logical system and reusing a theorem prover, it is noticed that signatures are translated from I to I' whereas sentences are translated from I' to I.

2.7.2 Institution Comorphisms

In contrast to an institution morphism, an institution comorphism from an institution I to an institution I' maps both signatures and sentences from I to I'.

Before we turn to the definition of an institution comorphism, recall presentations in Definition 2.14 and consider the following definition of presentation morphisms.

Definition 2.23 Given presentations $P = \langle \Sigma, E \rangle$ and $P' = \langle \Sigma', E' \rangle$, a *presentation morphism* $\sigma : P \longrightarrow P'$ is a signature morphism $\sigma : \Sigma \longrightarrow \Sigma'$ such that $Sen(\sigma)(e) \in E'$ for all $e \in E$.

The presentations and presentation morphisms over an institution form a category.

Definition 2.24 Let $I = \langle Sign, Sen, Mod, \models \rangle$ be an institution. $Pres$ is the category of presentations over I. The objects are presentations over I, and the morphisms are presentation morphisms.

Having a presentation $\langle \Sigma, E \rangle$, we may restrict our attention to the Σ-models that satisfy the sentences in E. Consequently, $Mod(\langle \Sigma, E \rangle)$ denotes the category of Σ-models that satisfy the sentences in E. This is a subcategory of $Mod(\Sigma)$, i.e., the objects and morphisms of $Mod(\langle \Sigma, E \rangle)$ are also in $Mod(\Sigma)$.

The concept of an institution comorphism can now be defined.

Definition 2.25 Let $I = \langle Sign, Sen, Mod, \models \rangle$ and $I' = \langle Sign', Sen', Mod', \models' \rangle$ be institutions, and let $Pres'$ be the category of presentations over I'. An *institution comorphism* $\varrho : I \longrightarrow I'$ is a triple $\langle \varrho, \alpha, \beta \rangle$ where

- $\varrho : Sign \longrightarrow Pres'$ is a functor that maps signatures into presentations,
- $\alpha : Sen \Longrightarrow (Sen' \circ \varrho)$ is a natural transformation,
- $\beta : (Mod' \circ \varrho) \Longrightarrow Mod$ is a natural transformation

so that

$$m' \models'_{Sig(\varrho(\Sigma))} \alpha_\Sigma(e) \text{ iff } \beta_\Sigma(m') \models_\Sigma e$$

for any signature $\Sigma \in |Sign|$, model $m' \in |Mod'(\varrho(\Sigma))|$, and sentence $e \in Sen(\Sigma)$.

Since institution comorphisms are also known as *institution representations*, the requirement is known as the *representation condition*. It can be illustrated by the following diagram:

$$\begin{tikzcd} m' \arrow[d, "\beta_\Sigma"'] & \models_{Sig(\varrho(\Sigma))} & \alpha_\Sigma(e) \\ \beta_\Sigma(m') & \models_\Sigma & e \arrow[u, "\alpha_\Sigma"'] \end{tikzcd}$$

The naturality of $\alpha : Sen \Longrightarrow (Sen' \circ \varrho)$ is illustrated in the below diagram in which $\sigma : \Sigma_1 \longrightarrow \Sigma_2$ is a signature morphism in I.

$$\begin{tikzcd} Sen(\Sigma_1) \arrow[r, "\alpha_{\Sigma_1}"] \arrow[d, "Sen(\sigma)"'] & Sen'(Sig(\varrho(\Sigma_1))) \arrow[d, "Sen'(Sig(\varrho(\sigma)))"] \\ Sen(\Sigma_2) \arrow[r, "\alpha_{\Sigma_2}"'] & Sen'(Sig(\varrho(\Sigma_2))) \end{tikzcd}$$

The natural transformation α is a family of morphisms in Set, i.e., α is a family of functions. The family is indexed by the objects in $Sign$, i.e., $\alpha = (\alpha_\Sigma)_{\Sigma \in |Sign|}$.

The function α_Σ translates sentences over the signature Σ in I to sentences over the signature $Sig(\varrho(\Sigma))$ in I'. If E is a set of sentences over Σ, then $\alpha_\Sigma(E)$ is the set $\{\alpha_\Sigma(e) \mid e \in E\}$ of translated sentences. In other words, the function α_Σ is lifted to work on sets of sentences in the usual manner.

The naturality of $\beta : (Mod' \circ \varrho^{op}) \Longrightarrow Mod$ is illustrated in the below diagram in which $\sigma : \Sigma_1 \longrightarrow \Sigma_2$ is a signature morphism in I.

$$\begin{tikzcd} Mod'(\varrho(\Sigma_2)) \arrow[r, "\beta_{\Sigma_2}"] \arrow[d, "Mod'(\varrho(\sigma))"'] & Mod(\Sigma_2) \arrow[d, "Mod(\sigma)"] \\ Mod'(\varrho(\Sigma_1)) \arrow[r, "\beta_{\Sigma_1}"'] & Mod(\Sigma_1) \end{tikzcd}$$

The natural transformation β is a family of morphisms in Cat, i.e., β is a family of functors. The family is indexed by the objects in $Sign$, i.e., $\beta = (\beta_\Sigma)_{\Sigma \in |Sign|}$.

The functor β_Σ is from the category of models of $\varrho(\Sigma)$ to the category of models of Σ, i.e., from models in I' to models in I and from morphisms between models in I' to morphisms between models in I.

2.7.3 Light Institution Comorphisms

We now present a modified version of an institution comorphism.

Definition 2.26 Let $I = \langle Sign, Sen, Mod, \models\rangle$ and $I' = \langle Sign', Sen', Mod', \models'\rangle$ be institutions, and let $Pres'$ be the category of presentations over I'. A *light institution comorphism* $\varrho : I \longrightarrow I'$ is a triple $\langle \varrho, \alpha, \beta\rangle$ where

- $\varrho : |Sign| \longrightarrow |Pres'|$ is a function that maps signatures into presentations,
- $\alpha = (\alpha_\Sigma : Sen(\Sigma) \longrightarrow Sen'(Sig(\varrho(\Sigma))))_{\Sigma\in|Sign|}$ is a family of functions,
- $\beta = (\beta_\Sigma : |Mod'(\varrho(\Sigma))| \longrightarrow |Mod(\Sigma)|)_{\Sigma\in|Sign|}$ is a family of partial functions

so that

$$m' \models'_{Sig(\varrho(\Sigma))} \alpha_\Sigma(e) \quad \text{iff} \quad \beta_\Sigma(m') \models_\Sigma e$$

for any signature $\Sigma \in |Sign|$, sentence $e \in Sen(\Sigma)$, and model $m' \in |Mod'(\varrho(\Sigma))|$ for which $\beta_\Sigma(m')$ is defined.

The light institution comorphism is very similar to the institution comorphism defined in Definition 2.25. However, $\varrho : |Sign| \longrightarrow |Pres'|$ is a function and not a functor $Sign \longrightarrow Pres'$. Consequently, ϱ is not concerned with mapping signature morphisms. Furthermore, the family of functions $\alpha = (\alpha_\Sigma)_{\Sigma\in|Sign|}$ is not required to be natural, i.e., signature morphisms are not considered here either. Likewise, $\beta = (\beta_\Sigma : |Mod'(\varrho(\Sigma))| \longrightarrow |Mod(\Sigma)|)_{\Sigma\in|Sign|}$ is a family of partial functions and not a family of functors. Since β_Σ may be partial, the representation condition is only required to hold when $\beta_\Sigma(m')$ is defined.

The partiality of β_Σ reflects that the institution I' may have a larger collection of models than the institution I. However, $Sign$ may contain more signatures and signature morphisms than found in $Sign'$.

2.7.4 Related Concepts

The concepts of institution morphisms and comorphisms have previously been subjects to modifications and changes. The definition of an institution comorphism has, e.g., previously been weakened by relaxing the requirements of naturality – the result was *semi-natural* institution comorphisms. As Goguen and Rosu put it in [24]: “The point to note here is that β need not be natural; this condition is not satisfied in some examples, and is not needed to ensure some significant properties.”

In [3], *simulations* likewise have partial, surjective mappings of models.

2.8 Reusing Theorem Provers

We now turn to the issue of reusing theorem provers, i.e., having a specification and a proof obligation in one logical system and then use a theorem prover of a different logical system to prove the proof obligation.

2.8.1 Proof Systems

To consider proof systems, we must consider the relations between sentences.

Definition 2.27 An *entailment relation* of an institution $\langle Sign, Sen, Mod, \models\rangle$ is a family of relations $\vdash = (\vdash_\Sigma)_{\Sigma \in |Sign|}$ such that

$$\vdash_\Sigma \subseteq \mathcal{P}(Sen(\Sigma)) \times Sen(\Sigma)$$

for $\Sigma \in |Sign|$.

If $(\Gamma, e) \in \vdash_\Sigma$ for a set Γ of Σ-sentences and a Σ-sentence e, we write $\Gamma \vdash_\Sigma e$ and say that Γ *entails* e or that e is *derivable* from Γ.

An entailment relation is not the same as a proof calculus. It is a relation between sets of sentences and sentences, and such a relation may be defined in different ways. For instance, an entailment relation may be defined on the basis of a satisfaction relation: for signatures Σ, sets of Σ-sentences E, and Σ-sentences e, we write $E \vdash_\Sigma e$ if $\langle \Sigma, E\rangle \models e$.

An entailment relation may also be understood as an abstract representation of a proof calculus. The entailment relation is then defined such that $E \vdash_\Sigma e$ if e can be proved from the set E of hypotheses using the axioms and inference rules of the proof calculus.

If $SP = \langle \Sigma, E\rangle$, the notation $SP \vdash e$ is used for $E \vdash_\Sigma e$, and we say that e can be proved in the context of the specification SP.

Definition 2.28 The entailment relation $\vdash = (\vdash_\Sigma)_{\Sigma \in |Sign|}$ is *sound* with respect to the satisfaction relation $\models$ if and only if

$$\Gamma \vdash_\Sigma e \text{ implies } \Gamma^* \models_\Sigma e$$

for all signatures $\Sigma \in |Sign|$, sets of sentences $\Gamma \subseteq \mathcal{P}(Sen(\Sigma))$, and sentences $e \in Sen(\Sigma)$.

2.8.2 The Reusing Theorem

A light institution comorphism may be used to reuse a theorem prover for flat specifications by translating the specifications (and proof obligations) along the light institution comorphism as follows:

Definition 2.29 Let $I = \langle Sign, Sen, Mod, \models\rangle$ and $I' = \langle Sign', Sen', Mod', \models'\rangle$ be institutions, let $\langle \varrho, \alpha, \beta\rangle$ be a light institution comorphism from I to I', and let $SP = \langle \Sigma, E\rangle$ be a flat specification over the institution I.

When $\varrho(\Sigma) = \langle \Sigma', E'\rangle$, the result of translating SP along ϱ is a flat specification

$$\langle \Sigma', \alpha_\Sigma(E) \cup E'\rangle$$

and the result of translating a proof obligation (Σ-sentence) φ along ϱ is a Σ'-sentence $\alpha_\Sigma(\varphi)$.

Theorem 2.30 *Let $I = \langle Sign, Sen, Mod, \models\rangle$ and $I' = \langle Sign', Sen', Mod', \models'\rangle$ be institutions, let $\langle \varrho, \alpha, \beta\rangle$ be a light institution comorphism from I to I', and let $\vdash'$ be an entailment relation for I' that is sound with respect to $\models'$.*

If β_Σ is surjective for all $\Sigma \in |Sign|$, then

$$\varrho(\langle \Sigma, E\rangle) \vdash' \alpha_\Sigma(\varphi) \textit{ implies } \langle \Sigma, E\rangle \models \varphi \textit{ with respect to loose semantics}$$

for all signatures $\Sigma \in |Sign|$, sets $E \subseteq Sen(\Sigma)$, and sentences $\varphi \in Sen(\Sigma)$.

Proof. A signature $\Sigma \in |Sign|$ is mapped to a presentation $\varrho(\Sigma) = \langle \Sigma', E'\rangle$. Moreover, a specification $\langle \Sigma, E\rangle$ in I is translated to a specification $\varrho(\langle \Sigma, E\rangle) = \langle \Sigma', \alpha_\Sigma(E) \cup E'\rangle$ in I', and a proof obligation $\varphi \in Sen(\Sigma)$ in I is translated to the proof obligation $\alpha_\Sigma(\varphi) \in Sen'(\Sigma')$ in I'.

Assume that β_Σ is surjective for all $\Sigma \in |Sign|$, and assume that $\vdash'$ is sound with respect to $\models'$ and that $\varrho(\langle \Sigma, E\rangle) \vdash' \alpha_\Sigma(\varphi)$. The soundness of $\vdash'$ then gives us $(\alpha_\Sigma(E) \cup E')^* \models'_{\Sigma'} \alpha_\Sigma(\varphi)$, after which the representation condition gives us $\beta_\Sigma((\alpha_\Sigma(E) \cup E')^*) \models_\Sigma \varphi$. The question is now whether or not $\beta_\Sigma((\alpha_\Sigma(E) \cup E')^*)$ equals E^*.

We have

$$E^* = \{m \in |Mod(\Sigma)| \mid \forall e \in E : m \models_\Sigma e\}$$

and

$$\begin{aligned}
& (\alpha_\Sigma(E) \cup E')^* \\
= \;& \{m' \in |Mod'(Sig(\varrho(\Sigma)))| \mid \forall e' \in (\alpha_\Sigma(E) \cup E') : m' \models'_{\Sigma'} e'\} \\
= \;& \{m' \in |Mod'(\varrho(\Sigma))| \mid \forall e' \in \alpha_\Sigma(E) : m' \models'_{\Sigma'} e'\} \\
= \;& \{m' \in |Mod'(\varrho(\Sigma))| \mid \forall e \in E : m' \models'_{\Sigma'} \alpha_\Sigma(e)\}
\end{aligned}$$

since models in $|Mod'(\varrho(\Sigma))|$ trivially satisfies E'.

We then have:

$$\begin{aligned}
& \beta_\Sigma((\alpha_\Sigma(E) \cup E')^*) \\
= \;& \{m \in |Mod(\Sigma)| \mid \exists m' \in (\alpha_\Sigma(E) \cup E')^* : \beta_\Sigma(m') = m\} \\
= \;& \{m \in |Mod(\Sigma)| \mid \exists m' \in |Mod'(\varrho(\Sigma))| : \beta_\Sigma(m') = m \wedge [\forall e \in E : m' \models'_{\Sigma'} \alpha_\Sigma(e)]\} \\
& \text{since } \beta_\Sigma(m') \text{ is defined} \\
= \;& \{m \in |Mod(\Sigma)| \mid \exists m' \in |Mod'(\varrho(\Sigma))| : \beta_\Sigma(m') = m \wedge [\forall e \in E : \beta_\Sigma(m') \models_\Sigma e]\} \\
= \;& \{m \in |Mod(\Sigma)| \mid \exists m' \in |Mod'(\varrho(\Sigma))| : \beta_\Sigma(m') = m \wedge [\forall e \in E : m \models_\Sigma e]\}.
\end{aligned}$$

Suppose β_Σ is surjective, i.e.,

$$\forall m \in |Mod(\Sigma)| : \exists m' \in |Mod'(\varrho(\Sigma))| : \beta_\Sigma(m') = m.$$

Suppose $m \in E^*$. Then $\forall e \in E : m \models_\Sigma e$, which together with surjectivity of β_Σ implies $m \in \beta_\Sigma((\alpha_\Sigma(E) \cup E')^*)$. Consequently, $E^* \subseteq \beta_\Sigma((\alpha_\Sigma(E) \cup E')^*)$.

Suppose $m \in \beta_\Sigma((\alpha_\Sigma(E) \cup E')^*)$. Then $\forall e \in E : m \models_\Sigma e$. Consequently, $m \in E^*$ and $\beta_\Sigma((\alpha_\Sigma(E) \cup E')^*) \subseteq E^*$.

We can conclude that if β_Σ is surjective, then $E^* = \beta_\Sigma((\alpha_\Sigma(E) \cup E')^*)$ which means that we can prove $E^* \models_\Sigma \varphi$ by proving $(\alpha_\Sigma(E) \cup E') \vdash' \alpha_\Sigma(\varphi)$.

As a consequence, we can reuse a theorem prover for *flat specifications* if β_Σ is surjective. $\square$

For an institution comorphism, where the components of β are not only functions but functors, each β_Σ is required to be surjective on objects. If β_Σ is surjective on objects for any $\Sigma \in |Sign|$, we also say that the institution comorphism $\langle \varrho, \alpha, \beta \rangle$ is *faithful* [45], and that $\langle \varrho, \alpha, \beta \rangle$ has the *β-expansion property* [11].

For reusing theorem provers for structured specifications, additional requirements must be imposed on the institution comorphism [45, 11].

CHAPTER 3

The RAISE Specification Language

In this chapter, we give a brief overview of RSL. This provides an introduction to the syntax and the semantics, as well as the RAISE method with justifications (proofs) and tool support. The purpose is to give an impression of RSL before an institution for a subset of RSL is defined in Chapter 4 – a subset summarized in Appendix A.

RSL is a wide-spectrum specification language which includes and integrates different specification styles (algebraic, model-oriented, applicative, imperative, and concurrent specifications) in a common conceptual framework. RSL can be used for writing abstract specifications as well as concrete specifications close to implementations in programming languages, and it may therefore serve as a specification language in all phases of development.

For a thorough description of RSL, we refer to [54] which contains a tutorial and a reference description. A shorter introduction to RAISE is found in [20]. The RAISE Development Method is described in [55], and a collection of case studies is found in [61]. A denotational semantics of RSL is given in [43] and further described in [44, 10].

3.1 Specifications

A basic RSL specification consists of declarations of types, values, axioms, variables, and channels.

RSL has a series of built-in types and type constructors. In addition, user-declared types may be introduced as abstract sort types or as abbreviation types which are new names for other types (typically for model-oriented types built from the basic types and type constructors). Values denote constants and functions that are specified by axioms. Functions may access variables and communicate on channels, i.e., imperative aspects and process behaviour may be described. Moreover, language constructs known from programming languages as well as logical connectives, quantifiers, and nondeterminism are included in RSL.

Specifications may be built from other specifications by renaming declared entities, hiding declared entities, or adding more declarations. Moreover, specifications may be parameterized.

In the rest of this section, we consider types, values, variables, channels, and modules of RSL.

3.1.1 Types

Type expressions denote types which are sets in the RSL semantics.

Type Expressions

The built-in types are represented by type literals:

- **Bool** is the type of truth values
- **Int** is the type of integers
- **Nat** is the type of natural numbers, non-negative integers
- **Real** is the type of reals
- **Char** is the type of characters
- **Text** is the type of texts, i.e., finite lists of characters
- **Unit** is the singleton type.

The elements of each built-in type are represented by value literals.

Composite types may be constructed from the built-in types and user-declared types by type operators:

- $T_1 \overset{\sim}{\underset{m}{\rightarrow}} T_2$ is the type of nondeterministic maps that map elements of type T_1 to elements of type T_2.
- $T_1 \underset{m}{\rightarrow} T_2$ is the type of finite, deterministic maps that map elements of type T_1 to elements of type T_2.
- $T_1 \overset{\sim}{\rightarrow} T_2$ is the type of functions from type T_1 to type T_2. The functions may be partial (they may not terminate for all argument values), and they may yield nondeterministic results.

- $T_1 \to T_2$ is the type of total functions from type T_1 to type T_2. A total function from T_1 to T_2 yields a deterministic result for all arguments of type T_1.
- $T_1 \times T_2$ is the Cartesian product.
- T-**set** is the type of finite sets of elements of type T.
- T-**infset** is the type of finite and infinite sets of elements of type T.
- T^* is the type of finite lists of elements of type T.
- T^ω is the type of finite and infinite lists of elements of type T.

Type expressions may be disambiguated by parentheses.

When a type T is understood as a set, a *subtype* of T is a subset of that set. A subtype expression is usually of the form

{| x : T • P(x) |}

where P(x) is a Boolean expression that may contain x. The subtype expression represents the subtype of T that contains the elements x of type T for which $P(x)$ is true.

Type Declarations

Types may also be introduced by the user (specifier) in type declarations.

A *sort type* is an abstract type which is not further specified. A sort type is declared with a name:

type s

An *abbreviation type* is a name for a type. An abbreviation type is declared with a name and the type it is an abbreviation of:

type a = te

where *te* is a type expression.

A *variant definition* is a shorthand for declaring a sort type together with some values and axioms. Variant definitions come in many forms. As an example, the variant definition

type H = A | B | C | On(H,H)

declares a sort type H together with the values:

A : H,
B : H,
C : H,
On : H × H → H

The values are *constructors* that can generate the elements of the type H. Axioms stating that different constructors generate different elements and that the constructors can generate all the elements of H are also declared as a result of the variant definition.

Union definitions and *short record definitions* are shorthands for certain kinds of variant definitions.

3.1.2 Values

The constants and functions that are specified in a specification are declared as values.

Value Expressions

Values are represented by value expressions, and a value expression denotes an element of a type. Strictly speaking, a value expression denotes a function – this is explained in Section 3.2.

There are four basic expressions:

- **chaos** represents divergence, the non-terminating process
- **skip** represents the process that terminates without performing communication or accessing variables
- **stop** represents a deadlock
- **swap** is the completely unspecified expression.

There are value literals that represent the elements of the built-in types:

- The literals **true** and **false** denote the elements of the type **Bool**
- Numbers represent elements of the types **Int**, **Nat**, and **Real**
- Characters in single quotes represent elements of **Char**, e.g., ′a′ and ′z′
- Strings in double quotes represent elements of **Text**, e.g., ″The brown fox jumped over the fence″.
- The single element of the type **Unit** is represented by ().

Elements of composite types may be constructed in various ways, some of which are listed below.

- Maps may be explicitly written as $[x_1 \mapsto y_1, x_2 \mapsto y_1]$ expressing that x_1 is mapped to y_1 and x_2 is mapped to y_2.
- Functions may be written as λ-abstractions of the form λ x : T • f(x) where f(x) is a value expression in which x may occur.
- Products are written (x,y).

$\wedge$	**true**	**false**	**chaos**
true	**true**	**false**	**chaos**
false	**false**	**false**	**false**
chaos	**chaos**	**chaos**	**chaos**

$\vee$	**true**	**false**	**chaos**
true	**true**	**true**	**true**
false	**true**	**false**	**chaos**
chaos	**chaos**	**chaos**	**chaos**

$\Rightarrow$	**true**	**false**	**chaos**
true	**true**	**false**	**chaos**
false	**true**	**true**	**true**
chaos	**chaos**	**chaos**	**chaos**

$\sim$	**true**	**false**	**chaos**
	false	**true**	**chaos**

Figure 3.1: Truth tables of the Boolean connectives.

- Sets may be explicitly written as $\{a, b, c\}$ or as comprehended set expressions such as { 3 * n | n : **Nat** • n ≤ 10 } which reads "the set of values 3 * n where n is a natural number such that n is less than or equal to 10".
- Lists may be explicitly written as $\langle a, b, c \rangle$.

In addition to the expressions shown above, there are comprehended map expressions, comprehended list expressions, etc.

A wide variety of built-in operators are available for constructing value expressions.

For every type, there is an equality operator =, an inequality operator ≠, and an equivalence operator ≡ available. The equality operators evaluate expressions and compare the results, while the equivalence operator compares expressions taking side-effects into account and without evaluating the expressions. Side-effects are accessing variables and communication on channels.

For writing Boolean value expressions, the Boolean literals, quantifiers, and well-known operators are used. Three kinds of quantifiers are available: ∀ (universal quantification), ∃ (existence), and ∃! (unique existence). And four Boolean operators are available: ∧ (conjunction), ∨ (disjunction), ⇒ (implication), and ∼ (negation). The operators are introduced as *connectives*, and the following equivalences holds:

b1 ∧ b2 ≡ **if** b1 **then** b2 **else false end**
b1 ∨ b2 ≡ **if** b1 **then true else** b2 **end**
b1 ⇒ b2 ≡ **if** b1 **then** b2 **else true end**
∼b ≡ **if** b **then false else true end**

A Boolean expression may be divergent, and the truth tables in Figure 3.1 describe how the connectives behave when divergence occurs:

Although **chaos** is included in the truth tables, the logic in RSL is still said to be two-valued[1]: the type **Bool** only contains the two elements **true** and **false**.

[1]Sometimes, **chaos** is said to represent undefinedness. It is worth noticing that the logical connectives are defined differently from those of Kleene's three-valued logic [37]. Incorporating Kleene's logic and process algebra is explored in [7].

Value expressions may also be formed by applying functions or maps to arguments, or by using the built-in operators in infix expressions such as $8 + 8$. Well-known aritmetic operators (e.g., $+$, $-$, and $/$), as well as operators on sets and maps (e.g., $\cup$ and $\cap$), are available.

A value expression may also refer to a user-declared value or involve variables or channels.

Value Declarations

A value is declared with an identifier and a type:

value v : T

Values are specified by axioms, and there are various ways of writing a value declaration and an axiom together in a single declaration. Functions may be specified by general axioms, by post and pre-conditions (*implicit function definitions*), or by explicitly stating the result of applying the function (*explicit function definitions*).

Below, axiom declarations and explicit function declarations are considered.

Axiom Declarations

The axioms that specify values are essentially equivalence expressions. An axiom declaration has the general form:

axiom
  [axiom_name] value_expr_1 ≡ value_expr_2

where the value expressions value_expr_1 and value_expr_2 have the same type. An axiom may also be declared by:

axiom
  [axiom_name] value_expr

if the value expression value_expr is Boolean. This declaration is essentially a shorthand for:

axiom
  [axiom_name] value_expr ≡ **true**

Explicit Function Definitions

Functional values are often declared and defined in an *explicit function definition* which is a shorthand for a value declaration and an axiom declaration. The declaration

value
  sq : **Int** $\xrightarrow{\sim}$ **Int**
  sq(x) ≡ x ∗ x

is short for the two declarations

value
  $\mathrm{sq} : \mathbf{Int} \xrightarrow{\sim} \mathbf{Int}$
axiom
  $\forall\, \mathrm{x} : \mathbf{Int} \bullet \mathrm{sq(x)} \equiv \mathrm{x} * \mathrm{x}$

where the optional name of the axiom has been left out.

3.1.3 Variables

A variable is declared with an identifier and the type of the variable:

variable x : T

The variable x may then occur in value expressions.

Imperative expressions that correspond to statements in imperative programming languages are available, and if a function may access variables, it must be specified in the type of the function in an *access description*.

3.1.4 Channels

A channel is declared with an identifier and the type of the channel, i.e., the type of the values that can be communicated through the channel:

channel c : T

The channel c may then occur in value expressions.

The meaning of an output expression of the form c!v is to offer to output the value v on the channel, and the meaning of an input expression of the form c? is to offer to input from the channel c. If the expressions c!v and c? occur in a parallel composition, communication may occur such that the output expression c!v returns the unit value, and the input expression c? returns the value v input from c.

If a function may communicate on channels, it must be specified in the type of the function in an access description.

3.1.5 Modules

The basis for RSL modules is class expressions. A *class expression* contains a set of declarations and represents a collection of models that conform with the declarations, i.e., the class of all models that interpret the declared sorts and values in such a way that the axioms are satisfied.

A *module* is either a scheme or an object. A *scheme* is a named class expression, and as a consequence, a scheme represents a collection of models. An *object* is an instantiation of a scheme and represents a specific model. In other words, an object is a named model which can be any of the models in the collection of models represented by the scheme that the object instantiates.

A basic scheme is of the form

scheme A =
  class
    ...
  end

However, a scheme may also be built from another scheme by hiding entities, renaming entities, or extending the scheme with new declarations.

3.2 Semantics

In the above introduction to the components of RSL specifications, the semantics have already been described informally.

In the following, it is explained how semantics have been given to type expressions, value expressions, explicit function definitions, and specifications. These parts of the language are considered because they cover the subset of RSL considered in Chapter 4 and are relevant to the examples in Chapter 7.

3.2.1 Semantics of Type Expressions

A type is a set known as a *value domain*. There are value domains for the basic types and composite types. An example of a domain for a basic type is that of the Booleans: $Value_{Bool} = \{True, False\}$. Regarding composite types, notice that the value domains representing function types are the function spaces containing *all* functions.

The value domains form the basis for the *process domains* used for giving semantics to value expressions.

3.2.2 Semantics of Value Expressions

Applicative, imperative, and concurrent features have been integrated in the RSL semantics. There is only one syntactic category of value expressions that in other languages typically would be distinguished as expressions, statements, and process expressions. Consequently, any kind of value expression denotes a function from a store (containing the state of the variables) to a process! As an example, the value expression **true** denotes the function that takes a store to the process that immediately terminates without side-effects (i.e., without changing the store) and returns the Boolean value $True$.

A process may not terminate, in which case it is said to be divergent. A non-divergent process is a quadruple (R, I, O, A), where

- R is the set of possible results (a result is a new store reflecting side-effects and a returned value),
- I is a map describing how the process behaves after input of a value on a channel,
- O is a map describing how the process behaves after output of a value on a channel,
- A is a set describing the state of the process (possibilities for communication and termination).

A naive[2] description of the domain of processes is:

$$\begin{array}{lll} \mathit{Procdom} = \{(R, I, O, A) & | & R \in \mathcal{P}_f(\mathit{Store} \times \mathit{Value}), \\ & & I \in E_{in} \underset{m}{\rightarrow} (\mathit{Value} \rightarrow \mathit{Procdom}), \\ & & O \in E_{out} \underset{m}{\rightarrow} (\mathit{Value} \underset{m}{\rightarrow} \mathit{Procdom}), \\ & & A \in \mathcal{P}_f(\mathcal{P}_f(E_{in}) \times \mathcal{P}_f(E_{out}) \times \mathcal{P}_f(\{\surd\})) \\ & & \bullet \ \ldots \\ \} \cup \{\perp\} & & \end{array}$$

where $\mathcal{P}_f(\_)$ gives the set of finite subsets of a set, Store is the collection of all stores, and Value is the union of the value domains. The set R of possible results expresses nondeterminism caused by internal and external choices. E_{in} and E_{out} are sets of "events", i.e., identifiers of channels on which the process may communicate, and the tick $\surd$ denotes termination.

The components of processes are required to have certain properties and relations. These requirements, however, have been left out (the ellipsis). See [10] for details.

3.2.3 Axiomatic Interpretation

Explicit function definitions are interpreted axiomatically, not operationally. This should be kept in mind when looking at the treatment of non-termination. Consider the below definitions:

value
  f : **Int** $\xrightarrow{\sim}$ **Int**
  f(x) ≡ **if** x = 0 **then** 1 **else** x ∗ f(x−1) **end**

The explicit function definition of f is a shorthand for writing a value declaration and an axiom declaration:

value
  f : **Int** $\xrightarrow{\sim}$ **Int**
axiom
  ∀ x : **Int** • f(x) ≡ **if** x = 0 **then** 1 **else** x ∗ f(x−1) **end**

[2] In order to avoid undesired self references, the domains should be typed according to a hierarchy laid down by the static semantics. See [10].

This is satisfied by the function given by:

$$f(x) = \begin{cases} \textbf{chaos} & \text{if } x < 0 \\ 1 & \text{if } x = 0 \\ x! & \text{if } x > 0 \end{cases}$$

However, the function given by:

$$f(x) = \begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x = 0 \\ x! & \text{if } x > 0 \end{cases}$$

also satisfies the axiom. Consequently, f(-1) is **chaos** in some models, but certainly not in all models.

3.2.4 Semantics of Specifications

A basic RSL specification is a scheme which is a named class expression. As mentioned in Section 3.1.5, a class expression represents the collection (class) of all models that conforms with the declarations of the class expression.

A *model* of a class expression interprets the identifiers of values, types, variables, and channels introduced by the declarations in the class expression. But a model is also allowed to interpret more identifiers. To allow extensions as implementations, the models of a class expression include all possible extensions ([54], pp. 204-205).

In Chapter 2, the concept of looser semantics was introduced to give a description of this kind of semantics of specifications using terminology from the field of algebraic specification.

3.3 The RAISE Method

The RAISE Method is a software development method described in [55].

Stepwise refinement is an important paradigm in the RAISE method. In a series of development steps, an initial abstract specification is refined into a concrete specification suited for implementation. There is a *refinement relation* between specifications. A specification SP_r refines a specification SP if (i) SP_r defines at least the same types, values, variables, and channels as SP, and (ii) the axioms of SP are consequences of SP_r such that properties of SP are preserved by SP_r. The first requirement is known as *static refinement* or *static implementation*. Due to the definition of refinement, a specification which is the result of a series of refinements has the properties of the initial specification.

When checking that a specification SP_r refines a specification SP, it is checked that the types, values, variables, and channels of SP are declared in SP_r. Values must be declared with the same type, and abbreviation types must also abbreviate the same types. Sort types may be refined into abbreviation types. SP_r may, of course, declare more types and values than SP.

To check that the axioms of SP are consequences of SP_r, they should be proved (or informally *justified*) in the context of SP_r. In other words, the axioms of SP become *proof obligations* to be proved using the RSL proof rules and the axioms of SP_r.

Proof obligations may also occur for other reasons. Given a specification, so-called *confidence conditions* may be generated [19]. They should generally be true to ensure that the specification is consistent. For instance, subtyping gives rise to confidence conditions (or *type correctness conditions*) – values and arguments that are supposed to be in subtypes, should be in subtypes.

Initial specifications are often applicative since reasoning about applicative specifications is easier than reasoning about imperative specifications. Developing an imperative specification from an applicative specification is not a formal refinement. However, it can be done in a systematic way which is not likely to introduce errors.

3.4 Justifications

Proof obligations and confidence conditions may be *justified* using the RAISE proof system [55, 21].

3.4.1 Justifications

A *justification* in RAISE is a *rigorous* argument for the truth of a condition. As the term "rigorous" indicates, a justification need not be a formal proof – the user decides on the level of formality and detail.

When developing a justification for a condition, the user typically starts with the condition and tries to simplify it, rewrite it, or replace it with simpler conditions. This kind of reasoning is known as *backwards justification*, and the condition that the user tries to prove is called a *goal*.

3.4.2 Proof Rules

Proof rules are used for manipulating a goal or replacing a goal with other goals. *Inference rules* are used to replace a goal with a series of subgoals (new goals), and *equivalence rules* are used to rewrite a goal by substituting expressions.

There is a wide variety of built-in RSL proof rules, and rules are also derived from the context specification. Some of the rules have side-conditions, i.e., they can only be applied when certain conditions are met.

Proof rules are also discussed in Chapter 7.

3.5 Tool Support

There are two tool suites that support RAISE: The RAISE tools from CRI [28] and the RSLTC tool from UNU/IIST [19].

The RAISE tools from CRI are centered around the syntax oriented *eden editor* which can be used for developing and type checking modules. Automatic translation of executable specifications to Ada and C++ is supported, as well as development of proofs or informal justifications. Figure 3.2 shows the proof editor in use.

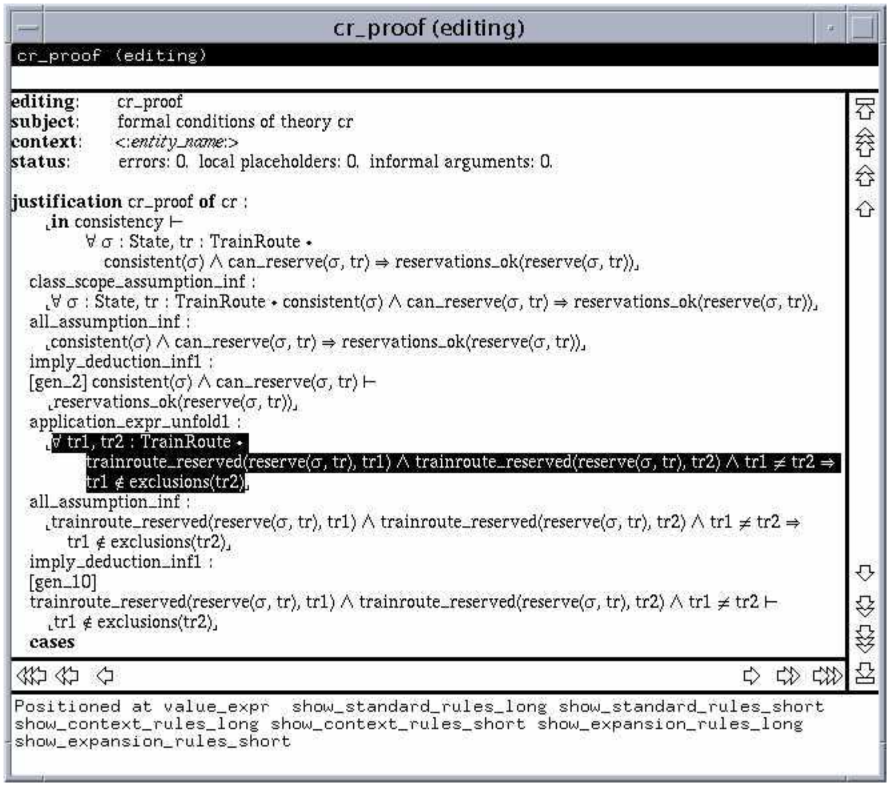

Figure 3.2: The eden editor supports proof development.

The RSLTC tool suports type checking and automatic translation to Ada, C++, SML, and PVS. The translation to the specification language of PVS provides proof support by means of PVS [17].

CHAPTER 4

An Institution for a Subset of RSL

In this chapter, an institution for a subset of RSL is defined. The subset is referred to as "mRSL" (mini-RSL).

The RSL subset of discourse is applicative and deterministic, and it comprises declarations of sort types and abbreviation types, declarations of values, and declarations of axioms. Variables and channels are not considered, i.e., declarations of variables and channels, assignments, communication on channels, and access descriptions in function types are not part of the subset. Moreover, only some of the built-in types and type operators are considered: the types **Bool**, **Int**, **Real**, and **Unit**, and the type operators $\times$ and $\xrightarrow{\sim}$. We consider value expressions involving value literals, names of user-declared values, the basic expressions **chaos** and **skip**, binary products, λ-abstractions, function applications, quantified expressions, built-in operators and connectives, and if-expressions. A summary of the syntax of mRSL is found in Appendix A.

According to the RAISE method, most proofs are made in the early development steps where specifications are applicative (cf. Section 3.3), and it is therefore a natural choice to consider an applicative subset.

Moreover, if we restrict our attention to an applicative and deterministic subset, a simple notion of processes suffices for giving semantics to value expressions. Consequently, the encoding in higher-order logic becomes simpler and easier to comprehend and use.

EXAMPLE 4.1 *The following RSL specification is in the subset referred to as mRSL:*

scheme C1 =

class
  type Colour
  value black : Colour,
      white : Colour
  axiom [black_not_white] (black $\neq$ white) $\equiv$ **true**
end

The original definition of RSL is not in the form of an institution. In [54], the syntax of RSL is given by a BNF grammar augmented with information about types, and the semantics is described informally. Moreover, a denotational semantics of RSL is given in [43]. However, there are some deviations from the description of the semantics found in [54]. Some of the deviations mentioned in [43] are that overloading is not handled fully, accesses are only given little attention, semantics of basic operators is not provided, and axioms regarding variant definitions are not incorporated. The definition of an institution for mRSL that is presented in this chapter is based on the original definitions of RSL found in [54] and [43]. Some, but not all, of the deviations in [43] concern the language features in mRSL. We have chosen to discard overloading, and adopt the semantics of basic operators that is described informally in [54].

Regarding products, RSL features general products with a semantics expressed using lists. In mRSL, however, only binary products are considered, and they are given a straightforward semantics without using lists.

In the following, we describe the components of the mRSL institution (signatures, sentences, models, and the satisfaction relation) before we prove the satisfaction condition and conclude the chapter with the actual definition of an institution for mRSL.

4.1 Signatures and Signature Morphisms

The first component of the mRSL institution comprises signatures and signature morphisms.

4.1.1 Signatures

A signature contains names of types and values that have been declared by the user/specifier and may be used in the axioms/sentences of the specification. Moreover, each value name is assigned a *static type* which is the abstract syntax of a type expression.

Definition 4.1 For a set X of identifiers, $T(X)$ is the least set that satisfies:

- $t \in T(X)$ if $t \in \{s\_bool, s\_int, s\_real, s\_unit\}$
- $t \in T(X)$ if $t \in X$

- $s\_prod(t_1, t_2) \in T(X)$ if $t_1 \in T(X)$ and $t_2 \in T(X)$
- $s\_fct(t_1, t_2) \in T(X)$ if $t_1 \in T(X)$ and $t_2 \in T(X)$.

The elements of $T(X)$ are *static types*.

The above definition says that for a set X of identifiers, $T(X)$ denotes the set of static types that can be generated using the basic types (s_bool, s_int, s_real, s_unit), identifiers in X, and the type constructors s_prod and s_fct.

The concept of an mRSL signature is defined as follows:

Definition 4.2 An *mRSL signature* Σ is a quadruple $\langle S, A, \Psi, V\rangle$ consisting of

- a set S of sort names
- a set A of abbreviation type names
- a function $\Psi : A \longrightarrow T(S \cup A)$ that maps abbreviation type names to static types that they are abbreviations for
- a $T(S)$-sorted set $V = (V_t)_{t \in T(S)}$ of value names

such that

- S and A are disjoint ($S \cap A = \emptyset$)
- type abbreviations (specified by Ψ) are not cyclic
- the components in V are disjoint ($V_i \cap V_j = \emptyset$ for $i \neq j$), i.e., no overloading is allowed.

The function Ψ is lifted to work on static types in the following way: If a static type t contains identifiers a_i of abbreviation types, we obtain the static type $\Psi(t)$ from t by replacing a_i with $\Psi(a_i)$. The result $\Psi(t)$ may contain identifiers a'_i of abbreviation types, in which case we obtain the static type $\Psi(\Psi(t))$ from $\Psi(t)$ by replacing a'_i with $\Psi(a'_i)$. Since the type abbreviations of a signature are assumed non-cyclic by Definition 4.2, they can be expanded repeatedly until the result is a static type without any abbreviation types (the canonical form of the static type).

Definition 4.3 Ψ^* is the function that takes a static type to its canonical form by recursively expanding abbrevation types as prescribed by Ψ.

If a static type does not contain identifiers of abbreviation types, applying Ψ^* has no effect. This is expressed by the following lemma:

Lemma 4.4 *For every mRSL signature $\langle S, A, \Psi, V\rangle$, we have $\Psi^*(s) = s$ for $s \in S$, and $\Psi^*(t) = t$ for $t \in T(S)$.*

Proof. Ψ^* only replaces occurrences of abbreviation type names. There are no occurrences of abbreviation type names in $s \in S$ nor in $t \in T(S)$, so $\Psi^*(s) = s$ and $\Psi^*(t) = t$. □

Signatures and mRSL Specifications

The signature of an mRSL specification[1] stems from the type and value declarations. Sort names and abbreviation type names are introduced by type declarations which also define the function Ψ. Value names are introduced by value declarations.

Example 4.2 *The signature of the specification*

scheme C2 =
  class
    type Intensity,
        Colour = **Bool**
    value black : Colour,
          white : Colour
    axiom [black_def] black ≡ **true**,
          [white_def] white ≡ **false**
  end

is $\langle\{\text{Intensity}\}, \{\text{Colour}\}, \Psi, V\rangle$ *where* $\Psi(\text{Colour}) = s\_bool$ *and the sorted set of values is given by* $V_{s\_bool} = \{\text{black}, \text{white}\}$ *and* $V_t = \emptyset$ *for* $t \in T(\{\text{Intensity}\})$ *and* $t \neq s\_bool$*.*

4.1.2 Signature Morphisms

Signatures may relate to each other and we formalize these relations by signature morphisms.

Definition 4.5 Let $\Sigma = \langle S, A, \Psi, V\rangle$ and $\Sigma' = \langle S', A', \Psi', V'\rangle$ be mRSL signatures. An *mRSL signature morphism* $\sigma : \Sigma \longrightarrow \Sigma'$ is a triple $\sigma = \langle \sigma_S, \sigma_A, \sigma_V\rangle$ consisting of

- a mapping $\sigma_S : S \longrightarrow S' \cup A'$ of sort names
- a mapping $\sigma_A : A \longrightarrow A'$ of abbreviation type names
- a mapping $\sigma_V : V \longrightarrow V'$ of value names

such that

- the mapping σ_V is $T(S)$-sorted and respects σ_S:

  $$\sigma_V = (\sigma_{V_t} : V_t \longrightarrow V'_{\Psi'^*(T(\sigma_S)(t))})_{t \in T(S)}$$

  where $T(\sigma_S)$ is σ_S lifted to work on static types in $T(S)$ by replacing occurrences of sort names as prescribed by σ_S:

[1] Since mRSL is a subset of RSL, an mRSL specification is an RSL specification that only makes use of the features included in mRSL.

- $T(\sigma_S)(s) = \sigma_S(s)$ if $s \in S$
- $T(\sigma_S)(t) = t$ if $t \in \{s\_bool, s\_int, s\_real, s\_unit\}$
- $T(\sigma_S)(s\_prod(t_1, t_2)) = s\_prod(T(\sigma_S)(t_1), T(\sigma_S)(t_2))$
- $T(\sigma_S)(s\_fct(t_1, t_2)) = s\_fct(T(\sigma_S)(t_1), T(\sigma_S)(t_2))$

- the following diagram commutes:

$$\begin{tikzcd} A \arrow[rr, "\sigma_A"] \arrow[d, "\Psi^*"'] & & A' \arrow[d, "\Psi'^*"] \\ T(S) \arrow[r, "T(\sigma_S)"'] & T(S' \cup A') \arrow[r, "\Psi'^*"'] & T(S') \end{tikzcd}$$

In other words, $\Psi'^*(\sigma_A(a)) = \Psi'^*(T(\sigma_S)(\Psi^*(a)))$ for $a \in A$.

Signature Morphisms and mRSL Specifications

Two specifications are related if one is a refinement of the other or if it is obtained from the other by a specification-building operation. Such relations between specifications may be described as signature morphisms between the signatures of the specifications.

Recall the definition of refinement in RSL given in Section 3.3. A specification $SP' = \langle \Sigma', E' \rangle$ *statically refines* a specification $SP = \langle \Sigma, E \rangle$ if and only if there is a signature morphism $\sigma = \langle \sigma_S, \sigma_A, \sigma_V \rangle$ from Σ to Σ' such that

- $\sigma_S(x) = x$ and $x \in S' \cup A'$ for $x \in S$, i.e., $S \subseteq S' \cup A'$
- $\sigma_A(a) = a$ and $a \in A'$ for $a \in A$, i.e., $A \subseteq A'$
- $\sigma_{V_t}(v) = v$ and $v \in V'_{\Psi'^*(T(\sigma_S)(t))}$ for $t \in T(S)$ and $v \in V_t$.

In other words, the components of σ are inclusions.

In RSL refinement steps, declarations of sort types may be replaced by declarations of abbreviation types (with the same names). Consequently, σ_S may map a sort type into an abbreviation type.

Example 4.3 *Recall specifications* C1 *and* C2 *from Example 4.1 and Example 4.2. There is a signature morphism from the signature of* C1 *to the signature of* C2 *given by:*

- $\sigma_S(\text{Colour}) = \text{Colour}$
- $\sigma_V(\text{black}) = \text{black}$
- $\sigma_V(\text{white}) = \text{white}$

Furthermore, C2 *implements* C1.

We say that a signature morphism $\langle \sigma_S, \sigma_A, \sigma_V \rangle$ is an *injection*, if σ_S, σ_A, and the components of σ_V are in injective.

If a specification SP' is an *extension* of a specification SP, then the signature of SP is *included* in the signature of SP' and there is a signature morphism from $Sig(SP)$ into $Sig(SP')$ which includes names of sorts, abbreviation types, and values. We say that a signature morphism $\langle \sigma_S, \sigma_A, \sigma_V \rangle$ is an *inclusion*, if σ_S, σ_A, and the components of σ_V are inclusions.

If a specification SP' is obtained from a specification SP by *renaming* entities, then there is a signature morphism from $Sig(SP)$ to $Sig(SP')$ which, in a one-to-one manner, renames sorts, abbreviation types, and values. We say that a signature morphism $\langle \sigma_S, \sigma_A, \sigma_V \rangle$ is a *renaming*, if σ_S, σ_A, and the components of σ_V are bijective (i.e., surjective and injective) and $\sigma_S(s) \in S'$ for all $s \in S$.

Composition of Signature Morphisms

In the following, let $\Sigma = \langle S, A, \Psi, V \rangle$, $\Sigma' = \langle S', A', \Psi', V' \rangle$, and $\Sigma'' = \langle S'', A'', \Psi'', V'' \rangle$ be signatures; and let $\sigma = \langle \sigma_S, \sigma_A, \sigma_V \rangle$ and $\sigma' = \langle \sigma'_{S'}, \sigma'_{A'}, \sigma'_{V'} \rangle$ be signature morphisms $\Sigma \xrightarrow{\sigma} \Sigma'$ and $\Sigma' \xrightarrow{\sigma'} \Sigma''$, respectively.

The composition of two signature morphisms is defined as follows:

Definition 4.6 The composition $\sigma' \circ \sigma$ of σ and σ' is the morphism $\Sigma \xrightarrow{\sigma' \circ \sigma} \Sigma''$ consisting of the components given by

- $(\sigma' \circ \sigma)_S(s) = \begin{cases} \sigma'_{S'}(\sigma_S(s)) & \text{if } \sigma_S(s) \in S' \\ \sigma'_{A'}(\sigma_S(s)) & \text{if } \sigma_S(s) \in A' \end{cases} \qquad \text{for } s \in S$
- $(\sigma' \circ \sigma)_A = \sigma'_{A'} \circ \sigma_A$
- $(\sigma' \circ \sigma)_{V_t} = \sigma'_{V'_{\Psi'^*(T(\sigma_S)(t))}} \circ \sigma_{V_t}$ for $t \in T(S)$.

Note that $\sigma'_{S'}(\sigma_S(s)) \in S'' \cup A''$ and $\sigma'_{A'}(\sigma_S(s)) \in A''$ in the above definition so that $(\sigma' \circ \sigma)_S$ is a function $S \longrightarrow S'' \cup A''$.

The following lemma is used for proving that composition of signature morphisms is associative:

Lemma 4.7 $\Psi''^*(T((\sigma' \circ \sigma)_S)(t)) = \Psi''^*(T(\sigma'_{S'})(\Psi'^*(T(\sigma_S)(t))))$ *for* $t \in T(S)$.

Proof. Let $t \in T(S)$. In the following, we shall view t as a binary tree where the nodes represent applications of type operators and the leaves represent names of built-in types and sort types. Dealing with static types, the type operators are *s_prod* and *s_fct* which represents $\times$ and $\tilde{\rightarrow}$, respectively. The built-in types are *s_bool*, *s_int*, *s_real*, and *s_unit* which represents **Bool**, **Int**, **Real**, and **Unit**, respectively.

If t does not contain sort type names, the lemma clearly holds (by definition of $T(\sigma_S)$ and Lemma 4.4).

Assume that t contains sort type names s_1 and s_2. This is illustrated by:

$$\begin{array}{ccc} & t & \\ & \triangle & \\ s_1 & & s_2 \end{array}$$

The definition of $T(\sigma_S)$ prescribes that when $T(\sigma_S)$ is applied to t, then σ_S is applied to s_1 and s_2:

$$\begin{array}{ccc} & T(\sigma_S)(t) & \\ & \triangle & \\ \sigma_S(s_1) & & \sigma_S(s_2) \end{array}$$

For a sort type name $s \in S$, the result of $\sigma_S(s)$ is either in S' or in A'. To consider both cases, let $\sigma_S(s_1) \in S'$ and $\sigma_S(s_2) \in A'$. By Lemma 4.4, applying Ψ'^* yields:

$$\begin{array}{ccc} & \Psi'^*(T(\sigma_S)(t)) & \\ & \triangle & \\ \sigma_S(s_1) & & \Psi'^*(\sigma_S(s_2)) \end{array}$$

Applying $T(\sigma'_{S'})$ yields:

$$\begin{array}{ccc} & T(\sigma'_{S'})(\Psi'^*(T(\sigma_S)(t))) & \\ & \triangle & \\ \sigma'_{S'}(\sigma_S(s_1)) & & T(\sigma'_{S'})(\Psi'^*(\sigma_S(s_2))) \end{array}$$

And applying Ψ''^* yields:

$$\begin{array}{ccc} & \Psi''^*(T(\sigma'_{S'})(\Psi'^*(T(\sigma_S)(t)))) & \\ & \triangle & \\ \Psi''^*(\sigma'_{S'}(\sigma_S(s_1))) & & \Psi''^*(T(\sigma'_{S'})(\Psi'^*(\sigma_S(s_2)))) \end{array}$$

Now apply $T((\sigma' \circ \sigma)_S)$ to t:

$$\begin{array}{ccc} & T((\sigma' \circ \sigma)_S)(t) & \\ & \triangle & \\ \sigma'_{S'}(\sigma_S(s_1)) & & \sigma'_{A'}(\sigma_S(s_2)) \end{array}$$

Applying Ψ''^* yields:

$$\begin{array}{ccc} & \Psi''^*(T((\sigma' \circ \sigma)_S)(t)) & \\ & \triangle & \\ \Psi''^*(\sigma'_{S'}(\sigma_S(s_1))) & & \Psi''^*(\sigma'_{A'}(\sigma_S(s_2))) \end{array}$$

Since $\sigma_S(s_2) \in A'$, and $\sigma'_{S'}$ and $\sigma'_{A'}$ are components of a signature morphism, we have $\Psi''^*(T(\sigma'_{S'})(\Psi'^*(\sigma_S(s_2)))) = \Psi''^*(\sigma'_{A'}(\sigma_S(s_2)))$ by the commuting diagram in Definition 4.5. Consequently, the trees representing $\Psi''^*(T(\sigma'_{S'})(\Psi'^*(T(\sigma_S)(t))))$ and $\Psi''^*(T((\sigma' \circ \sigma)_S)(t))$ are equal. $\square$

The following proposition states that composition is associative:

Proposition 4.8 *For all signature morphisms* $\sigma : \Sigma \longrightarrow \Sigma'$, $\sigma' : \Sigma' \longrightarrow \Sigma''$, *and* $\sigma'' : \Sigma'' \longrightarrow \Sigma'''$:

$$\sigma'' \circ (\sigma' \circ \sigma) = (\sigma'' \circ \sigma') \circ \sigma.$$

Proof. Composition is defined componentwise, and associativity is shown componentwise.

Let $s \in S$ and $s' \in S'$. By definition of composition:

$$(\sigma' \circ \sigma)_S(s) = \begin{cases} \sigma'_{S'}(\sigma_S(s)) \in S'' \cup A'' & \text{if } \sigma_S(s) \in S' \\ \sigma'_{A'}(\sigma_S(s)) \in A'' & \text{if } \sigma_S(s) \in A' \end{cases}$$

and

$$(\sigma'' \circ \sigma')_{S'}(s') = \begin{cases} \sigma''_{S''}(\sigma'_{S'}(s')) \in S''' \cup A''' & \text{if } \sigma'_{S'}(s') \in S'' \\ \sigma''_{A'}(\sigma'_{S'}(s')) \in A''' & \text{if } \sigma'_{S'}(s') \in A''. \end{cases}$$

Furthermore, by definition of composition:

$$(\sigma'' \circ (\sigma' \circ \sigma))_S(s) = \begin{cases} \sigma''_{S''}((\sigma' \circ \sigma)_S(s)) \in S''' \cup A''' & \text{if } (\sigma' \circ \sigma)_S(s) \in S'' \\ \sigma''_{A''}((\sigma' \circ \sigma)_S(s)) \in A''' & \text{if } (\sigma' \circ \sigma)_S(s) \in A'' \end{cases}$$

which expands to:

$$(\sigma''\circ(\sigma'\circ\sigma))_S(s) = \begin{cases} \sigma''_{S''}(\sigma'_{S'}(\sigma_S(s))) \in S''' \cup A''' & \text{if } \sigma'_{S'}(\sigma_S(s)) \in S'' \text{ and } \sigma_S(s) \in S' \\ \sigma''_{A''}(\sigma'_{S'}(\sigma_S(s))) \in A''' & \text{if } \sigma'_{S'}(\sigma_S(s)) \in A'' \text{ and } \sigma_S(s) \in S' \\ \sigma''_{A''}(\sigma'_{A'}(\sigma_S(s))) \in A''' & \text{if } \sigma'_{A'}(\sigma_S(s)) \in A'' \text{ and } \sigma_S(s) \in A'. \end{cases}$$

Moreover, by definition of composition:

$$((\sigma'' \circ \sigma') \circ \sigma)_S(s) = \begin{cases} (\sigma'' \circ \sigma')_{S'}(\sigma_S(s)) \in S''' \cup A''' & \text{if } \sigma_S(s) \in S' \\ (\sigma'' \circ \sigma')_{A'}(\sigma_S(s)) \in A''' & \text{if } \sigma_S(s) \in A' \end{cases}$$

which expands to:

$$((\sigma''\circ\sigma')\circ\sigma)_S(s) = \begin{cases} \sigma''_{S''}(\sigma'_{S'}(\sigma_S(s))) \in S''' \cup A''' & \text{if } \sigma'_{S'}(\sigma_S(s)) \in S'' \text{ and } \sigma_S(s) \in S' \\ \sigma''_{A''}(\sigma'_{S'}(\sigma_S(s))) \in A''' & \text{if } \sigma'_{S'}(\sigma_S(s)) \in A'' \text{ and } \sigma_S(s) \in S' \\ \sigma''_{A''}(\sigma'_{A'}(\sigma_S(s))) \in A''' & \text{if } \sigma'_{A'}(\sigma_S(s)) \in A'' \text{ and } \sigma_S(s) \in A'. \end{cases}$$

Consequently, $(\sigma'' \circ (\sigma' \circ \sigma))_S = ((\sigma'' \circ \sigma') \circ \sigma)_S$.

Let $a \in A$. By definition of composition:

$$\begin{aligned} & (\sigma'' \circ (\sigma' \circ \sigma))_A(a) \\ = \; & \sigma''_{A''}((\sigma' \circ \sigma)_A(a)) \\ = \; & \sigma''_{A''}(\sigma'_{A'}(\sigma_A(a))) \\ = \; & (\sigma'' \circ \sigma')_{A'}(\sigma_A(a)) \end{aligned}$$

$$= ((\sigma'' \circ \sigma') \circ \sigma)_A(a).$$

Consequently, $(\sigma'' \circ (\sigma' \circ \sigma))_A = ((\sigma'' \circ \sigma') \circ \sigma)_A$.

Let $t \in T(S)$ and $v \in V_t$. We then have:

$$
\begin{aligned}
& (\sigma'' \circ (\sigma' \circ \sigma))_{V_t}(v) \\
= {} & (\sigma''_{V''_{\Psi''^*(T((\sigma' \circ \sigma)_S)(t))}} \circ (\sigma' \circ \sigma)_{V_t})(v) \\
= {} & \sigma''_{V''_{\Psi''^*(T((\sigma' \circ \sigma)_S)(t))}}((\sigma' \circ \sigma)_{V_t}(v)) \\
= {} & \sigma''_{V''_{\Psi''^*(T((\sigma' \circ \sigma)_S)(t))}}((\sigma'_{V'_{\Psi'^*(T(\sigma_S)(t))}} \circ \sigma_{V_t})(v)) \\
= {} & \sigma''_{V''_{\Psi''^*(T((\sigma' \circ \sigma)_S)(t))}}(\sigma'_{V'_{\Psi'^*(T(\sigma_S)(t))}}(\sigma_{V_t}(v))).
\end{aligned}
$$

Moreover,

$$
\begin{aligned}
& ((\sigma'' \circ \sigma') \circ \sigma)_{V_t}(v) \\
= {} & ((\sigma'' \circ \sigma')_{V'_{\Psi'^*(T(\sigma_S)(t))}} \circ \sigma_{V_t})(v) \\
= {} & (\sigma'' \circ \sigma')_{V'_{\Psi'^*(T(\sigma_S)(t))}}(\sigma_{V_t}(v)) \\
= {} & (\sigma''_{V''_{\Psi''^*(T(\sigma'_{S'})(\Psi'^*(T(\sigma_S)(t))))}} \circ \sigma'_{V'_{\Psi'^*(T(\sigma_S)(t))}})(\sigma_{V_t}(v)) \\
= {} & \sigma''_{V''_{\Psi''^*(T(\sigma'_{S'})(\Psi'^*(T(\sigma_S)(t))))}}(\sigma'_{V'_{\Psi'^*(T(\sigma_S)(t))}}(\sigma_{V_t}(v))).
\end{aligned}
$$

By Lemma 4.7, $(\sigma'' \circ (\sigma' \circ \sigma))_V = ((\sigma'' \circ \sigma') \circ \sigma)_V$.

Consequently, $\sigma'' \circ (\sigma' \circ \sigma) = (\sigma'' \circ \sigma') \circ \sigma$. □

Identities on Signatures

Some signature morphisms are endomorphisms with certain properties.

Definition 4.9 Let $\Sigma = \langle S, A, \Psi, V \rangle$. The *identity* $id_\Sigma : \Sigma \longrightarrow \Sigma$ is the signature morphism $\langle id_S, id_A, id_V \rangle$ where

- id_S is the identity on S
- id_A is the identity on A
- $id_V = (id_{V_t} : V_t \longrightarrow V_t)_{t \in T(S)}$ is the $T(S)$-sorted mapping of value names whose components are the identities on the sets V_t of value names.

Proposition 4.10 *For every mRSL signature Σ, the identity $id_\Sigma : \Sigma \longrightarrow \Sigma$ satisfies $\sigma \circ id_\Sigma = \sigma$ for every $\sigma : \Sigma \longrightarrow \Sigma'$ and $id_\Sigma \circ \sigma' = \sigma'$ for every $\sigma' : \Sigma' \longrightarrow \Sigma$.*

PROOF. To show that $\sigma \circ id_\Sigma = \sigma$, the individual components of $\sigma \circ id_\Sigma$ are considered.

For $s \in S$:

$$
\begin{aligned}
& (\sigma \circ id_\Sigma)_S(s) \\
= & \begin{cases} \sigma_S(id_S(s)) & \text{if } id_S(s) \in S \\ \sigma_A(id_S(s)) & \text{if } id_S(s) \in A \end{cases} \\
= & \ \sigma_S(id_S(s)) \\
= & \ \sigma_S(s).
\end{aligned}
$$

Consequently, $(\sigma \circ id_\Sigma)_S = \sigma_S$. Moreover, $(\sigma \circ id_\Sigma)_A = \sigma_A \circ id_A = \sigma_A$ since id_A is the identity on A, and $(\sigma \circ id_\Sigma)_{V_t} = \sigma_{V_{\Psi^*(T(id_S)(t))}} \circ id_{V_t} = \sigma_{V_{\Psi^*(t)}} \circ id_{V_t} = \sigma_{V_t} \circ id_{V_t} = \sigma_{V_t}$. Thus, we have shown that $\sigma \circ id_\Sigma = \sigma$.

To show that $id_\Sigma \circ \sigma' = \sigma'$, the individual components of $id_\Sigma \circ \sigma'$ are considered.

For $s' \in S'$:

$$
\begin{aligned}
& (id_\Sigma \circ \sigma')_{S'}(s') \\
= & \begin{cases} id_S(\sigma'_{S'}(s')) & \text{if } \sigma'_{S'}(s') \in S \\ id_A(\sigma'_{S'}(s')) & \text{if } \sigma'_{S'}(s') \in A \end{cases} \\
= & \begin{cases} \sigma'_{S'}(s') & \text{if } \sigma'_{S'}(s') \in S \\ \sigma'_{S'}(s') & \text{if } \sigma'_{S'}(s') \in A \end{cases} \\
= & \ \sigma'_{S'}(s').
\end{aligned}
$$

Consequently, $(id_\Sigma \circ \sigma')_{S'} = \sigma'_{S'}$. Moreover, $(id_\Sigma \circ \sigma')_{A'} = id_A \circ \sigma'_{A'} = \sigma'_{A'}$ since id_A is the identity on A, and $(id_\Sigma \circ \sigma')_{V'_{t'}} = id_{V_{\Psi^*(T(\sigma'_{S'})(t'))}} \circ \sigma'_{V'_{t'}} = \sigma'_{V'_{t'}}$. Thus, we have shown that $id_\Sigma \circ \sigma' = \sigma'$. □

4.1.3 The Category $Sign_{mRSL}$

The mRSL signatures and mRSL signature morphisms form a category:

- Each signature morphism has a domain and a codomain which are signatures.
- For all signature morphisms $\sigma : \Sigma \longrightarrow \Sigma'$ and $\sigma' : \Sigma' \longrightarrow \Sigma''$, the composition $(\sigma' \circ \sigma) : \Sigma \longrightarrow \Sigma''$ is defined.
- Composition is associative.
- For every signature Σ, the identity id_Σ on Σ is defined.

DEFINITION 4.11 $Sign_{mRSL}$ is the category of mRSL signatures and mRSL signature morphisms.

As described above, signature morphisms may be used to express static implementation/refinement and static relations associated with specification-building operations.

Composition of refinements gives a new refinement, i.e., refinement is transitive. This is important for stepwise development.

Furthermore, introducing composition enabled the definition of a category of signatures and signature morphisms. This category is used in the definition of an institution for mRSL.

4.2 Sentences and Sentence Morphisms

Sentences are what you can write in axioms, and sentences are written using the vocabulary provided by a signature. In mRSL specifications, axioms are named value expressions of a certain form and type. Consequently, value expressions must be considered, in order to define sentences.

In the following, type expressions are considered first. Then an account of value expressions is given, sentences are defined on the basis of value expressions, and sentence morphisms are defined by lifting signature morphisms. Finally, a functor mapping signatures to sets of sentences is defined.

4.2.1 Type Expressions

The syntax of mRSL type expressions is introduced first, and then the static semantics is presented.

Syntax of Type Expressions

Type expressions over a signature $\langle S, A, \Psi, V \rangle$ are formed using RSL type literals, sort names in S, abbreviation type names in A, and type operators. However, not all RSL type expressions are considered in mRSL.

Type expressions over a signature $\Sigma = \langle S, A, \Psi, V \rangle$ are of the following forms:

Bool	(the type of truth values)
Int	(the type of integers)
Real	(the type of reals)
Unit	(the singleton type)
$typeexpr_1 \times typeexpr_2$	(binary product type)
$typeexpr_1 \xrightarrow{\sim} typeexpr_2$	(partial functions)
$type\_id$	(type name; $type\_id \in S$ or $type\_id \in A$)
$(typeexpr)$	(bracketed type).

The type operator $\xrightarrow{\sim}$ associates to the right and it has lower precedence than $\times$. Consequently, $t_1 \xrightarrow{\sim} t_2 \times t_3 \xrightarrow{\sim} t_4$ means $t_1 \xrightarrow{\sim} ((t_2 \times t_3) \xrightarrow{\sim} t_4)$.

Static Semantics of Type Expressions

Well-formedness of type expressions is laid down by the static semantics given by a set of inference rules. In the static semantics, a type expression denotes a static type. A type expression t_expr is well-formed and denotes the static type t in the context of the signature Σ if there exists a proof of the judgement $\Sigma \vdash t\_expr \rhd t$ constructed from the inference rules of the static semantics. The rules are:

$$\frac{}{\Sigma \vdash \mathbf{Bool} \rhd s\_bool} \qquad \frac{}{\Sigma \vdash \mathbf{Int} \rhd s\_int}$$

$$\frac{}{\Sigma \vdash \mathbf{Real} \rhd s\_real} \qquad \frac{}{\Sigma \vdash \mathbf{Unit} \rhd s\_unit}$$

$$\frac{\Sigma = \langle S, A, \Psi, V \rangle \quad s \in S}{\Sigma \vdash s \rhd s} \qquad \frac{\Sigma = \langle S, A, \Psi, V \rangle \quad a \in A \quad \Psi^*(a) = t}{\Sigma \vdash a \rhd t}$$

$$\frac{\Sigma \vdash te_1 \rhd t_1 \quad \Sigma \vdash te_2 \rhd t_2}{\Sigma \vdash te_1 \times te_2 \rhd s\_prod(t_1, t_2)} \quad \frac{\Sigma \vdash te_1 \rhd t_1 \quad \Sigma \vdash te_2 \rhd t_2}{\Sigma \vdash te_1 \stackrel{\sim}{\to} te_2 \rhd s\_fct(t_1, t_2)} \quad \frac{\Sigma \vdash te \rhd t}{\Sigma \vdash (te) \rhd t}$$

We let $TypeExpr(\Sigma)$ denote the set of well-formed type expressions over the signature Σ. For each signature $\Sigma = \langle S, A, \Psi, V \rangle$, the static semantics maps type expressions in $TypeExpr(\Sigma)$ to static types in $T(S)$.

4.2.2 Value Expressions

The syntax of mRSL value expression is introduced first, and then the static semantics is presented.

Syntax of Value Expressions

Value expressions over a signature $\Sigma = \langle S, A, \Psi, V \rangle$ are formed according to the grammar of RSL using value literals, value names in V, basic expressions, product expressions, function expressions (λ-abstractions), function application, quantified expressions, infix expressions, prefix expressions, and if-expressions. Function expressions and quantified expressions may contain type expressions in $TypeExpr(\Sigma)$.

Value expressions over a signature $\Sigma = \langle S, A, \Psi, V \rangle$ are of the following forms:

true	(Boolean literal)
false	(Boolean literal)
i	(integer literal, i.e., a non-empty sequence of digits)
r	(real literal of the form x.0 where x is a non-empty sequence of digits)
()	(the unit literal)
v	(value names where $v \in V$)
chaos	(basic expression)
skip	(basic expression)
$(expr_1,expr_2)$	(product expression)
$\lambda\ x : te \bullet expr$	(function expression)
$expr\ afp$	(application expression)
$\forall\ x : te \bullet expr$	(universally quantified expression)
$\exists\ x : te \bullet expr$	(existentially quantified expression)
$expr_1 \equiv expr_2$	(equivalence expression)
$(expr)$	(bracketed expression)
$expr_1\ op\ expr_2$	(value infix expression)
$expr_1\ conn\ expr_2$	(axiom infix expression)
$op\ expr$	(value prefix expression)
$conn\ expr$	(axiom prefix expression)
if $expr_1$ **then** $expr_2$ **else** $expr_3$ **end**	(structured expression)

where te ranges over type expressions in $TypeExpr(\Sigma)$, and $expr$ (possibly with an index) ranges over value expressions. In application expressions, afp is the actual function parameter which is of one of the forms (), $(expr)$, or $(expr_1,expr_2)$. Furthermore, op and $conn$ range over operators and connectives, respectively, that are considered below.

In function expressions and quantified expressions, identifiers are introduced. In the above, x ranges over such identifiers which are required not to be value names[2] in the signatures. In other words, the name spaces for value names and bound variable names are assumed to be disjoint. As a consequence, a bound variable x cannot occur in a signature as a value name. This requirement is introduced to ease the later definition of the functor Sen_{mRSL}. Similar requirements are seen in the institution for HOL presented in Chapter 5.

The requirements on identifiers in function expressions and quantified expressions are enforced by the static semantics of value expressions presented below.

The following infix operators, op, may occur in the value infix expressions:

[2]This is a deviation from RSL, in which identifiers introduced by typings in function expressions and quantified expressions can overshadow value names. Allowing this, however, causes problems when signature morphisms are lifted to work on sentences and it must be proved that there is a functor that maps signatures to sets of sentences and signature morphisms to functions between sets of sentences.

Equality	$= : \mathrm{T} \times \mathrm{T} \xrightarrow{\sim} \mathbf{Bool}$
Inequality	$\neq : \mathrm{T} \times \mathrm{T} \xrightarrow{\sim} \mathbf{Bool}$
Integer less than	$< : \mathbf{Int} \times \mathbf{Int} \xrightarrow{\sim} \mathbf{Bool}$
Real less than	$< : \mathbf{Real} \times \mathbf{Real} \xrightarrow{\sim} \mathbf{Bool}$
Integer greater than	$> : \mathbf{Int} \times \mathbf{Int} \xrightarrow{\sim} \mathbf{Bool}$
Real greater than	$> : \mathbf{Real} \times \mathbf{Real} \xrightarrow{\sim} \mathbf{Bool}$
Integer less than or equal to	$\leq : \mathbf{Int} \times \mathbf{Int} \xrightarrow{\sim} \mathbf{Bool}$
Real less than or equal to	$\leq : \mathbf{Real} \times \mathbf{Real} \xrightarrow{\sim} \mathbf{Bool}$
Integer greater than or equal to	$\geq : \mathbf{Int} \times \mathbf{Int} \xrightarrow{\sim} \mathbf{Bool}$
Real greater than or equal to	$\geq : \mathbf{Real} \times \mathbf{Real} \xrightarrow{\sim} \mathbf{Bool}$
Integer addition	$+ : \mathbf{Int} \times \mathbf{Int} \xrightarrow{\sim} \mathbf{Int}$
Real addition	$+ : \mathbf{Real} \times \mathbf{Real} \xrightarrow{\sim} \mathbf{Real}$
Integer subtraction	$- : \mathbf{Int} \times \mathbf{Int} \xrightarrow{\sim} \mathbf{Int}$
Real subtraction	$- : \mathbf{Real} \times \mathbf{Real} \xrightarrow{\sim} \mathbf{Real}$
Integer multiplication	$* : \mathbf{Int} \times \mathbf{Int} \xrightarrow{\sim} \mathbf{Int}$
Real multiplication	$* : \mathbf{Real} \times \mathbf{Real} \rightarrow \mathbf{Real}$
Integer division	$/ : \mathbf{Int} \times \mathbf{Int} \xrightarrow{\sim} \mathbf{Int}$
Integer remainder	$\backslash : \mathbf{Int} \times \mathbf{Int} \xrightarrow{\sim} \mathbf{Int}$.

The following infix connectives, *conn*, may occur in the axiom infix expressions:

Boolean and	$\wedge$
Boolean or	$\vee$
Boolean implication	$\Rightarrow$.

The prefix operator **abs** may occur in value prefix expressions:

Integer absolute value	$\mathbf{abs} : \mathbf{Int} \xrightarrow{\sim} \mathbf{Int}$
Real absolute value	$\mathbf{abs} : \mathbf{Real} \xrightarrow{\sim} \mathbf{Real}$.

The prefix connective $\sim$ may occur in axiom prefix expressions:

Boolean not $\sim$.

Static Semantics of Value Expressions

Type-correctness of value expressions is defined by a static semantics expressed for mRSL in a set of inference rules defining type assertions of the form $\Sigma \vdash ve \rhd t$ where Σ is a signature, ve is a value expression, and t is a static type. A value expression ve is well-formed and has the static type t in the context of the signature Σ if there exists a proof of the judgement $\Sigma \vdash ve \rhd t$ constructed from the inference rules of the static semantics.

Additional notions regarding bound variables and mRSL signatures are introduced.

Definition 4.12 Let $\Sigma = \langle S, A, \Psi, V \rangle$ be an mRSL signature. A *Σ-signature fragment* is a pair (x, st) where x is the name of a bound variable and st is a static type in $T(S)$.

We say that (x, st) is the signature fragment that maps x to st.

Definition 4.13 Let $\Sigma = \langle S, A, \Psi, V \rangle$ be an mRSL signature, and let (x, st) be a Σ-signature fragment. Then $\Sigma \dagger (x, st) = \langle S, A, \Psi, V^\dagger \rangle$ where $V^\dagger = (V_t^\dagger)_{t \in T(S)}$ such that $V_{st}^\dagger = V_{st} \cup \{x\}$, and $V_t^\dagger = V_t$ for $t \in T(S)$ and $t \neq st$.

When we write $\Sigma \dagger (x, st)$, we say "Σ extended with x", and we regard $\Sigma \dagger (x, st)$ as a signature although it contains x which, strictly speaking, should not be in a signature. However, it is convenient to allow it in this case, and since only one signature is considered at a time at this point, problems regarding signature morphisms and sentence morphisms do not occur.

In the inference rules presented in the remaining part of this section, we use a series of schematic variables:

Σ	for signatures
Δ	for signature fragments
t	for static types
i	for integer literals
r	for real literals
v	for identifiers/value names
ve	for value expressions
te	for type expressions
x	for bound variables in typings
afp	for actual function parameters: (), (ve), or (ve_1, ve_2)
op	for infix and prefix operators
$conn$	for infix and prefix connectives.

Some of the schematic variables are indexed, e.g., ve_1 and ve_2.

The static semantics of mRSL value expressions is now given by a series of inference rules.

We let $ValExpr(\Sigma)$ denote the set of type-correct value expressions over the signature Σ. For each signature $\Sigma = \langle S, A, \Psi, V \rangle$, the static semantics maps value expressions in $ValExpr(\Sigma)$ to static types in $T(S)$.

Static Semantics of Value Literals

$$\frac{}{\Sigma \vdash \textbf{true} \rhd s\_bool} \qquad \frac{}{\Sigma \vdash \textbf{false} \rhd s\_bool}$$

$$\frac{}{\Sigma \vdash i \rhd s\_int} \qquad \frac{}{\Sigma \vdash r \rhd s\_real} \qquad \frac{}{\Sigma \vdash () \rhd s\_unit}$$

Static Semantics of Value Names

$$\frac{\Sigma = \langle S, A, \Psi, V \rangle \quad t \in T(S) \quad v \in V_t}{\Sigma \vdash v \rhd t}$$

Static Semantics of Basic Expressions

$$\frac{\Sigma = \langle S, A, \Psi, V \rangle \quad t \in T(S)}{\Sigma \vdash \textbf{chaos} \rhd t} \qquad \frac{}{\Sigma \vdash \textbf{skip} \rhd s\_unit}$$

Static Semantics of Product Expressions

$$\frac{\Sigma \vdash ve_1 \rhd t_1 \quad \Sigma \vdash ve_2 \rhd t_2}{\Sigma \vdash (ve_1,ve_2) \rhd s\_prod(t_1,t_2)}$$

Static Semantics of Function Expressions

$$\frac{\Sigma = \langle S, A, \Psi, V\rangle \quad \forall\, t \in T(S)\,.\, x \notin V_t \quad \Sigma \vdash te \rhd t_1 \quad \Sigma \dagger (x, t_1) \vdash ve \rhd t_2}{\Sigma \vdash \lambda\ x : te \bullet ve \rhd s\_fct(t_1, t_2)}$$

Static Semantics of Application Expressions

$$\frac{\Sigma \vdash ve \rhd s\_fct(t_1, t_2) \quad \Sigma \vdash afp \rhd t_1}{\Sigma \vdash ve\ afp \rhd t_2}$$

Static Semantics of Quantified Expressions

$$\frac{\Sigma = \langle S, A, \Psi, V\rangle \quad \forall\, t' \in T(S)\,.\, x \notin V_{t'} \quad \Sigma \vdash te \rhd t \quad \Sigma \dagger (x, t) \vdash ve \rhd s\_bool}{\Sigma \vdash \forall\ x : te \bullet ve \rhd s\_bool}$$

$$\frac{\Sigma = \langle S, A, \Psi, V\rangle \quad \forall\, t' \in T(S)\,.\, x \notin V_{t'} \quad \Sigma \vdash te \rhd t \quad \Sigma \dagger (x, t) \vdash ve \rhd s\_bool}{\Sigma \vdash \exists\ x : te \bullet ve \rhd s\_bool}$$

Static Semantics of Equivalence Expressions

$$\frac{\Sigma \vdash ve_1 \rhd t \quad \Sigma \vdash ve_2 \rhd t}{\Sigma \vdash ve_1 \equiv ve_2 \rhd s\_bool}$$

Static Semantics of Bracketed Expressions

$$\frac{\Sigma \vdash ve \rhd t}{\Sigma \vdash (ve) \rhd t}$$

Static Semantics of Value Infix Expressions The types of the infix operators are defined by the following rules:

$$\frac{}{\vdash\ =\ \rhd\ s\_fct(s\_prod(t,t), s\_bool)}$$

$$\frac{}{\vdash\ \neq\ \rhd\ s\_fct(s\_prod(t,t), s\_bool)}$$

$$\frac{}{\vdash\ <\ \rhd\ s\_fct(s\_prod(s\_int, s\_int), s\_bool)}$$

$$\frac{}{\vdash\ <\ \rhd\ s\_fct(s\_prod(s\_real, s\_real), s\_bool)}$$

$$\frac{}{\vdash\ >\ \rhd\ s\_fct(s\_prod(s\_int, s\_int), s\_bool)}$$

$$\overline{\vdash \ > \ \rhd \ s\_fct(s\_prod(s\_real, s\_real), s\_bool)}$$

$$\overline{\vdash \ \leq \ \rhd \ s\_fct(s\_prod(s\_int, s\_int), s\_bool)}$$

$$\overline{\vdash \ \leq \ \rhd \ s\_fct(s\_prod(s\_real, s\_real), s\_bool)}$$

$$\overline{\vdash \ \geq \ \rhd \ s\_fct(s\_prod(s\_int, s\_int), s\_bool)}$$

$$\overline{\vdash \ \geq \ \rhd \ s\_fct(s\_prod(s\_real, s\_real), s\_bool)}$$

$$\overline{\vdash \ + \ \rhd \ s\_fct(s\_prod(s\_int, s\_int), s\_int)}$$

$$\overline{\vdash \ + \ \rhd \ s\_fct(s\_prod(s\_real, s\_real), s\_real)}$$

$$\overline{\vdash \ - \ \rhd \ s\_fct(s\_prod(s\_int, s\_int), s\_int)}$$

$$\overline{\vdash \ - \ \rhd \ s\_fct(s\_prod(s\_real, s\_real), s\_real)}$$

$$\overline{\vdash \ * \ \rhd \ s\_fct(s\_prod(s\_int, s\_int), s\_int)}$$

$$\overline{\vdash \ * \ \rhd \ s\_fct(s\_prod(s\_real, s\_real), s\_real)}$$

$$\overline{\vdash \ / \ \rhd \ s\_fct(s\_prod(s\_int, s\_int), s\_int)}$$

$$\overline{\vdash \ \backslash \ \rhd \ s\_fct(s\_prod(s\_int, s\_int), s\_int)}$$

The types of the arguments must conform with the type of the infix operator.

$$\frac{\Sigma \vdash ve_1 \rhd t_1 \quad \Sigma \vdash ve_2 \rhd t_2 \quad \vdash op \rhd s\_fct(s\_prod(t_1, t_2), t_3)}{\Sigma \vdash ve_1 \ op \ ve_2 \rhd t_3}$$

Static Semantics of Axiom Infix Expressions

$$\frac{\Sigma \vdash ve_1 \rhd s\_bool \quad \Sigma \vdash ve_2 \rhd s\_bool}{\Sigma \vdash ve_1 \ conn \ ve_2 \rhd s\_bool}$$

Static Semantics of Value Prefix Expressions The type of the prefix operator **abs** is defined by the following rules:

$$\overline{\vdash \ \mathbf{abs} \ \rhd \ s\_fct(s\_int, s\_int)} \qquad \overline{\vdash \ \mathbf{abs} \ \rhd \ s\_fct(s\_real, s\_real)}$$

The type of the argument must conform with the type of the prefix operator.

$$\frac{\Sigma \vdash ve \rhd t_1 \quad \vdash op \rhd s\_fct(t_1, t_2)}{\Sigma \vdash op \ ve \rhd t_2}$$

Static Semantics of Axiom Prefix Expressions

$$\frac{\Sigma \vdash ve \rhd s\_bool}{\Sigma \vdash conn\ ve \rhd s\_bool}$$

Static Semantics of Structured Expressions

$$\frac{\Sigma \vdash ve_1 \rhd s\_bool \quad \Sigma \vdash ve_2 \rhd t \quad \Sigma \vdash ve_3 \rhd t}{\Sigma \vdash \textbf{if}\ ve_1\ \textbf{then}\ ve_2\ \textbf{else}\ ve_3\ \textbf{end} \rhd t}$$

4.2.3 Sentences

The sentences over a signature Σ are equivalence expressions that are well-formed with respect to Σ.

Definition 4.14 An *mRSL sentence* is an equivalence expression that is well-formed according to the static semantics in Section 4.2.1 and Section 4.2.2.

We let $Sen(\Sigma)$ denote the set of sentences over Σ.

Example 4.4 *The equivalence expressions:*

$(\exists\ \mathrm{x} : \textbf{Int} \bullet \mathrm{f(x)} \Rightarrow (\forall\ \mathrm{y} : \textbf{Int} \bullet \mathrm{f(y)})) \equiv \textbf{true}$
$\mathrm{f} \equiv (\lambda\ \mathrm{z} : \textbf{Int} \bullet 18 \leq \mathrm{z} \wedge \mathrm{z} < 70)$

are sentences over a signature $\langle S, A, \Psi, V\rangle$ for which $f \in V_{s\_fct(s\_int,s\_bool)}$, and $x \notin V$, $y \notin V$, and $z \notin V$.

Sentences and mRSL Specifications

The sentences of an mRSL specification stem from the axiom declarations.

Example 4.5 *The specification in Example 4.1 contained the axiom declaration:*

axiom [black_not_white] (black $\neq$ white) $\equiv$ **true**

with the sentence:

(black $\neq$ white) $\equiv$ **true**

Example 4.6 *The specification in Example 4.2 contained the axiom declarations:*

axiom [black_def] black $\equiv$ **true**,
[white_def] white $\equiv$ **false**

with the sentences:

black $\equiv$ **true**
white $\equiv$ **false**

4.2.4 Sentence Morphisms

Signature morphisms $\sigma = \langle \sigma_S, \sigma_A, \sigma_V \rangle$ are lifted to work on sentences by replacing occurrences of sort names, abbreviation types, and value names as prescribed by σ_S, σ_A, and σ_V, respectively.

Lifting signature morphisms to work on type expressions is considered first.

Definition 4.15 Let $\Sigma = \langle S, A, \Psi, V \rangle$ and $\Sigma' = \langle S', A', \Psi', V' \rangle$ be mRSL signatures, and let $\sigma = \langle \sigma_S, \sigma_A, \sigma_V \rangle$ be an mRSL signature morphism from Σ to Σ'. Then $Sen(\sigma) : TypeExpr(\Sigma) \longrightarrow TypeExpr(\Sigma')$ maps type expressions recursively as follows:

$$
\begin{array}{lcl}
Sen(\sigma)(\mathbf{Bool}) & = & \mathbf{Bool} \\
Sen(\sigma)(\mathbf{Int}) & = & \mathbf{Int} \\
Sen(\sigma)(\mathbf{Real}) & = & \mathbf{Real} \\
Sen(\sigma)(\mathbf{Unit}) & = & \mathbf{Unit} \\
Sen(\sigma)(typeexpr_1 \times typeexpr_2) & = & Sen(\sigma)(typeexpr_1) \times Sen(\sigma)(typeexpr_2) \\
Sen(\sigma)(typeexpr_1 \overset{\sim}{\to} typeexpr_2) & = & Sen(\sigma)(typeexpr_1) \overset{\sim}{\to} Sen(\sigma)(typeexpr_2) \\
Sen(\sigma)(type\_id) & = & \left\{ \begin{array}{ll} \sigma_S(type\_id) & \text{if } type\_id \in S \\ \sigma_A(type\_id) & \text{if } type\_id \in A \end{array} \right. \\
Sen(\sigma)((typeexpr)) & = & (Sen(\sigma)(typeexpr)).
\end{array}
$$

Then signature morphisms are lifted to work on value expressions.

Definition 4.16 Let $\Sigma = \langle S, A, \Psi, V \rangle$ and $\Sigma' = \langle S', A', \Psi', V' \rangle$ be mRSL signatures, and let $\sigma = \langle \sigma_S, \sigma_A, \sigma_V \rangle$ be an mRSL signature morphism from Σ to Σ'. Then $Sen(\Sigma)$ is the function $ValExpr(\Sigma) \longrightarrow ValExpr(\Sigma')$ given by:

$Sen(\sigma)(\mathbf{true}) = \mathbf{true}$

$Sen(\sigma)(\mathbf{false}) = \mathbf{false}$

$Sen(\sigma)(i) = i$

$Sen(\sigma)(r) = r$

$Sen(\sigma)(()) = ()$

$Sen(\sigma)(v) = \sigma_{V_t}(v)$ for $v \in V_t$

$Sen(\sigma)(\mathbf{chaos}) = \mathbf{chaos}$

$Sen(\sigma)(\mathbf{skip}) = \mathbf{skip}$

$Sen(\sigma)((expr_1, expr_2)) = (Sen(\sigma)(expr_1), Sen(\sigma)(expr_2))$

$Sen(\sigma)(\lambda\ x : te \bullet expr\ ) = \lambda\ x : Sen(\sigma)(te) \bullet Sen(\dot{\sigma})(expr)$

where $\dot{\sigma}$ is the signature morphism that maps x to x and maps as σ otherwise. Note that $x \notin \Sigma$ due to the restrictions on value expressions imposed by the static semantics and that $x \notin \Sigma'$ due to the assumption of disjoint name spaces.

$Sen(\sigma)(ve\ afp) \ = \ Sen(\sigma)(ve)\ Sen(\sigma)(afp)$

$Sen(\sigma)(\forall\ x : te \bullet expr) \ = \ \forall\ x : Sen(\sigma)(te) \bullet Sen(\dot{\sigma})(expr)$
where $\dot{\sigma}$ is the signature morphism that maps x to x and maps as σ otherwise.

$Sen(\sigma)(\exists\ x : te \bullet expr) \ = \ \exists\ x : Sen(\sigma)(te) \bullet Sen(\dot{\sigma})(expr)$
where $\dot{\sigma}$ is the signature morphism that maps x to x and maps as σ otherwise.

$Sen(\sigma)(expr_1 \equiv expr_2) \ = \ Sen(\sigma)(expr_1) \equiv Sen(\sigma)(expr_2)$

$Sen(\sigma)((expr)) \ = \ (Sen(\sigma)(expr))$

$Sen(\sigma)(expr_1\ op\ expr_2) \ = \ Sen(\sigma)(expr_1)\ op\ Sen(\sigma)(expr_2)$

$Sen(\sigma)(expr_1\ conn\ expr_2) \ = \ Sen(\sigma)(expr_1)\ conn\ Sen(\sigma)(expr_2)$

$Sen(\sigma)(op\ expr) \ = \ op\ Sen(\sigma)(expr)$

$Sen(\sigma)(conn\ expr) \ = \ conn\ Sen(\sigma)(expr)$

$Sen(\sigma)(\textbf{if}\ expr_1\ \textbf{then}\ expr_2\ \textbf{else}\ expr_3\ \textbf{end}) \ =$
$\textbf{if}\ Sen(\sigma)(expr_1)\ \textbf{then}\ Sen(\sigma)(expr_2)\ \textbf{else}\ Sen(\sigma)(expr_3)\ \textbf{end}.$

4.2.5 The Functor Sen_{mRSL}

The functor $Sen_{mRSL} : Sign_{mRSL} \longrightarrow Set$ maps each signature to the set of sentences over the signature, and lifts each signature morphism to work on sentences.

Definition 4.17 $Sen_{mRSL} : Sign_{mRSL} \longrightarrow Set$ is the functor that maps each signature Σ in $Sign_{mRSL}$ to the set $Sen(\Sigma)$ of sentences over Σ, and maps each signature morphism $\sigma : \Sigma \longrightarrow \Sigma'$ in $Sign_{mRSL}$ to $Sen(\sigma) : Sen(\Sigma) \longrightarrow Sen(\Sigma')$ which is the function defined in Definition 4.16.

Proposition 4.18 *For all signature morphisms $\sigma : \Sigma \longrightarrow \Sigma'$ and $\sigma' : \Sigma' \longrightarrow \Sigma''$, we have:*

$$Sen(\sigma' \circ \sigma) = Sen(\sigma') \circ Sen(\sigma)$$

where the latter composition is function composition in Set.

Proof. The proof is by induction on the structure of value expressions.

From Definition 4.16, it is clear that value literals and basic expressions are mapped identically by $Sen(\sigma' \circ \sigma)$ and $Sen(\sigma') \circ Sen(\sigma)$.

For a value name $v \in V$, we have that $Sen(\sigma' \circ \sigma)(v) = (\sigma' \circ \sigma)_V(v) = \sigma'_{V'}(\sigma_V(v)) = \sigma'_{V'}(Sen(\sigma)(v)) = Sen(\sigma')(Sen(\sigma)(v)) = (Sen(\sigma') \circ Sen(\sigma))(v)$.

Function expressions and quantified expressions contain typings. Function expressions are considered:

$$Sen(\sigma' \circ \sigma)(\lambda\ x : te \bullet expr)$$

$= \lambda\ x : Sen(\sigma' \circ \sigma)(te) \bullet Sen(\sigma' \dot{\circ} \sigma)(expr)$

$= \lambda\ x : Sen(\sigma')(Sen(\sigma)(te)) \bullet Sen(\dot{\sigma}')(Sen(\dot{\sigma})(expr))$

$= Sen(\sigma')(\lambda\ x : Sen(\sigma)(te) \bullet Sen(\dot{\sigma})(expr))$

$= Sen(\sigma')(Sen(\sigma)(\lambda\ x : te \bullet expr))$

$= (Sen(\sigma') \circ Sen(\sigma))(\lambda\ x : te \bullet expr).$

Likewise for quantified expressions.

For composite value expressions of the general form $\mathcal{F}(expr_1, \ldots, expr_n)$:

$Sen(\sigma' \circ \sigma)(\mathcal{F}(expr_1, \ldots, expr_n))$

$= \mathcal{F}(Sen(\sigma' \circ \sigma)(expr_1), \ldots, Sen(\sigma' \circ \sigma)(expr_n))$ by Definition 4.16

$= \mathcal{F}((Sen(\sigma') \circ Sen(\sigma))(expr_1), \ldots, (Sen(\sigma') \circ Sen(\sigma))(expr_n))$ by the induction hypothesis

$= \mathcal{F}(Sen(\sigma')(Sen(\sigma)(expr_1)), \ldots, Sen(\sigma')(Sen(\sigma)(expr_n)))$

$= Sen(\sigma')(\mathcal{F}(Sen(\sigma)(expr_1), \ldots, Sen(\sigma)(expr_n)))$ by Definition 4.16

$= Sen(\sigma')(Sen(\sigma)(\mathcal{F}(expr_1, \ldots, expr_n)))$ by Definition 4.16

$= (Sen(\sigma') \circ Sen(\sigma))(\mathcal{F}(expr_1, \ldots, expr_n)).$

Consequently, $Sen(\sigma' \circ \sigma) = Sen(\sigma') \circ Sen(\sigma)$. □

The following lemma is used for proving the next proposition.

Lemma 4.19 *For all signatures Σ and all type expressions te, we have:*

$Sen(id_\Sigma)(te) = te.$

Proof. By induction on the structure of type expressions.

From Definition 4.15, it is clear that $Sen(id_\Sigma)(te) = te$ when te is a type literal.

For composite type expressions of the general form $\mathcal{T}(te_1, \ldots, te_n)$:

$Sen(id_\Sigma)(\mathcal{T}(te_1, \ldots, te_n))$

$= \mathcal{T}(Sen(id_\Sigma)(te_1), \ldots, Sen(id_\Sigma)(te_n))$ by Definition 4.15

$= \mathcal{T}(te_1, \ldots, te_n)$ by the induction hypothesis.

In mRSL, only the composite type expressions $te_1 \times te_2$ and $te_1 \overset{\sim}{\rightarrow} te_2$ are considered.

A type name is either a sort type name $s \in S$ or an abbreviation type name $a \in A$. From Definition 4.15, it follows that $Sen(id_\Sigma)(s) = id_S(s) = s$ for $s \in S$, and $Sen(id_\Sigma)(a) = id_A(a) = a$ for $a \in A$. □

Proposition 4.20 *For all signatures Σ we have:*

$$Sen(id_\Sigma) = id_{Sen(\Sigma)}.$$

Proof. The proof is by induction on the structure of value expressions.

By definition, $Sen(id_\Sigma)$ maps value literals and basic expressions as the identity on $Sen(\Sigma)$.

Let $v \in V$ be a value name. By Definition 4.9 and Definition 4.16, $Sen(id_\Sigma)(v) = (id_\Sigma)_V(v) = id_V(v) = v$.

Function expressions and quantified expressions contain typings. Function expressions are considered:

$$Sen(id_\Sigma)(\lambda\ x : te \bullet expr)$$

$$= \lambda\ x : Sen(id_\Sigma)(te) \bullet Sen(\dot{id_\Sigma})(expr) \text{ by Definition 4.16}$$

$$= \lambda\ x : te \bullet Sen(\dot{id_\Sigma})(expr) \text{ by Lemma 4.19}$$

$$= \lambda\ x : te \bullet expr \text{ by the induction hypothesis.}$$

Likewise for quantified expressions.

For composite value expressions of the general form $\mathcal{F}(expr_1, \ldots, expr_n)$:

$$Sen(id_\Sigma)(\mathcal{F}(expr_1, \ldots, expr_n))$$

$$= \mathcal{F}(Sen(id_\Sigma)(expr_1), \ldots, Sen(id_\Sigma)(expr_n)) \text{ by Definition 4.16}$$

$$= \mathcal{F}(expr_1, \ldots, expr_n) \text{ by the induction hypothesis.}$$

$\square$

Proposition 4.18 and Proposition 4.20 show that Sen_{mRSL}, as defined in Definition 4.17, is a functor.

4.3 Models

A model over a signature Σ interprets the entities in Σ, i.e., the type names and the value names. The interpretation makes use of *semantic domains* that stem from the original RSL semantics.

In the following, we present semantic domains and the operators used for describing the semantics of value expressions. Then we define mRSL models, categories of models, and functors between them.

4.3.1 Semantic Domains

A type is a set of values, also known as a *value domain*. The notation $Value_t$ denotes the value domain for the static type t.

The collection of value domains for the types built from the basic types (**Bool**, **Int**, **Real**, **Unit**) and the type operators $\times$ and $\stackrel{\sim}{\rightarrow}$ is:

$$Types = \{Value_t \mid t \in T(\emptyset)\}.$$

For each type t, there is a *process domain* based on the value domain for t:

$$Procs_t = Value_t \cup \{\perp\}.$$

This describes that a process either terminates immediately with a deterministic result or diverges.

The collection of process domains for the types built from basic types and type operators is:

$$Processes = \{Procs_t \mid t \in T(\emptyset)\}.$$

The value domains for the basic types are:

- $Value_{s\_bool} = \{tt, ff\}$
- $Value_{s\_int} = \mathbb{Z}$
- $Value_{s\_real} = \mathbb{R}$
- $Value_{s\_unit} = \{d\_unit\}$.

Value domains for function types and product types are built from value domains of the constituent types.

The value domain for the product $s\_prod(t_1, t_2)$ is the Cartesian product of $Value_{t_1}$ and $Value_{t_2}$:

$$Value_{s\_prod(t_1,t_2)} = Value_{t_1} \times Value_{t_2}.$$

The value domain for the function type $s\_fct(t_1, t_2)$, denoted by $Value_{s\_fct(t_1,t_2)}$, is the set of all functions from the value domain $Value_{t_1}$ into the process domain $Procs_{t_2}$:

$$Value_{s\_fct(t_1,t_2)} = Value_{t_1} \longrightarrow Procs_{t_2}.$$

4.3.2 Semantic Operators

To express the semantics of value expressions, semantic operators are introduced. The operators are originally from [43], but have been adapted for the simplified setting in mRSL.

Piping

The semantics of a value expression is a process that either terminates with a result or diverges. The semantic operator *pipe* takes a process and a function as arguments. If the process terminates with a result, the result is passed on to the function; but if the process diverges (is chaos), the entire application of *pipe* also yields chaos.

Definition 4.21 Let t_1 and t_2 be static types, let $x \in Procs_{t_1}$, and let f be a function $Value_{t_1} \longrightarrow Procs_{t_2}$. The semantic operator *pipe* is defined by:

$$pipe : Procs_{t_1} \longrightarrow (Value_{t_1} \longrightarrow Procs_{t_2}) \longrightarrow Procs_{t_2}$$

$$pipe\, x\, f = \begin{cases} \perp & \text{if } x = \perp \\ f(x) & \text{otherwise.} \end{cases}$$

Infix notation may be used for *pipe* such that $pipe\ x\ f$ is written $x\ pipe\ f$.

From Values to Processes

Given a value, the operator $v2p$ yields the process that terminates immediately with the value as the result. The operator is overloaded such that it takes arguments of all value domains.

Definition 4.22 For $t \in T(\emptyset)$, the operator

$$v2p : Value_t \longrightarrow Procs_t$$

is defined by

$$v2p(x) = x.$$

In the defining equation, the x that occurs as argument of $v2p$ is a value $x \in Value_t$, and the x on the right-hand side is interpreted as the process in $Procs_t$ that terminates immediately with the result value x.

It may be convenient to lift a value domain to a process domain.

Definition 4.23 The function

$$Procs : Types \longrightarrow Processes$$

is defined by

$$Procs(Value_t) = Procs_t$$

for $t \in T(\emptyset)$.

4.3.3 Models

A model of a signature $\Sigma = \langle S, A, \Psi, V\rangle$ must interpret each sort type name in S, each abbreviation type name in A, and each value name in V. The interpretation of sort and abbreviation type names must be consistent with Ψ. This leads to the following definition of an mRSL model:

DEFINITION 4.24 Let $\Sigma = \langle S, A, \Psi, V\rangle$ be a signature. A Σ-*model* is a triple $\langle m_S, m_A, m_V\rangle$ with the components

- $m_S : S \longrightarrow Types$
- $m_A : A \longrightarrow Types$
- $m_V = (m_{V_t} : V_t \longrightarrow m_{T(S)}(t))_{t\in T(S)}$

where

- $m_{T(S)}$ is m_S lifted to interpret type expressions in $T(S)$:
  - $m_{T(S)}(t) = Value_t$ for $t \in \{s\_bool, s\_int, s\_real, s\_unit\}$
  - $m_{T(S)}(s) = m_S(s)$ for $s \in S$
  - $m_{T(S)}(s\_prod(t_1, t_2)) = m_{T(S)}(t_1) \times m_{T(S)}(t_2)$
  - $m_{T(S)}(s\_fct(t_1, t_2)) = m_{T(S)}(t_1) \longrightarrow Procs(m_{T(S)}(t_2))$
- $m_A(a) = m_{T(S)}(\Psi^*(a))$ for $a \in A$.

The function m_S interprets the sort names in S as sets in $Types$. Static types that correspond to type literals and the type operators $\times$ and $\xrightarrow{\sim}$ have fixed interpretations used to lift m_S to a map $m_{T(S)}$ which interprets static types containing names of sorts in S.

Static types that correspond to type literals are always interpreted as their respective value domains. Static types that correspond to composite types formed using the type operators $\times$ and $\xrightarrow{\sim}$ are interpreted as the corresponding value domains built from the value domains of the constituent types as described in Section 4.3.1.

The function m_A interprets the abbreviation types in A such that $m_A(a) = m_{T(S)}(\Psi^*(a))$ for $a \in A$. Since $a \in A$ may be seen as a static type, Ψ^* may be applied to a.

Essentially, m_A is redundant since it is fully defined by m_S and Ψ. It is included to make it clear how names of abbreviation types are interpreted.

The $T(S)$-sorted function m_V interprets value names in V as elements of sets in $Types$, i.e., value names in V_t are interpreted as elements of $m_{T(S)}(t)$ which is the interpretation of t.

EXAMPLE 4.7 *Recall the signature of the specification* C2 *in Example 4.2:* $Sig(C2) = \langle\{\text{Intensity}\}, \{\text{Colour}\}, \Psi, V\rangle$ *where* $\Psi(\text{Colour}) = s\_bool$, $V_{s\_bool} = \{\text{black}, \text{white}\}$, *and* $V_t = \emptyset$ *for* $t \in T(\{\text{Intensity}\})$ *and* $t \neq s\_bool$.

A model m *of* $Sig(C2)$ *may be defined as follows:*

- $m_S(\text{Intensity}) = Value_{s\_int}$, *i.e., the sort Intensity is interpreted as the type for integers*
- $m_A(\text{Colour}) = Value_{s\_bool}$ *as prescribed by* Ψ
- $m_V(\text{black}) = tt$ *and* $m_V(\text{white}) = ff$.

Another $Sig(C2)$*-model* n *may be defined as follows:*

- $n_S(\text{Intensity}) = Value_{s\_bool}$, *i.e., the sort Intensity is interpreted as the type for Booleans*
- $n_A(\text{Colour}) = Value_{s\_bool}$ *as prescribed by* Ψ
- $n_V(\text{black}) = tt$ *and* $n_V(\text{white}) = tt$.

4.3.4 Categories of Models

For each signature, there is a collection of models of the signature. These collections are categories.

Definition 4.25 For a signature Σ, let $Mod(\Sigma)$ denote the discrete category of Σ-models. The objects of $Mod(\Sigma)$ are all the models over Σ, and the morphisms of $Mod(\Sigma)$ are the identities on the models.

4.3.5 The Reduct Functor

In the following, let $\Sigma = \langle S, A, \Psi, V\rangle$ and $\Sigma' = \langle S', A', \Psi', V'\rangle$ be signatures, let $\sigma : \Sigma \longrightarrow \Sigma'$ be a signature morphism $\sigma = \langle \sigma_S, \sigma_A, \sigma_V\rangle$, and let $m' = \langle m'_{S'}, m'_{A'}, m'_{V'}\rangle$ be a Σ'-model.

Definition 4.26 The *σ-reduct* $m'|_\sigma$ of the Σ'-model m' is the Σ-model $m = \langle m_S, m_A, m_V\rangle$, defined by:

- $m_S(s) = m'_{T(S')}(\Psi'^*(\sigma_S(s)))$ for all $s \in S$ (the result of $\sigma_S(s)$ is in $S' \cup A'$ and is seen as a static type in $T(S' \cup A')$)
- $m_A(a) = m'_{A'}(\sigma_A(a))$ for all $a \in A$
- $m_V = (m'_{V'_{\Psi'^*(T(\sigma_S)(t))}} \circ \sigma_{V_t} : V_t \longrightarrow m'_{T(S')}(\Psi'^*(T(\sigma_S)(t))))_{t\in T(S)}$.

Example 4.8 *Recall the model* m *defined in Example 4.7 and let* σ *be the signature morphism defined in Example 4.3. The* σ*-reduct of* m *is the model* $m|_\sigma$ *defined by:*

- $(m|_\sigma)_S(\text{Colour}) = m_{T(S)}(\Psi^*(\sigma(\text{Colour}))) = m_{T(S)}(\Psi^*(\text{Colour}))$
  $= m_{T(S)}(s\_bool) = Value_{s\_bool}$
- $(m|_\sigma)_A$ *is the empty function*

- $(m|_\sigma)_V(\text{black}) = m_V(\sigma(\text{black})) = tt$ *and*
  $(m|_\sigma)_V(\text{white}) = m_V(\sigma(\text{white})) = ff$.

Given a signature morphism, we can define a functor.

Definition 4.27 The *reduct functor* $Mod(\sigma) : Mod(\Sigma') \longrightarrow Mod(\Sigma)$ maps models in $Mod(\Sigma')$ to their σ-reducts in $Mod(\Sigma)$ and morphisms in $Mod(\Sigma')$ to morphisms in $Mod(\Sigma)$ such that $Mod(\sigma)(id_{m'}) = id_{Mod(\sigma)(m')}$ for $m' \in |Mod(\Sigma')|$.

The only morphisms in $Mod(\Sigma)$ and $Mod(\Sigma')$ are the identities on models. As a consequence, the only way to map a morphism $id_{m'} : m' \longrightarrow m'$ to a morphism $Mod(\sigma)(m') \longrightarrow Mod(\sigma)(m')$ is $Mod(\sigma)(id_{m'}) = id_{Mod(\sigma)(m')}$ as done in the above definition.

Since the considered categories of models are discrete categories, the following proposition states that $Mod(\sigma)$ is a functor.

Proposition 4.28 *For all signatures Σ and Σ', signature morphisms $\sigma : \Sigma \longrightarrow \Sigma'$, and models m' in $Mod(\Sigma')$:*

- $Mod(\sigma)(id_{m'} \circ id_{m'}) = Mod(\sigma)(id_{m'}) \circ Mod(\sigma)(id_{m'})$
- $Mod(\sigma)(id_m) = id_{Mod(\sigma)(m)}$.

Proof. The first property follows from properties of identities and Definition 4.27:

$$
\begin{aligned}
& Mod(\sigma)(id_{m'} \circ id_{m'}) \\
= {} & Mod(\sigma)(id_{m'}) \\
= {} & Mod(\sigma)(id_{m'}) \circ id_{Mod(\sigma)(m')} \\
= {} & Mod(\sigma)(id_{m'}) \circ Mod(\sigma)(id_{m'}).
\end{aligned}
$$

The second property follows directly from Definition 4.27. □

The notation $Mod(\sigma)(m')$ is often used interchangeably with $m'|_\sigma$.

4.3.6 The Functor Mod_{mRSL}

In the previous section, we defined a category $Mod(\Sigma)$ of models for each signature Σ and a functor $Mod(\sigma) : Mod(\Sigma') \longrightarrow Mod(\Sigma)$ for each signature morphism $\sigma : \Sigma \longrightarrow \Sigma'$. This constitutes a functor.

Definition 4.29 $Mod_{mRSL} : Sign_{mRSL} \longrightarrow Cat^{op}$ is the functor that maps each signature Σ in $Sign_{mRSL}$ to the discrete category $Mod(\Sigma)$ of models over the signature, and maps each signature morphism $\sigma : \Sigma \longrightarrow \Sigma'$ in $Sign_{mRSL}$ to the reduct functor $Mod(\sigma) : Mod(\Sigma') \longrightarrow Mod(\Sigma)$ between the categories $Mod(\Sigma')$ and $Mod(\Sigma)$ of models.

Before we prove that Mod_{mRSL} is a functor, we introduce some helpful propositions.

Proposition 4.30 *For all signature morphisms $\sigma = \langle \sigma_S, \sigma_A, \sigma_V \rangle$ from a signature $\Sigma = \langle S, A, \Psi, V \rangle$ to a signature $\Sigma' = \langle S', A', \Psi', V' \rangle$, all Σ'-models m', and all $t \in T(S)$:*

$$(Mod(\sigma)(m'))_{T(S')}(t) = m'_{T(S')}(\Psi'^*(T(\sigma_S)(t))).$$

Proof. By induction over the structure of static types $t \in T(S)$.
For every built-in type $t \in \{s\_bool, s\_int, s\_real, s\_unit\}$:

$$\begin{aligned}
& (Mod(\sigma)(m'))_{T(S')}(t) \\
= \ & Value_t \\
= \ & m'_{T(S')}(\Psi'^*(T(\sigma_S)(t))).
\end{aligned}$$

For sort type names $s \in S$:

$$\begin{aligned}
& (Mod(\sigma)(m'))_{T(S')}(s) \\
= \ & (Mod(\sigma)(m'))_S(s) \\
= \ & m'_{T(S')}(\Psi'^*(\sigma_S(s))) \\
= \ & m'_{T(S')}(\Psi'^*(T(\sigma_S)(s))).
\end{aligned}$$

For Cartesian products $s\_prod(t_1, t_2)$:

$$\begin{aligned}
& (Mod(\sigma)(m'))_{T(S')}(s_prod(t_1, t_2)) \\
= \ & (Mod(\sigma)(m'))_{T(S')}(t_1) \times (Mod(\sigma)(m'))_{T(S')}(t_2) \\
= \ & m'_{T(S')}(\Psi'^*(T(\sigma_S)(t_1))) \times m'_{T(S')}(\Psi'^*(T(\sigma_S)(t_1))) \text{ by the induction hypothesis} \\
= \ & m'_{T(S')}(s_prod(\Psi'^*(T(\sigma_S)(t_1)), \Psi'^*(T(\sigma_S)(t_1)))) \\
= \ & m'_{T(S')}(\Psi'^*(s_prod(T(\sigma_S)(t_1), T(\sigma_S)(t_2)))) \\
= \ & m'_{T(S')}(\Psi'^*(T(\sigma_S)(s_prod(t_1, t_2)))).
\end{aligned}$$

For function types $s\_fct(t_1, t_2)$:

$$\begin{aligned}
& (Mod(\sigma)(m'))_{T(S')}(s_fct(t_1, t_2)) \\
= \ & (Mod(\sigma)(m'))_{T(S')}(t_1) \longrightarrow Procs((Mod(\sigma)(m'))_{T(S')}(t_2)) \\
= \ & m'_{T(S')}(\Psi'^*(T(\sigma_S)(t_1))) \longrightarrow Procs(m'_{T(S')}(\Psi'^*(T(\sigma_S)(t_2)))) \text{ by the induction hypothesis}
\end{aligned}$$

$$= m'_{T(S')}(s\_fct(\Psi'^*(T(\sigma_S)(t_1)), \Psi'^*(T(\sigma_S)(t_2))))$$

$$= m'_{T(S')}(\Psi'^*(s\_fct(T(\sigma_S)(t_1), T(\sigma_S)(t_2))))$$

$$= m'_{T(S')}(\Psi'^*(T(\sigma_S)(s\_fct(t_1, t_2)))).$$

□

PROPOSITION 4.31 *For all signature morphisms $\sigma = \langle \sigma_S, \sigma_A, \sigma_V \rangle$ from $\Sigma = \langle S, A, \Psi, V \rangle$ to $\Sigma' = \langle S', A', \Psi', V' \rangle$ and $\sigma' = \langle \sigma'_{S'}, \sigma'_{A'}, \sigma'_{V'} \rangle$ from $\Sigma' = \langle S', A', \Psi', V' \rangle$ to $\Sigma'' = \langle S'', A'', \Psi'', V'' \rangle$, and all Σ''-models $m'' = \langle m''_{S''}, m''_{A''}, m''_{V''} \rangle$, we have:*

$$(Mod(\sigma' \circ \sigma)(m''))_S = ((Mod(\sigma) \circ Mod(\sigma'))(m''))_S.$$

PROOF. For $s \in S$, consider the left-hand side:

$$(Mod(\sigma' \circ \sigma)(m''))_S(s)$$

$$= m''_{T(S'')}(\Psi''^*((\sigma' \circ \sigma)_S(s))) \text{ by Definition 4.26}$$

$$= \begin{cases} m''_{T(S'')}(\Psi''^*(\sigma'_{S'}(\sigma_S(s)))) & \text{if } \sigma_S(s) \in S' \\ m''_{T(S'')}(\Psi''^*(\sigma'_{A'}(\sigma_S(s)))) & \text{if } \sigma_S(s) \in A'. \end{cases}.$$

Then consider the right-hand side:

$$((Mod(\sigma) \circ Mod(\sigma'))(m''))_S(s)$$

$$= (Mod(\sigma)(Mod(\sigma')(m'')))_S(s) \text{ by composition of functors}$$

$$= (Mod(\sigma')(m''))_{T(S')}(\Psi'^*(\sigma_S(s))) \text{ by Definition 4.26}$$

$$= m''_{T(S')}(\Psi''^*(T(\sigma'_{S'})(\Psi'^*(\sigma_S(s))))) \text{ by Proposition 4.30}$$

$$= \begin{cases} m''_{T(S'')}(\Psi''^*(T(\sigma'_{S'})(\sigma_S(s)))) & \text{if } \sigma_S(s) \in S' \\ m''_{T(S'')}(\Psi''^*(T(\sigma'_{S'})(\Psi'^*(\sigma_S(s))))) & \text{if } \sigma_S(s) \in A' \end{cases}$$

$$= \begin{cases} m''_{T(S'')}(\Psi''^*(\sigma'_{S'}(\sigma_S(s)))) & \text{if } \sigma_S(s) \in S' \\ m''_{T(S'')}(\Psi''^*(T(\sigma'_{S'})(\Psi'^*(\sigma_S(s))))) & \text{if } \sigma_S(s) \in A' \end{cases}$$

$$= \begin{cases} m''_{T(S'')}(\Psi''^*(\sigma'_{S'}(\sigma_S(s)))) & \text{if } \sigma_S(s) \in S' \\ m''_{T(S'')}(\Psi''^*(\sigma'_{A'}(\sigma_S(s)))) & \text{if } \sigma_S(s) \in A' \end{cases}.$$

Note that $\sigma_S(s) \in S'$ implies $\Psi'^*(\sigma_S(s)) = \sigma_S(s)$ according to Lemma 4.4, and that $\sigma_S(s) \in A'$ implies $\Psi''^*(T(\sigma'_{S'})(\Psi'^*(\sigma_S(s)))) = \Psi''^*(\sigma'_{A'}(\sigma_S(s)))$ according to Definition 4.5. □

Proposition 4.32 *For all signature morphisms $\sigma = \langle \sigma_S, \sigma_A, \sigma_V \rangle$ from $\Sigma = \langle S, A, \Psi, V \rangle$ to $\Sigma' = \langle S', A', \Psi', V' \rangle$ and $\sigma' = \langle \sigma'_{S'}, \sigma'_{A'}, \sigma'_{V'} \rangle$ from $\Sigma' = \langle S', A', \Psi', V' \rangle$ to $\Sigma'' = \langle S'', A'', \Psi'', V'' \rangle$, and all Σ''-models $m'' = \langle m''_{S''}, m''_{A''}, m''_{V''} \rangle$, we have:*

$$(Mod(\sigma' \circ \sigma)(m''))_A = ((Mod(\sigma) \circ Mod(\sigma'))(m''))_A.$$

Proof. For all $a \in A$:

$$\begin{aligned}
& (Mod(\sigma' \circ \sigma)(m''))_A(a) \\
= \ & m''_{A''}((\sigma' \circ \sigma)_A(a)) \text{ by Definition 4.26} \\
= \ & m''_{A''}(\sigma'_{A'}(\sigma_A(a))) \text{ by Definition 4.6} \\
= \ & (Mod(\sigma')(m''))_{A'}(\sigma_A(a)) \text{ by Definition 4.26} \\
= \ & (Mod(\sigma)(Mod(\sigma')(m'')))_A(a) \text{ by Definition 4.26} \\
= \ & ((Mod(\sigma) \circ Mod(\sigma'))(m''))_A(a) \text{ by composition of functors.}
\end{aligned}$$

□

To prove the next proposition, we introduce the following lemma:

Lemma 4.33 *For all signature morphisms $\sigma = \langle \sigma_S, \sigma_A, \sigma_V \rangle$ from $\Sigma = \langle S, A, \Psi, V \rangle$ to $\Sigma' = \langle S', A', \Psi', V' \rangle$ and $\sigma' = \langle \sigma'_{S'}, \sigma'_{A'}, \sigma'_{V'} \rangle$ from $\Sigma' = \langle S', A', \Psi', V' \rangle$ to $\Sigma'' = \langle S'', A'', \Psi'', V'' \rangle$, and all Σ''-models $m'' = \langle m''_{S''}, m''_{A''}, m''_{V''} \rangle$, we have:*

$$m''_{V''_{\Psi''^*(T((\sigma' \circ \sigma)_S)(t))}} \circ (\sigma'_{V'_{\Psi'^*(T(\sigma_S)(t))}} \circ \sigma_{V_t}) = (m''_{V''_{\Psi''^*(T((\sigma' \circ \sigma)_S)(t))}} \circ \sigma'_{V'_{\Psi'^*(T(\sigma_S)(t))}}) \circ \sigma_{V_t}.$$

Proof. For all $t \in T(S)$ and $v \in V_t$:

$$\begin{aligned}
& (m''_{V''_{\Psi''^*(T((\sigma' \circ \sigma)_S)(t))}} \circ (\sigma'_{V'_{\Psi'^*(T(\sigma_S)(t))}} \circ \sigma_{V_t}))(v) \\
= \ & m''_{V''_{\Psi''^*(T((\sigma' \circ \sigma)_S)(t))}}((\sigma'_{V'_{\Psi'^*(T(\sigma_S)(t))}} \circ \sigma_{V_t})(v)) \\
= \ & m''_{V''_{\Psi''^*(T((\sigma' \circ \sigma)_S)(t))}}(\sigma'_{V'_{\Psi'^*(T(\sigma_S)(t))}}(\sigma_{V_t}(v))) \\
= \ & (m''_{V''_{\Psi''^*(T((\sigma' \circ \sigma)_S)(t))}} \circ \sigma'_{V'_{\Psi'^*(T(\sigma_S)(t))}})(\sigma_{V_t}(v)) \\
= \ & ((m''_{V''_{\Psi''^*(T((\sigma' \circ \sigma)_S)(t))}} \circ \sigma'_{V'_{\Psi'^*(T(\sigma_S)(t))}}) \circ \sigma_{V_t})(v).
\end{aligned}$$

□

Proposition 4.34 *For all signature morphisms $\sigma = \langle \sigma_S, \sigma_A, \sigma_V \rangle$ from $\Sigma = \langle S, A, \Psi, V \rangle$ to $\Sigma' = \langle S', A', \Psi', V' \rangle$ and $\sigma' = \langle \sigma'_{S'}, \sigma'_{A'}, \sigma'_{V'} \rangle$ from $\Sigma' = \langle S', A', \Psi', V' \rangle$ to $\Sigma'' = \langle S'', A'', \Psi'', V'' \rangle$, and all Σ''-models $m'' = \langle m''_{S''}, m''_{A''}, m''_{V''} \rangle$, we have:*

$$(Mod(\sigma' \circ \sigma)(m''))_V = ((Mod(\sigma) \circ Mod(\sigma'))(m''))_V$$

Proof.

$$\begin{aligned}
& (Mod(\sigma' \circ \sigma)(m''))_V \\
= \; & (m''_{V''_{\Psi''^*(T((\sigma' \circ \sigma)_S)(t))}} \circ (\sigma' \circ \sigma)_{V_t} : V_t \longrightarrow m''_{T(S'')}(\Psi''^*(T((\sigma' \circ \sigma)_S)(t))))_{t \in T(S)} \\
= \; & (m''_{V''_{\Psi''^*(T((\sigma' \circ \sigma)_S)(t))}} \circ (\sigma'_{V'_{\Psi'^*(T(\sigma_S)(t))}} \circ \sigma_{V_t}) : \\
& \quad V_t \longrightarrow m''_{T(S'')}(\Psi''^*(T((\sigma' \circ \sigma)_S)(t))))_{t \in T(S)} \\
= \; & ((m''_{V''_{\Psi''^*(T((\sigma' \circ \sigma)_S)(t))}} \circ \sigma'_{V'_{\Psi'^*(T(\sigma_S)(t))}}) \circ \sigma_{V_t} : \\
& \quad V_t \longrightarrow m''_{T(S'')}(\Psi''^*(T((\sigma' \circ \sigma)_S)(t))))_{t \in T(S)} \\
= \; & ((m''_{V''_{\Psi''^*(T(\sigma'_{S'})(\Psi'^*(T(\sigma_S)(t))))}} \circ \sigma'_{V'_{\Psi'^*(T(\sigma_S)(t))}}) \circ \sigma_{V_t} : \\
& \quad V_t \longrightarrow m''_{T(S'')}(\Psi''^*(T(\sigma'_{S'})(\Psi'^*(T(\sigma_S)(t))))))_{t \in T(S)} \\
= \; & (Mod(\sigma)(Mod(\sigma')(m'')))_V \\
= \; & ((Mod(\sigma) \circ Mod(\sigma'))(m''))_V.
\end{aligned}$$

□

Proposition 4.35 *For all signature morphisms $\sigma : \Sigma \longrightarrow \Sigma'$ and $\sigma' : \Sigma' \longrightarrow \Sigma''$:*

$$Mod(\sigma' \circ \sigma) = Mod(\sigma) \circ Mod(\sigma').$$

Proof. Follows from Proposition 4.31, Proposition 4.32, and Proposition 4.34. □

The following lemma is used in the proof of the next proposition.

Lemma 4.36 *For all signatures $\Sigma = \langle S, A, \Psi, V \rangle$ and Σ-models $m = \langle m_S, m_A, m_V \rangle$, we have:*

$$(m_{T(S)} \circ \Psi^*) \circ id_S = m_S.$$

Proof. For $s \in S$:

$$\begin{aligned}
& ((m_{T(S)} \circ \Psi^*) \circ id_S)(s) \\
= \; & (m_{T(S)} \circ \Psi^*)(id_S(s))
\end{aligned}$$

$= (m_{T(S)} \circ \Psi^*)(s)$ since id_S is the identity on S

$= m_{T(S)}(\Psi^*(s))$

$= m_{T(S)}(s)$ by Lemma 4.4

$= m_S(s)$ by the definition of $m_{T(S)}$.

□

PROPOSITION 4.37 *For all signatures $\Sigma \in |Sign_{mRSL}|$:*

$$Mod(id_\Sigma) = id_{Mod(\Sigma)}.$$

PROOF. Let $id_\Sigma = \langle id_S, id_A, id_V \rangle$ be the identity on a signature Σ, and let $m = \langle m_S, m_A, m_V \rangle$ be a Σ-model. By Lemma 4.36 and Lemma 4.4 ($\Psi^*(t) = t$ for $t \in T(S)$):

$Mod(id_\Sigma)(m)$

$= \langle (m_{T(S)} \circ \Psi^*) \circ id_S,$
$m_A \circ id_A,$
$(m_{V_{\Psi^*(T(id_S)(t))}} \circ id_{V_t} : V_t \longrightarrow m_{T(S)}(\Psi^*(T(id_S)(t))))_{t \in T(S)} \rangle$

$= \langle m_S, m_A, (m_{V_{\Psi^*(t)}} : V_t \longrightarrow m_{T(S)}(\Psi^*(t)))_{t \in T(S)} \rangle$

$= \langle m_S, m_A, (m_{V_t} : V_t \longrightarrow m_{T(S)}(t))_{t \in T(S)} \rangle$

$= m.$

□

The following proposition states that Mod_{mRSL} is a functor:

PROPOSITION 4.38 *Mod_{mRSL} meets the requirements for a functor:*

- *$Mod_{mRSL}(\sigma' \circ \sigma) = Mod_{mRSL}(\sigma) \circ Mod_{mRSL}(\sigma')$ for $\sigma : \Sigma \longrightarrow \Sigma'$ and $\sigma' : \Sigma' \longrightarrow \Sigma''$ in $Sign_{mRSL}$*
- *$Mod_{mRSL}(id_\Sigma) = id_{Mod_{mRSL}(\Sigma)}$ for Σ in $Sign_{mRSL}$.*

PROOF. Follows from Proposition 4.35 and Proposition 4.37. □

4.4 Satisfaction Relation

The dynamic denotational semantics of mRSL is specified by the meaning function M. The meaning function is dependently typed:

$$M : \Pi\,\Sigma \in |Sign_{mRSL}| \bullet Mod(\Sigma) \longrightarrow ValExpr(\Sigma) \longrightarrow \bigcup_{t \in T(\emptyset)} Procs_t.$$

The application $M(\Sigma)(m)(ve)$ essentially evaluates a value expression ve, interpreting value names as in the model m. The result is a process; if $\Sigma \vdash ve \rhd t$, then the result type of $M(\Sigma)(m)(ve)$ corresponds to the meaning of t in the context of m. The meaning function M is defined for value expressions that conform with the static semantics, and it is used for expressing the satisfaction relation $\models$:

Definition 4.39 For each mRSL signature Σ, Σ-model m, and value expressions ve_1 and ve_2 over Σ:

$$m \models_\Sigma ve_1 \equiv ve_2 \qquad \text{iff} \qquad M(\Sigma)(m)(ve_1 \equiv ve_2) = v2p(tt).$$

In other words, m satisfies an equivalence expression if and only if the semantics of the constituent value expressions are equal.

The meaning function M will be defined in Section 4.4.2 in which the dynamic semantics of value expressions is considered. But first, the dynamic semantics of type expressions is considered.

4.4.1 Dynamic Semantics of Type Expressions

The meaning function M is overloaded such that it also assigns a meaning to type expressions.

The meaning function that deals with type expressions is also dependently typed:

$$M : \Pi\,\Sigma \in |Sign_{mRSL}| \bullet Mod(\Sigma) \longrightarrow TypeExpr(\Sigma) \longrightarrow Types.$$

The meaning function is defined by a set of inference rules. The premisses express that the type expression of discourse must conform with the static semantics, and the conclusions describe the meaning function.

$$\frac{\Sigma \vdash \mathbf{Bool} \rhd s\_bool}{M(\Sigma)(m)(\mathbf{Bool}) = Value_{s\_bool}} \qquad \frac{\Sigma \vdash \mathbf{Int} \rhd s\_int}{M(\Sigma)(m)(\mathbf{Int}) = Value_{s\_int}}$$

$$\frac{\Sigma \vdash \mathbf{Real} \rhd s\_real}{M(\Sigma)(m)(\mathbf{Real}) = Value_{s\_real}} \qquad \frac{\Sigma \vdash \mathbf{Unit} \rhd s\_unit}{M(\Sigma)(m)(\mathbf{Unit}) = Value_{s\_unit}}$$

$$\frac{\Sigma = \langle S, A, \Psi, V\rangle \quad s \in S \quad \Sigma \vdash s \rhd s}{M(\Sigma)(m)(s) = m_S(s)}$$

$$\frac{\Sigma = \langle S, A, \Psi, V\rangle \quad a \in A \quad \Psi^*(a) = t \quad \Sigma \vdash a \rhd t}{M(\Sigma)(m)(a) = m_A(a)}$$

$$\frac{\Sigma \vdash te_1 \times te_2 \rhd s\_prod(t_1, t_2)}{M(\Sigma)(m)(te_1 \times te_2) = M(\Sigma)(m)(te_1) \times M(\Sigma)(m)(te_2)}$$

$$\frac{\Sigma \vdash te_1 \stackrel{\sim}{\rightarrow} te_2 \rhd s\_fct(t_1, t_2)}{M(\Sigma)(m)(te_1 \stackrel{\sim}{\rightarrow} te_2) = M(\Sigma)(m)(te_1) \longrightarrow Procs(M(\Sigma)(m)(te_2))}$$

where $Procs$ is the function from Definition 4.23 which lifts a value domain to its corresponding process domain: $Procs(Value_t) = Procs_t$ for $t \in T(\emptyset)$.

4.4.2 Dynamic Semantics of Value Expressions

The meaning function that assigns a meaning to value expressions is also defined in a set of inference rules.

Before presenting the inference rules, we introduce the notions of *model fragments* and *extension of models* which correspond to the notions of signature fragments and extension of signatures (Definition 4.12 and Definition 4.13).

Definition 4.40 Let $\Sigma = \langle S, A, \Psi, V\rangle$ be a signature, let $\Delta = (x, st)$ be a Σ-signature fragment, and let $m = \langle m_S, m_A, m_V\rangle$ be a Σ-model. A *Δ-fragment of m* is a pair (x, dv) where dv is an element of $m_{T(S)}(st)$.

We say that (x, dv) is the model fragment that interprets x as dv.

Definition 4.41 Let $\Sigma = \langle S, A, \Psi, V\rangle$ be a signature, let $\Delta = (x, st)$ be a Σ-signature fragment, let $m = \langle m_S, m_A, m_V\rangle$ be a Σ-model, and let (x, dv) be a Δ-fragment of m. Then $m \dagger (x, dv) = \langle m_S, m_A, m_{V^\dagger}\rangle$ where $m_{V^\dagger} = (m_{V^\dagger{}_t})_{t \in T(S)}$ such that $m_{V^\dagger{}_{st}}(x) = dv$, and $m_{V^\dagger{}_t}(v) = m_{V_t}(v)$ for $t \in T(S)$, $v \neq x$, and $v \in V_t$.

We say that $m \dagger (x, dv)$ is the model m *extended with* (x, dv).

In the description of the dynamic semantics of value expressions, we also need to be able to express the dynamic semantics of static types. This is defined such that it conforms with the dynamic semantics of type expressions given in Section 4.4.1. Consequently, the meaning function is further overloaded (it is assumed that $\Sigma = \langle S, A, \Psi, V\rangle$):

$$M : \Pi\, \Sigma \in |Sign_{mRSL}| \bullet Mod(\Sigma) \longrightarrow T(S \cup A) \longrightarrow Types.$$

For $te \in TypeExpr(\Sigma)$ and $t \in T(S \cup A)$:

if $\Sigma \vdash te \rhd t$, then $M(\Sigma)(m)(t) = M(\Sigma)(m)(te)$

for all Σ-models m.

The inference rules that lay down the dynamic semantics of value expressions are now to be presented. The meta-language used to express the semantics resembles that of RSL itself with let-expressions, set expressions, function applications, operators for comparison, etc. We shall not provide a formal semantics for this meta-language but rely on an intuitive understanding of it.

In the conclusions of the inference rules, the symbol $=$ is used in different ways: (i) for defining M, (ii) for binding values to new identifiers in let-expressions, and (iii) as a comparison operator.

For the sake of simplicity, we let the comparison operators $=$ and $\neq$ of the meta-language return values in $Value_{s\_bool} = \{tt, ff\}$ such that the result of a comparison can be used directly to describe an element in $Value_{s\_bool}$.

Dynamic Semantics of Value Literals

Value literals of the built-in types are given semantics which is independent of the signatures and models of discourse.

$$\frac{\Sigma \vdash \textbf{true} \rhd s\_bool}{M(\Sigma)(m)(\textbf{true}) = v2p(tt)} \qquad \frac{\Sigma \vdash \textbf{false} \rhd s\_bool}{M(\Sigma)(m)(\textbf{false}) = v2p(ff)}$$

$$\frac{\Sigma \vdash i \rhd s\_int}{M(\Sigma)(m)(i) = v2p(i)} \qquad \frac{\Sigma \vdash r \rhd s\_real}{M(\Sigma)(m)(r) = v2p(r)}$$

$$\frac{\Sigma \vdash () \rhd s\_unit}{M(\Sigma)(m)(()) = v2p(d\_unit)}$$

Dynamic Semantics of Value Names

References to user-declared values are given semantics according to the context model.

$$\frac{\Sigma \vdash v \rhd t}{M(\Sigma)(m)(v) = v2p(m_{V_t}(v))}$$

Dynamic Semantics of Basic Expressions

Basic expressions are given semantics independent of the signatures and models of discourse.

$$\frac{\Sigma \vdash \textbf{chaos} \rhd t}{M(\Sigma)(m)(\textbf{chaos}) = \bot} \qquad \frac{\Sigma \vdash \textbf{skip} \rhd s\_unit}{M(\Sigma)(m)(\textbf{skip}) = v2p(d\_unit)}$$

Dynamic Semantics of Product Expressions

Product expressions are given semantics that corresponds to evaluating the components from left to right.

$$\frac{\Sigma \vdash (ve_1{,}ve_2) \rhd s\_prod(t_1, t_2)}{\begin{array}{ll} M(\Sigma)(m)((ve_1{,}ve_2)) = & M(\Sigma)(m)(ve_1) \quad pipe \\ & (\lambda dv_1 \in M(\Sigma)(m)(t_1). \\ & \quad M(\Sigma)(m)(ve_2) \quad pipe \\ & \qquad (\lambda dv_2 \in M(\Sigma)(m)(t_2).v2p((dv_1, dv_2))) \\ &) \end{array}}$$

Dynamic Semantics of Function Expressions

Extensions of models are used when function expressions are given semantics.

$$\frac{\Sigma \vdash te \rhd t_1 \quad \Sigma \dagger (x, t_1) \vdash ve \rhd t_2 \quad \Sigma \vdash \lambda\ x : te \bullet ve \rhd s\_fct(t_1, t_2)}{M(\Sigma)(m)(\lambda\ x : te \bullet ve) = v2p(\lambda dv \in M(\Sigma)(m)(te).M(\Sigma \dagger (x, t_1))(m \dagger (x, dv))(ve))}$$

Dynamic Semantics of Application Expressions

When a function is applied to an argument, the semantics of the function is found and then the argument is evaluated and passed (piped) on to the semantics of the function.

$$\frac{\Sigma \vdash ve \rhd s\_fct(t_1, t_2) \quad \Sigma \vdash afp \rhd t_1 \quad \Sigma \vdash ve\ afp \rhd t_2}{\begin{array}{ll} M(\Sigma)(m)(ve\ afp) = & M(\Sigma)(m)(ve) \quad pipe \\ & (\lambda dv_1 \in M(\Sigma)(m)(s\_fct(t_1, t_2)). \\ & \quad M(\Sigma)(m)(afp) \quad pipe \\ & \qquad (\lambda dv_2 \in M(\Sigma)(m)(t_1).dv_1(dv_2)) \\ &) \end{array}}$$

Dynamic Semantics of Quantified Expressions

An auxiliary meaning function M^q is used for giving semantics to quantified expressions. Assuming that $\Sigma = \langle S, A, \Psi, V\rangle$ and Id is a set of identifiers, the function is typed as follows:

$$\begin{array}{l} M^q : \Pi\, \Sigma \in |Sign_{mRSL}| \bullet \Pi\, (x, t) \in Id \times T(S) \bullet \Pi\, W \in Types \bullet \\ \quad Mod(\Sigma) \longrightarrow ValExpr(\Sigma \dagger (x, t)) \longrightarrow \mathcal{P}(W). \end{array}$$

The function is defined by:

$$\frac{\Sigma \dagger (x, t) \vdash ve \rhd s\_bool}{M^q(\Sigma)((x, t))(W)(m)(ve) = \{dv \mid dv \in W.M(\Sigma \dagger (x, t))(m \dagger (x, dv))(ve) = v2p(tt)\}}$$

Given an identifier x, a set W, and a value expression ve (together with the other arguments needed), the function M^q yields the set of values dv in W for which ve is true when x is interpreted as dv.

The semantics of quantified expressions rely on the fact that types are sets.

$$\frac{\Sigma \vdash te \rhd t \quad \Sigma \dagger (x,t) \vdash ve \rhd s\_bool \quad \Sigma \vdash \forall\, x : te \bullet ve \rhd s\_bool}{\begin{array}{ll} M(\Sigma)(m)(\forall\, x : te \bullet ve) = & let \\ & \quad W_1 = M(\Sigma)(m)(te), \\ & \quad W_2 = M^q(\Sigma)((x,t))(W_1)(m)(ve), \\ & \quad b = (W_1 \cap W_2 = W_1) \\ & in \\ & \quad v2p(b) \\ & end \end{array}}$$

$$\frac{\Sigma \vdash te \rhd t \quad \Sigma \dagger (x,t) \vdash ve \rhd s\_bool \quad \Sigma \vdash \exists\, x : te \bullet ve \rhd s\_bool}{\begin{array}{ll} M(\Sigma)(m)(\exists\, x : te \bullet ve) = & let \\ & \quad W_1 = M(\Sigma)(m)(te), \\ & \quad W_2 = M^q(\Sigma)((x,t))(W_1)(m)(ve), \\ & \quad b = (W_1 \cap W_2 \neq \emptyset) \\ & in \\ & \quad v2p(b) \\ & end \end{array}}$$

Dynamic Semantics of Equivalence Expressions

Two value expressions are equivalent if and only if their semantics are identical.

$$\frac{\Sigma \vdash ve_1 \equiv ve_2 \rhd s\_bool}{\begin{array}{ll} M(\Sigma)(m)(ve_1 \equiv ve_2) = & let \\ & \quad b = (M(\Sigma)(m)(ve_1) = M(\Sigma)(m)(ve_2)) \\ & in \\ & \quad v2p(b) \\ & end \end{array}}$$

Dynamic Semantics of Value Infix Expressions

An auxiliary meaning function M^{io} is used for giving semantics to value infix expressions. Given an infix operator op of type $te_1 \times te_2 \stackrel{\sim}{\to} te_3$ and a pair of values in the value domains that correspond to te_1 and te_2, the function M^{io} yields a process which is the result of applying the operator to the pair of values. Operators that are assumed to be in the meta-language are used for defining M^{io}, and the result of M^{io} is only defined when the result of the meta-language operator is defined.

$$\frac{}{M^{io}(=)(x,y) = v2p(x = y)} \qquad \frac{}{M^{io}(\neq)(x,y) = v2p(x \neq y)}$$

$$\frac{}{M^{io}(<)(x,y) = v2p(x < y)} \qquad \frac{}{M^{io}(>)(x,y) = v2p(x > y)}$$

$$\frac{}{M^{io}(\leq)(x,y) = v2p(x \leq y)} \qquad \frac{}{M^{io}(\geq)(x,y) = v2p(x \geq y)}$$

$$\frac{}{M^{io}(+)(x,y) = v2p(x + y)} \qquad \frac{}{M^{io}(-)(x,y) = v2p(x - y)}$$

$$\frac{}{M^{io}(*)(x,y) = v2p(x * y)} \qquad \frac{}{M^{io}(/)(x,y) = v2p(x / y)}$$

$$\frac{}{M^{io}(\backslash)(x,y) = v2p(x \backslash y)}$$

Value infix expressions are given semantics that correspond to evaluating the arguments left to right and then apply the operator.

$$\frac{\Sigma \vdash ve_1 \rhd t_1 \quad \Sigma \vdash ve_2 \rhd t_2 \quad \vdash op \rhd s\_fct(s\_prod(t_1, t_2), t_3) \quad \Sigma \vdash ve_1\ op\ ve_2 \rhd t_3}{\begin{array}{rl} M(\Sigma)(m)(ve_1\ op\ ve_2) = & M(\Sigma)(m)(ve_1) \quad pipe \\ & \quad (\lambda dv_1 \in M(\Sigma)(m)(t_1). \\ & \quad\quad M(\Sigma)(m)(ve_2) \quad pipe \\ & \quad\quad\quad (\lambda dv_2 \in M(\Sigma)(m)(t_2).M^{io}(op)(dv_1, dv_2)) \\ & \quad) \end{array}}$$

Dynamic Semantics of Axiom Infix Expressions

An auxiliary meaning function M^{ic} is used for giving semantics to axiom infix expressions. Given an infix connective, a Boolean value, and a Boolean-valued process, the function M^{ic} yields a Boolean-valued process.

$$\frac{}{M^{ic}(\wedge)(b)(p) = if \quad (b = tt) \quad then \quad p \quad else \quad v2p(ff) \quad end}$$

$$\frac{}{M^{ic}(\vee)(b)(p) = if \quad (b = tt) \quad then \quad v2p(tt) \quad else \quad p \quad end}$$

$$\frac{}{M^{ic}(\Rightarrow)(b)(p) = if \quad (b = tt) \quad then \quad p \quad else \quad v2p(tt) \quad end}$$

Axiom infix expressions are given semantics that correspond to evaluating the left-hand argument and then evaluating the right-hand argument only if needed.

$$\frac{\Sigma \vdash ve_1 \; conn \; ve_2 \rhd s\_bool}{\begin{array}{ll} M(\Sigma)(m)(ve_1 \; conn \; ve_2) = & M(\Sigma)(m)(ve_1) \quad pipe \\ & \quad (\lambda dv \in Value_{s\_bool}. \\ & \qquad M^{ic}(conn)(dv)(M(\Sigma)(m)(ve_2)) \\ & \quad) \end{array}}$$

Dynamic Semantics of Value Prefix Expressions

An auxiliary function M^{po} is used for giving semantics to value prefix expressions.

$$\frac{\vdash \mathbf{abs} \rhd s\_fct(s\_int, s\_int)}{M^{po}(\mathbf{abs})(dv) = if \quad (dv < 0) \quad then \quad v2p(0 - dv) \quad else \quad v2p(dv) \quad end}$$

$$\frac{\vdash \mathbf{abs} \rhd s\_fct(s\_real, s\_real)}{M^{po}(\mathbf{abs})(dv) = if \quad (dv < 0.0) \quad then \quad v2p(0.0 - dv) \quad else \quad v2p(dv) \quad end}$$

Value prefix expressions are given semantics that correspond to evaluating the argument and then pass (pipe) the result to the operator.

$$\frac{\Sigma \vdash ve \rhd t_1 \quad \vdash op \rhd s\_fct(t_1, t_2) \quad \Sigma \vdash op \; ve \rhd t_2}{\begin{array}{ll} M(\Sigma)(m)(op \; ve) = & M(\Sigma)(m)(ve) \quad pipe \\ & \quad (\lambda dv \in M(\Sigma)(m)(t_1).M^{po}(op)(dv)) \end{array}}$$

Dynamic Semantics of Axiom Prefix Expressions

An auxiliary function M^{pc} is used for giving semantics to axiom prefix expressions.

$$\frac{}{M^{pc}(\sim)(b) = if \quad (b = tt) \quad then \quad v2p(ff) \quad else \quad v2p(tt) \quad end}$$

Axiom prefix expressions are given semantics that correspond to evaluating the argument and then pass (pipe) the result to the connective.

$$\frac{\Sigma \vdash conn \; ve \rhd s\_bool}{\begin{array}{ll} M(\Sigma)(m)(conn \; ve) = & M(\Sigma)(m)(ve) \quad pipe \\ & \quad (\lambda dv \in Value_{s\_bool}.M^{pc}(conn)(dv)) \end{array}}$$

Dynamic Semantics of Structured Expressions

If-expressions are given semantics that correspond to evaluating the condition and then select the branch that should be evaluated.

$$\frac{\Sigma \vdash \textbf{if}\ ve_1\ \textbf{then}\ ve_2\ \textbf{else}\ ve_3\ \textbf{end} \rhd t}{\begin{array}{ll} M(\Sigma)(m)(\textbf{if}\ ve_1\ \textbf{then}\ ve_2\ \textbf{else}\ ve_3\ \textbf{end}) = & M(\Sigma)(m)(ve_1) \quad pipe \\ & \quad (\lambda dv \in Values_{s\_bool}. \\ & \quad\quad if \\ & \quad\quad\quad dv = tt \\ & \quad\quad then \\ & \quad\quad\quad M(\Sigma)(m)(ve_2) \quad pipe \quad v2p \\ & \quad\quad else \\ & \quad\quad\quad M(\Sigma)(m)(ve_3) \quad pipe \quad v2p \\ & \quad\quad end \\ & \quad) \end{array}}$$

4.4.3 Examples

We conclude this presentation of the dynamic semantics of mRSL with a series of examples.

Example 4.9 *Let the model m interpret the value c as 2. Then $M(\Sigma)(m)(\mathrm{c} + 2)$ denotes the process that immediately terminates with the result 4.*

Example 4.10 *Recall the model m defined in Example 4.7. Since*

$$M(Sig(C2))(m)(\mathrm{black}) = v2p(tt) = M(Sig(C2))(m)(\textbf{true})$$

and

$$M(Sig(C2))(m)(\mathrm{white}) = v2p(ff) = M(Sig(C2))(m)(\textbf{false})$$

we have

$$m \models_{Sig(C2)} \mathrm{black} \equiv \textbf{true}$$

and

$$m \models_{Sig(C2)} \mathrm{white} \equiv \textbf{false}.$$

Moreover,

$$m \models_{Sig(C2)} (\mathrm{black} \neq \mathrm{white}) \equiv \textbf{true}.$$

Since m satifies all axioms (sentences) of the specification $C2$, it is a model of $C2$.

Example 4.11 *Recall the model n defined in Example 4.7. It is the case that*

$$n \models_{Sig(C2)} \text{black} \equiv \textbf{true}.$$

However, the model n does not satisfy all the axioms (sentences) of the specification $C2$, i.e., we do not have

$$n \models_{Sig(C2)} \text{white} \equiv \textbf{false}$$

nor

$$n \models_{Sig(C2)} (\text{black} \neq \text{white}) \equiv \textbf{true}.$$

Since n does not satisfy all axioms of the specification $C2$, it is not a model of $C2$.

4.5 The Satisfaction Condition

Before we prove the satisfaction condition, a preliminary result regarding type expressions is presented:

Proposition 4.42 *For all signatures Σ and Σ', signature morphisms $\sigma : \Sigma \longrightarrow \Sigma'$, Σ'-models m', and type expressions te over the signature Σ:*

$$M(\Sigma')(m')(Sen(\sigma)(te)) = M(\Sigma)(m'|_\sigma)(te).$$

Proof. By induction on the structure of type expressions. Assume that $\Sigma = \langle S, A, \Psi, V\rangle$, $\Sigma' = \langle S', A', \Psi', V'\rangle$, and $\sigma = \langle \sigma_S, \sigma_A, \sigma_V\rangle$.

By Definition 4.15, $Sen(\sigma)(t) = t$ for type literals t. And since type literals have the same interpretation in all mRSL models, the desired result holds for type literals.

Let $s \in S$. By Definition 4.15, the dynamic semantics of type expressions, Lemma 4.4, Definition 4.24, and Definition 4.26:

$$\begin{aligned}
& M(\Sigma')(m')(Sen(\sigma)(s)) \\
= \; & M(\Sigma')(m')(\sigma_S(s)) \\
= \; & \begin{cases} m'_{S'}(\sigma_S(s)) & \text{if } \sigma_S(s) \in S' \\ m'_{A'}(\sigma_S(s)) & \text{if } \sigma_S(s) \in A' \end{cases} \\
= \; & \begin{cases} m'_{T(S')}(\Psi'^*(\sigma_S(s))) & \text{if } \sigma_S(s) \in S' \\ m'_{T(S')}(\Psi'^*(\sigma_S(s))) & \text{if } \sigma_S(s) \in A' \end{cases} \\
= \; & (m'|_\sigma)_S(s) \\
= \; & M(\Sigma)(m'|_\sigma)(s).
\end{aligned}$$

Let $a \in A$. By Definition 4.15, the dynamic semantics of type expressions, and Definition 4.26:

$$M(\Sigma')(m')(Sen(\sigma)(a))$$
$$= M(\Sigma')(m')(\sigma_A(a))$$
$$= m'_{A'}(\sigma_A(a))$$
$$= (m'|_\sigma)_A(a)$$
$$= M(\Sigma)(m'|_\sigma)(a).$$

For Cartesian products $te_1 \times te_2$:

$$M(\Sigma')(m')(Sen(\sigma)(te_1 \times te_2))$$
$$= M(\Sigma')(m')(Sen(\sigma)(te_1) \times Sen(\sigma)(te_2))$$
$$= M(\Sigma')(m')(Sen(\sigma)(te_1)) \times M(\Sigma')(m')(Sen(\sigma)(te_2))$$
$$= M(\Sigma)(m'|_\sigma)(te_1) \times M(\Sigma)(m'|_\sigma)(te_2) \text{ by the induction hypothesis}$$
$$= M(\Sigma)(m'|_\sigma)(te_1 \times te_2).$$

The proofs for function type expressions and bracketed type expressions are very similar. □

The following proposition is essential for the satisfaction condition:

PROPOSITION 4.43 *For all signatures Σ and Σ', signature morphisms $\sigma : \Sigma \longrightarrow \Sigma'$, Σ'-models m', and value expressions ve over the signature Σ:*

$$M(\Sigma')(m')(Sen(\sigma)(ve)) = M(\Sigma)(m'|_\sigma)(ve).$$

PROOF. By induction on the structure of value expressions. Assume that $\Sigma = \langle S, A, \Psi, V\rangle$, $\Sigma' = \langle S', A', \Psi', V'\rangle$, and $\sigma = \langle \sigma_S, \sigma_A, \sigma_V\rangle$.

Since value literals and basic expressions have the same interpretation in all mRSL models, the desired result holds for value literals and basic expressions.

For value names $v \in V_t$ in Σ:

$$M(\Sigma')(m')(Sen(\sigma)(v))$$
$$= v2p(m'_{V'_{\Psi'^*(T(\sigma_S)(t))}}(Sen(\sigma)(v))) \text{ by the dynamic semantics}$$
$$= v2p(m'_{V'_{\Psi'^*(T(\sigma_S)(t))}}(\sigma_{V_t}(v))) \text{ by Definition 4.24}$$
$$= v2p((m'|_\sigma)_{V_t}(v)) \text{ by Definition 4.26}$$
$$= M(\Sigma)(m'|_\sigma)(v) \text{ by the dynamic semantics.}$$

Both function expressions and quantified expressions temporarily add an identifier to the signature (the "context", that is).

We are to show that $(m' \dagger (x, dv))|_{\dot\sigma} = (m'|_\sigma) \dagger (x, dv)$. The signature morphism $\dot\sigma$ maps the identifier x to x and maps as σ otherwise.

For $s \in S$:

$$\begin{aligned}
& ((m' \dagger (x, dv))|_{\dot\sigma})_S(s) \\
= \;& (m' \dagger (x, dv))_{T(S')}(\Psi'^*(\dot\sigma_S(s))) \\
= \;& m'_{T(S')}(\Psi'^*(\sigma_S(s))) \\
= \;& (m'|_\sigma)_S(s) \\
= \;& ((m'|_\sigma) \dagger (x, dv))_S(s).
\end{aligned}$$

For $a \in A$:

$$\begin{aligned}
& ((m' \dagger (x, dv))|_{\dot\sigma})_A(a) \\
= \;& (m' \dagger (x, dv))_{A'}(\dot\sigma_A(a)) \\
= \;& m'_{A'}(\sigma_A(a)) \\
= \;& (m'|_\sigma)_A(a) \\
= \;& ((m'|_\sigma) \dagger (x, dv))_A(a).
\end{aligned}$$

For $y \in V_t$, $y \neq x$:

$$\begin{aligned}
& ((m' \dagger (x, dv))|_{\dot\sigma})_{V_t}(y) \\
= \;& ((m' \dagger (x, dv))_{V'_{\Psi'^*(T(\dot\sigma_S)(t))}} \circ \dot\sigma_{V_t})(y) \\
= \;& (m' \dagger (x, dv))_{V'_{\Psi'^*(T(\dot\sigma_S)(t))}}(\dot\sigma_{V_t}(y)) \\
= \;& (m' \dagger (x, dv))_{V'_{\Psi'^*(T(\dot\sigma_S)(t))}}(\sigma_{V_t}(y)) \text{ since } y \neq x \\
= \;& m'_{V'_{\Psi'^*(T(\sigma_S)(t))}}(\sigma_{V_t}(y)) \\
= \;& (m'|_\sigma)_{V_t}(y) \\
= \;& ((m'|_\sigma) \dagger (x, dv))_{V_t}(y).
\end{aligned}$$

Finally:

$$\begin{aligned}
& ((m' \dagger (x, dv))|_{\dot\sigma})_{V_t}(x) \\
= \;& ((m' \dagger (x, dv))_{V'_{\Psi'^*(T(\dot\sigma_S)(t))}} \circ \dot\sigma_{V_t})(x) \\
= \;& (m' \dagger (x, dv))_{V'_{\Psi'^*(T(\dot\sigma_S)(t))}}(\dot\sigma_{V_t}(x))
\end{aligned}$$

$$= (m' \dagger (x, dv))_{V'_{\Psi'^*(T(\dot{\sigma}_S)(t))}}(x)$$

$$= dv$$

$$= ((m'|_\sigma) \dagger (x, dv))_{V_t}(x).$$

Consequently, $(m' \dagger (x, dv))|_{\sigma'} = (m'|_\sigma) \dagger (x, dv)$.
Let $\Sigma' \vdash Sen(\sigma)(te) \rhd t'$ and $\Sigma \vdash te \rhd t$.

$$M(\Sigma')(m')(Sen(\sigma)(\lambda\ x : te \bullet ve))$$

$$= M(\Sigma')(m')(\lambda\ x : Sen(\sigma)(te) \bullet Sen(\dot{\sigma})(ve))$$

$$= v2p(\lambda dv \in M(\Sigma')(m')(Sen(\sigma)(te)) \,.\, M(\Sigma' \dagger (x, t'))(m' \dagger (x, dv))(Sen(\dot{\sigma})(ve)))$$

$= v2p(\lambda dv \in M(\Sigma)(m'|_\sigma)(te) \,.\, M(\Sigma' \dagger (x, t'))(m' \dagger (x, dv))(Sen(\dot{\sigma})(ve)))$ by Proposition 4.42

$= v2p(\lambda dv \in M(\Sigma)(m'|_\sigma)(te) \,.\, M(\Sigma \dagger (x, t))((m' \dagger (x, dv))|_{\dot{\sigma}})(ve))$ by the induction hypothesis

$= v2p(\lambda dv \in M(\Sigma)(m'|_\sigma)(te) \,.\, M(\Sigma \dagger (x, t))(m'|_\sigma \dagger (x, dv))(ve))$ by the above

$$= M(\Sigma)(m'|_\sigma)(ve).$$

The proofs for quantified expressions are similar to the proof for function expressions.

The proofs for composite value expressions of the form $\mathcal{F}(expr_1, \ldots, expr_n)$ all follow the scheme:

$$M(\Sigma')(m')(Sen(\sigma)(\mathcal{F}(expr_1, \ldots, expr_n)))$$

$= M(\Sigma')(m')(\mathcal{F}(Sen(\sigma)(expr_1), \ldots, Sen(\sigma)(expr_n)))$ by Definition 4.16

$$= F(M(\Sigma')(m')(Sen(\sigma)(expr_1)), \ldots, M(\Sigma')(m')(Sen(\sigma)(expr_n)))$$

$= F(M(\Sigma)(m'|_\sigma)(expr_1), \ldots, M(\Sigma)(m'|_\sigma)(expr_n))$ by the induction hypothesis

$$= M(\Sigma)(m'|_\sigma)(\mathcal{F}(expr_1, \ldots, expr_n)).$$

where F is the semantic counterpart to $\mathcal{F}$. □

We can now state and prove the satisfaction condition expressing that truth is invariant under change of notation:

Proposition 4.44 *For all signatures Σ and Σ', signature morphisms $\sigma : \Sigma \longrightarrow \Sigma'$, Σ'-models m', and Σ-sentences e:*

$$m' \models_{\Sigma'} Sen(\sigma)(e) \qquad \textit{iff} \qquad m'|_\sigma \models_\Sigma e.$$

PROOF. For a Σ-sentence e of the form $ve_1 \equiv ve_2$ we have:

$m' \models_{\Sigma'} Sen(\sigma)(e)$

iff $m' \models_{\Sigma'} Sen(\sigma)(ve_1) \equiv Sen(\sigma)(ve_2)$

iff $M(\Sigma')(m')(Sen(\sigma)(ve_1)) = M(\Sigma')(m')(Sen(\sigma)(ve_2))$

iff $M(\Sigma)(m'|_\sigma)(ve_1) = M(\Sigma)(m'|_\sigma)(ve_2)$ by Proposition 4.43

iff $m'|_\sigma \models_\Sigma ve_1 \equiv ve_2$

iff $m'|_\sigma \models_\Sigma e$

which proves the satisfaction condition. □

EXAMPLE 4.12 *Recall the model m defined in Example 4.7, and recall the specifications* C1 *and* C2 *together with the signature morphism $\sigma : Sig(C1) \longrightarrow Sig(C2)$.*

The $Sig(C1)$-sentence

$$(\text{black} \neq \text{white}) \equiv \mathbf{true}$$

is translated to the identical $Sig(C2)$-sentence by $Sen(\sigma)$.

This, together with Example 4.10, gives us:

$$m \models_{Sig(C2)} Sen(\sigma)((\text{black} \neq \text{white}) \equiv \mathbf{true})$$

And by the satisfaction condition:

$$m|_\sigma \models_{Sig(C1)} (\text{black} \neq \text{white}) \equiv \mathbf{true}.$$

4.6 The Institution $mRSL$

In the previous sections, a subset of RSL has been described in terms of signatures, sentences, models, and a satisfaction relation. These notions form an institution:

DEFINITION 4.45 The institution $mRSL$ is the quadruple

$$\langle Sign_{mRSL}, Sen_{mRSL}, Mod_{mRSL}, \models \rangle$$

where $Sign_{mRSL}$ is given by Definition 4.11, Sen_{mRSL} is given by Definition 4.17, Mod_{mRSL} is given by Definition 4.29, and $\models$ is given by Definition 4.39.

Proposition 4.44 states that the satisfaction condition holds for the institution $mRSL$.

4.7 Discussion

After having presented an institution for a subset of RSL, we briefly return to the similarities and differences between the denotational semantics of RSL described in [43] and the institution. Furthermore, the choice of the subset of RSL is discussed, as well as possibilities for expanding the subset.

4.7.1 Relation to Original Semantics

The value domains for the basic mRSL types correspond directly to the original RSL value domains.

In full RSL, the semantics of a value expression is a function from a store to a process containing information about possible communication on channels, result values, and new states of the store (see Section 3.2.2). In mRSL, on the other hand, the semantics of a value expression is a simplified process. However, the mRSL semantics of a value expression is a special case of the full RSL semantics of a value expression.

Since mRSL value expressions do not contain access to variables, communication on channels, or nondeterminism, the full RSL semantics of a convergent mRSL value expression is a function that leaves the store unchanged, has only one possible result value, and does not communicate on channels. So the mRSL process $v2p(x)$ corresponds to the function

$$\lambda s \in \mathit{Store}.(\{(s, x)\}, [\,], [\,], \{(\emptyset, \emptyset, \{\surd\})\}).$$

The divergent mRSL process d_chaos corresponds to the function that maps all stores $s \in \mathit{Store}$ to the divergent RSL process $\perp$.

Since the original RSL semantics in [43] does not consider institutions, defining an institution for mRSL not only amounts to reducing the semantics of value expressions to a simple setting but also involves some restructuring.

As an example, RSL specifications have looser semantics whereas mRSL specifications have loose semantics. This change is reasonable since these two kinds of semantics are equivalent with respect to semantic consequence.

Another example of restructuring concerns notions of environments. In [43], there is a notion of *dynamic environments*, in which entities are assigned values in the semantic domains. Consequently, dynamic environments are similar to models in an institution.

Furthermore, there is a notion of static environments in the static semantics in [43]. A *static environment* assigns static values (static types) to the entities of a specification. Consequently, it is similar to the notion of a signature. However, a static environment also assigns static types to sort types, i.e., a static environment actually includes an interpretation of sort types. This is different from the institution in which sort types are interpreted by the models.

The meaning function M defined in a series of inference rules corresponds to the original denotational semantics; the inference rules were defined by simplifying the rules in the original denotational semantics. However, a strict proof of the correspondence and the correctness of the simplifications is not available.

4.7.2 Choice of Subset

The overall goal for defining an institution for RSL is to provide better proof support for RSL by means of a translation from RSL to Isabelle/HOL. Since it is easier to provide a high degree of automation in proof tools for simple languages than in proof tools for complex languages, it is sensible to consider a subset of RSL.

Using the RAISE method, proofs are most likely made for initial specifications, which usually are applicative. Thus, we chose mRSL to be applicative.

The translation must also be kept in mind, and to ensure that the Isabelle/HOL proof assistant can be reused without obtaining faulty results, the translation presented in Chapter 6 is closely related to the semantics. The semantics of RSL value expressions containing communication on channels and nondeterminism tend to become complex, and these features are therefore not considered in mRSL.

Including sorts in mRSL makes it possible to write algebraic specifications in mRSL; abbreviation types and the built-in types enable us to write model-oriented specifications.

4.7.3 Expanding mRSL

There are some straightforward expansions of mRSL. One of them is *pre-conditions* which can be expressed in the current mRSL. An equivalence expression with a pre-condition is of the form

$$\text{value\_expr\_1} \equiv \text{value\_expr\_2}\ \mathbf{pre}\ \text{value\_expr\_3}$$

and it is short for

$$(\text{value\_expr\_3} \equiv \mathbf{true}) \Rightarrow \text{value\_expr\_1} \equiv \text{value\_expr\_2}.$$

The RSL expressions that are shorthands for expressions in mRSL can be included in mRSL by defining their semantics as the semantics of their expansions.

An interesting expansion of mRSL would include sets and lists. This expansion is discussed in Section 6.7.3.

CHAPTER 5

Higher-Order Logic

This chapter concerns higher-order logic (HOL) and the proof assistant Isabelle/HOL.

HOL is presented as an institution, and the presentation is based on the formulation of an institution for HOL in [12] which is based on the model-oriented semantics of higher-order logic presented in [27]. The presentation introduces a lot of notation and only key notions are put in definitions.

The technical details of Isabelle/HOL presented in the last part of the chapter are taken from [47].

5.1 Signatures and Signature Morphisms

The first component of the institution contains signatures and signature morphisms.

5.1.1 Signatures

Signatures contain information about what types and constants can be used for writing sentences. Types are considered first.

$TyVars$ is an infinite set of *type variables* with a given linear order, and $TyNames$ is an infinite set of *names of type constants*. A *type constant* is a pair (ν, n) with the name $\nu \in TyNames$ and the arity n (a natural number, including 0). A type constant (ν, n) is also known as a *type constructor* or *type operator* with the arity n.

DEFINITION 5.1 A *type structure* Ω is a set of type constants (with distinct names). The set $Types_\Omega$ of *types* over a type structure Ω is generated by the grammar:

$$\tau \quad ::= \quad \gamma \quad | \quad c \quad | \quad (\tau_1, ...\tau_n)\nu \quad | \quad \tau_1 \rightarrow \tau_2$$

where $\gamma \in TyVars$, $(c, 0) \in \Omega$, and $(\nu, n) \in \Omega$.

In the following, τ (possibly with a subscript) is used for types, and γ, δ, and θ (possibly with subscripts) are used for type variables.

The function arrow $\rightarrow$ is a special type constant which has a fixed interpretation in the semantics. It is assumed that $\rightarrow$ is right associative so that we can write $\tau_1 \rightarrow \tau_2 \rightarrow \tau_3$ for $\tau_1 \rightarrow (\tau_2 \rightarrow \tau_3)$.

A type structure Ω and the associated set of types $Types_\Omega$ are *standard* if the distinguished type-constants $(o, 0)$ and $(\iota, 0)$ are in Ω, and they *have product* if $(\times, 2)$ is in Ω.

In the following, we use the infix notation $\gamma \times \delta$ and assume that $\times$ has higher precedence than $\rightarrow$ such that we can write $\gamma \times \delta \rightarrow \theta$ for $((\gamma, \delta)\times) \rightarrow \theta$.

A type τ may contain type variables. A *type context* $\boldsymbol{\gamma}$ is a list of type variables, and a *type-in-context* $\boldsymbol{\gamma}.\tau$ consists of a type context $\boldsymbol{\gamma}$ and a type τ such that all the type variables that appear in τ are in $\boldsymbol{\gamma}$. If $\boldsymbol{\gamma}$ contains exactly the type variables in τ, listed in order, then $\boldsymbol{\gamma}$ is the *canonical* type context of τ.

Constants representing values are now considered. $Names$ is an infinite set of *names of constants*. A *constant* has a name and a type. We write (c, τ) or c_τ for the constant with the name c and the type τ. When the type is known from the context, we may just write c.

Definition 5.2 A *HOL signature* is a pair $\Sigma = \langle \Omega, C \rangle$ consisting of

- a type structure Ω
- a set C of constants, i.e., names in $Names$ that are typed by types from the set $Types_\Omega$ of types over Ω.

A signature $\Sigma = \langle \Omega, C \rangle$ is *standard* if Ω is standard and C contains the constants $\mathbf{Q}_{\gamma \rightarrow \gamma \rightarrow o}$ and $\varepsilon_{(\gamma \rightarrow o) \rightarrow \gamma}$ where γ is a type variable. The signature Σ *has product* if Ω has product and C contains the constants $pair_{\gamma \rightarrow \delta \rightarrow \gamma \times \delta}$, $fst_{\gamma \times \delta \rightarrow \gamma}$, and $snd_{\gamma \times \delta \rightarrow \delta}$. In the following, we write (a, b) for $((pair\, a)\, b)$.

5.1.2 Signature Morphisms

Having defined signatures, we now turn to signature morphisms.

Definition 5.3 Let $\Sigma = \langle \Omega, C \rangle$ and $\Sigma' = \langle \Omega', C' \rangle$ be HOL signatures. A *HOL signature morphism* $\sigma : \Sigma \longrightarrow \Sigma'$ is a pair $\langle \sigma_\Omega, \sigma_C \rangle$ consisting of

- a mapping $\sigma_\Omega : \Omega \longrightarrow \Omega'$ of type constants that respect arities: if $\sigma_\Omega((\nu, n)) = (\nu', n')$, then $n = n'$.
- a mapping $\sigma_C : C \longrightarrow C'$ of constants that respects $\sigma_\Omega^{\#}$ (i.e., respects σ_Ω lifted to work on types in $Types_\Omega$): a constant c_τ is mapped to a constant $c'_{\sigma_\Omega^{\#}(\tau)}$.

The mapping σ_Ω is lifted to $\sigma_\Omega^{\#}$ that works on types in $Types_\Omega$ as follows:

- $\sigma^{\#}_{\Omega}(\gamma) = \gamma$
- $\sigma^{\#}_{\Omega}(c) = \sigma_{\Omega}(c)$
- $\sigma^{\#}_{\Omega}((\tau_1, \ldots, \tau_n)\nu) = (\sigma^{\#}_{\Omega}(\tau_1), \ldots, \sigma^{\#}_{\Omega}(\tau_n))(\sigma_{\Omega}(\nu))$
- $\sigma^{\#}_{\Omega}(\tau_1 \to \tau_2) = \sigma^{\#}_{\Omega}(\tau_1) \to \sigma^{\#}_{\Omega}(\tau_2)$.

A signature morphism $\sigma = \langle \sigma_{\Omega}, \sigma_C \rangle$ from the standard signature Σ to the standard signature Σ' is *standard* if the type constants $(o, 0)$ and $(\iota, 0)$ are preserved by σ_{Ω} and the constants $\mathbf{Q}_{\gamma \to \gamma \to o}$ and $\varepsilon_{(\gamma \to o) \to \gamma}$ are preserved by σ_C. If the signatures Σ and Σ' have product, then σ_{Ω} must also preserve the type constant $(\times, 2)$ and σ_C must also preserve the constants $pair_{\gamma \to \delta \to \gamma \times \delta}$, $fst_{\gamma \times \delta \to \gamma}$, and $snd_{\gamma \times \delta \to \delta}$ for σ to be standard.

Composition of Signature Morphisms

In the following, let $\Sigma = \langle \Omega, C \rangle$, $\Sigma' = \langle \Omega', C' \rangle$, and $\Sigma'' = \langle \Omega'', C'' \rangle$ be signatures; and let $\sigma = \langle \sigma_{\Omega}, \sigma_C \rangle$ and $\sigma' = \langle \sigma'_{\Omega'}, \sigma'_{C'} \rangle$ be signature morphisms $\sigma : \Sigma \longrightarrow \Sigma'$ and $\sigma' : \Sigma' \longrightarrow \Sigma''$.

Definition 5.4 The composition $\sigma' \circ \sigma$ of σ and σ' is the signature morphism $\Sigma \longrightarrow \Sigma''$ defined as $\langle \sigma'_{\Omega'} \circ \sigma_{\Omega}, \sigma'_{C'} \circ \sigma_C \rangle$.

Composition is associative, as stated in the following proposition:

Proposition 5.5 *For all signature morphisms $\sigma : \Sigma \longrightarrow \Sigma'$, $\sigma' : \Sigma' \longrightarrow \Sigma''$, and $\sigma'' : \Sigma'' \longrightarrow \Sigma'''$:*

$$\sigma'' \circ (\sigma' \circ \sigma) = (\sigma'' \circ \sigma') \circ \sigma.$$

Proof. Follows from associativity of function composition. □

Identities on Signatures

Some signature morphisms consist of mappings that are identities, in which case the entire signature morphism is called an *identity*.

Definition 5.6 Let $\Sigma = \langle \Omega, C \rangle$. The *identity* $id_{\Sigma} : \Sigma \longrightarrow \Sigma$ is the signature morphism $\langle id_{\Omega}, id_C \rangle$ where id_{Ω} is the identity on Ω and id_C is the identity on C.

These signature morphisms are identities in the category-theoretic sense.

Proposition 5.7 *For every HOL signature Σ, the identity $id_{\Sigma} : \Sigma \longrightarrow \Sigma$ satisfies $\sigma \circ id_{\Sigma} = \sigma$ for every $\sigma : \Sigma \longrightarrow \Sigma'$ and $id_{\Sigma} \circ \sigma' = \sigma'$ for every $\sigma' : \Sigma' \longrightarrow \Sigma$.*

PROOF. The components of the identity $id_\Sigma : \Sigma \longrightarrow \Sigma$ are identities on Ω and C, and composition of signature morphisms is defined by composition of components. Consequently, $\sigma \circ id_\Sigma = \langle \sigma_\Omega, \sigma_C \rangle \circ \langle id_\Omega, id_C \rangle = \langle \sigma_\Omega \circ id_\Omega, \sigma_C \circ id_C \rangle = \langle \sigma_\Omega, \sigma_C \rangle$ and $id_\Sigma \circ \sigma' = \langle id_\Omega, id_C \rangle \circ \langle \sigma'_{\Omega'}, \sigma'_{C'} \rangle = \langle id_\Omega \circ \sigma'_{\Omega'}, id_C \circ \sigma'_{C'} \rangle = \langle \sigma'_{\Omega'}, \sigma'_{C'} \rangle$. $\square$

5.1.3 The category $Sign_{HOL}$

The HOL signatures and HOL signature morphisms form a category.

DEFINITION 5.8 $Sign_{HOL}$ is the category of the standard HOL signatures with product and the standard HOL signature morphisms.

5.2 Sentences and Sentence Morphisms

Having defined signatures and signature morphisms, the sentences and sentence morphisms are considered.

5.2.1 Sentences

$VarNames$ is an infinite set of *names of variables*, assumed to be disjoint from $Names$ and with a given fixed linear order. A *variable* has a name and a type. We write (x, τ) or x^τ for the variable with the name x and the type τ. When the type is known from the context, we may just write x. If $\Sigma = \langle \Omega, C \rangle$ is a signature, then x^τ is a variable over Σ when $\tau \in Types_\Omega$. All variables are assumed to have distinct names: if x^{τ_1} and x^{τ_2} are variables, then $\tau_1 = \tau_2$.

Terms are formed by variables, constants, juxtapositions, and λ-abstractions. If $\Sigma = \langle \Omega, C \rangle$ is a signature, the set $Terms_\Sigma$ of terms over the signature Σ is generated by the grammar

$$t_\tau ::= x^\tau \mid c_\tau \mid (t_{\tau' \to \tau} t'_{\tau'})_\tau \mid (\lambda x^{\tau_1}.t_{\tau_2})_{\tau_1 \to \tau_2}$$

where the subscripts indicate the types of the terms[1], x^τ and x^{τ_1} are variables over Σ, c_τ is a constant in C, and $t_{\tau' \to \tau}$, $t'_{\tau'}$, and t_{τ_2} are terms.

The variable x^{τ_1} is *bound* in the term $(\lambda x^{\tau_1}.t_{\tau_2})_{\tau_1 \to \tau_2}$. A variable is *free* in a term if it occurs outside a binding λ-abstraction, and it is *bound* in a term t if it is bound in a subterm of t.

In λ-abstractions $(\lambda x^{\tau_1}.t_{\tau_2})_{\tau_1 \to \tau_2}$, the variable x^{τ_1} is assumed only to occur in the term t_{τ_2} as a free variable, i.e., λ-abstractions in t_{τ_2} do not introduce x^{τ_1} once more.

A term may contain variables and types that contain type variables. A *context* $\boldsymbol{\gamma}, \boldsymbol{x}$ consists of a type context $\boldsymbol{\gamma}$ and a list $\boldsymbol{x}$ of variables, and a *term-in-context* $\boldsymbol{\gamma}, \boldsymbol{x}.t$ consists of a context $\boldsymbol{\gamma}, \boldsymbol{x}$ and a term t such that

[1]The type τ may be the function type $\tau_1 \to \tau_2$.

- all the type variables that appear in the variables in $\boldsymbol{x}$ or in t are in $\boldsymbol{\gamma}$,
- all the free variables in t are in $\boldsymbol{x}$,
- no bound variables in t are in $\boldsymbol{x}$.

Consequently, a variable may not appear as both free or bound in t. If $\boldsymbol{\gamma}$ contains exactly the type variables in $\boldsymbol{x}$ and t, listed in order, and $\boldsymbol{x}$ contains exactly the variables in t, listed in order, then $\boldsymbol{\gamma}, \boldsymbol{x}$ is the *canonical* context of t.

Logical operators are used as abbreviations for terms [1, 12] as shown in Table 5.1. As we shall see in Section 5.6, there are other equivalent ways of defining the logical operators.

Table 5.1: Abbreviations for terms.

Abbreviation	Term
$t_\tau = t'_\tau$	$\mathbf{Q}_{\tau\to\tau\to o} t_\tau t'_\tau$
$True_o$	$\mathbf{Q}_{\tau\to\tau\to o} = \mathbf{Q}_{\tau\to\tau\to o}$
$False_o$	$(\lambda x^o.True_o) = (\lambda x^o.x^o)$
$\forall x^\tau.t_o$	$((\mathbf{Q}_{(\tau\to o)\to(\tau\to o)\to o}(\lambda x^\tau.True_o))(\lambda x^\tau.t_o))$
$\wedge_{o\to o\to o}$	$(\lambda x^o.(\lambda y^o.(\lambda g^{o\to o\to o}.(g\, True_o\, True_o)) = (\lambda g^{o\to o\to o}.(g\, x\, y))))$
$\Rightarrow_{o\to o\to o}$	$(\lambda x^o.(\lambda y^o.(x^o = (x^o \wedge y^o))))$
$\neg_{o\to o}$	$(\mathbf{Q}_{o\to o\to o}False_o)$
$\exists x^\tau.t_o$	$\neg(\forall x^\tau.\neg t_o)$
$IF_{o\to\tau\to\tau\to\tau\to o}$	$(\lambda b^o.(\lambda x^\tau.(\lambda y^\tau.(\lambda z^\tau.(b^o \Rightarrow z^\tau = x^\tau) \wedge (\neg b^o \Rightarrow z^\tau = y^\tau)))))$
$(if\, t_o\, then\, t'_\tau\, else\, t''_\tau)_\tau$	$(\varepsilon(IF\, t_o\, t'_\tau\, t''_\tau))$.

Having introduced quantifiers as abbreviations, a variable x^τ is also bound in terms $\forall x^\tau.t_o$ and $\exists x^\tau.t_o$.

A *formula* over the standard signature Σ is a Σ-term of type o that does not contain free variables.

Definition 5.9 A *HOL sentence* over a standard signature Σ is a Σ-formula.

For each standard signature Σ, we let $Sen(\Sigma)$ denote the set of sentences over Σ.

5.2.2 Sentence Morphisms

Signature morphisms are lifted to work on sentences.

Let $\Sigma = \langle\Omega, C\rangle$ and $\Sigma' = \langle\Omega', C'\rangle$ be signatures, and let $\sigma = \langle\sigma_\Omega, \sigma_C\rangle$ be a signature morphism from Σ to Σ'.

The mapping σ_C is extended to $\sigma_C^\#$ that works on terms:

- $\sigma_C^\#(x^\tau) = x^{\sigma_\Omega^\#(\tau)}$
- $\sigma_C^\#(c_\tau) = \sigma_C(c_\tau)$

- $\sigma_C^{\#}((t_{\tau'\to\tau}t'_{\tau'})_\tau) = (\sigma_C^{\#}(t_{\tau'\to\tau})\sigma_C^{\#}(t'_{\tau'}))_{\sigma_\Omega^{\#}(\tau)}$
- $\sigma_C^{\#}((\lambda x^{\tau_1}.t_{\tau_2})_{\tau_1\to\tau_2}) = (\lambda\sigma_C^{\#}(x^{\tau_1}).\sigma_C^{\#}(t_{\tau_2}))_{\sigma_\Omega^{\#}(\tau_1\to\tau_2)}$.

5.2.3 The Functor Sen_{HOL}

The functor Sen_{HOL} maps signatures to sets of sentences and lifts signature morphisms to work on sentences.

Definition 5.10 $Sen_{HOL} : Sign_{HOL} \longrightarrow Set$ is the functor that maps each signature in $Sign_{HOL}$ to the set $Sen(\Sigma)$ of sentences over Σ, and maps each signature morphism $\sigma = \langle\sigma_\Omega, \sigma_C\rangle$ from Σ to Σ' in $Sign_{HOL}$ to the function $\sigma_C^{\#}$.

5.3 Models

We now turn to the models (or interpretations) of HOL signatures. First, we look at a collection of sets which is the basis for defining the models. Then, we look at models and related functors.

5.3.1 The Universe $\mathcal{U}$

The syntax of higher-order logic comprises types and terms. Types denote sets, and terms denote elements of sets. The semantics is given using a fixed collection of sets $\mathcal{U}$, called the *universe*, with the following properties:

- the elements of the universe $\mathcal{U}$ are non-empty sets,
- if $S \in \mathcal{U}$, then $\mathcal{U}$ contains all non-empty subsets of S and the powerset of S,
- $\mathcal{U}$ is closed under Cartesian product, i.e., if $S_1 \in \mathcal{U}$ and $S_2 \in \mathcal{U}$ then $S_1 \times S_2 \in \mathcal{U}$,
- $\mathcal{U}$ contains a distinguished infinite set Ind,
- there is a distinguished element $ch \in \Pi_{S\in\mathcal{U}}S$ which is a function; $ch(S) \in S$ witnesses that the sets $S \in \mathcal{U}$ are non-empty.

The properties imply that $\mathcal{U}$ contains finite Cartesian products, function spaces, one-element sets, and two-element sets.

The notation $S_1 \longrightarrow S_2$ is used for the function space (the set of all functions) from S_1 to S_2 where $S_1 \in \mathcal{U}$ and $S_2 \in \mathcal{U}$.

A particular one-element set and a particular two-element set are selected such that:

- $\mathcal{U}$ contains a distinguished singleton set $\mathbf{1} = \{0\}$
- $\mathcal{U}$ contains a distinguished two-element set $\mathbf{2} = \{0, 1\}$.

The distinguished sets are used for providing standard interpretations of datatypes – the set **1** is used for the singleton type, and **2** is used for Booleans.

The notion of a universe $\mathcal{U}$ is introduced in [27] and presented in [12].

5.3.2 Models

Models provide interpretations of type constants and constants, and thereby interpretations of types and terms.

Definition 5.11 A *HOL model* m of a signature $\Sigma = \langle \Omega, C \rangle$ is a pair $\langle m_\Omega, m_C \rangle$ where

- m_Ω is a model of the type structure Ω, i.e., an interpretation of each type constant (ν, n) in Ω. If $n > 0$, then $m_\Omega(\nu, n)$ is a function $\mathcal{U}^n \longrightarrow \mathcal{U}$. If $n = 0$, then $m_\Omega(\nu, n)$ is a set in $\mathcal{U}$,
- m_C is a model of C, i.e., an interpretation of each constant c_τ in C as an element in the set $[\![\tau]\!]_{m_\Omega}$ which is the interpretation of τ with respect to m_Ω.

Given an interpretation m_Ω of a type structure Ω, the meaning of a type-in-context $\boldsymbol{\gamma}.\tau$, where $\boldsymbol{\gamma} = (\gamma_1, \ldots, \gamma_n)$, is a function $[\![\boldsymbol{\gamma}.\tau]\!]_{m_\Omega} : \mathcal{U}^n \longrightarrow \mathcal{U}$ defined as follows:

- $[\![\boldsymbol{\gamma}.\gamma_i]\!]_{m_\Omega}(S_1, \ldots, S_n) = S_i$ for $1 \leq i \leq n$ and $S_i \in \mathcal{U}$
- $[\![\boldsymbol{\gamma}.c]\!]_{m_\Omega}(\boldsymbol{S}) = m_\Omega(c, 0)$ for $\boldsymbol{S} \in \mathcal{U}^n$
- $[\![\boldsymbol{\gamma}.(\tau_1, \ldots, \tau_k)\nu]\!]_{m_\Omega}(\boldsymbol{S}) = m_\Omega(\nu, k)([\![\boldsymbol{\gamma}.\tau_1]\!]_{m_\Omega}(\boldsymbol{S}), \ldots, [\![\boldsymbol{\gamma}.\tau_k]\!]_{m_\Omega}(\boldsymbol{S}))$ for $\boldsymbol{S} \in \mathcal{U}^n$
- $[\![\boldsymbol{\gamma}.\tau_1 \to \tau_2]\!]_{m_\Omega}(\boldsymbol{S}) = [\![\boldsymbol{\gamma}.\tau_1]\!]_{m_\Omega}(\boldsymbol{S}) \longrightarrow [\![\boldsymbol{\gamma}.\tau_2]\!]_{m_\Omega}(\boldsymbol{S})$ for $\boldsymbol{S} \in \mathcal{U}^n$.

Given an interpretation m_Ω of a type structure Ω, the meaning of a type τ containing $n > 0$ type variables is a function $[\![\tau]\!]_{m_\Omega} : \mathcal{U}^n \longrightarrow \mathcal{U}$ defined as $[\![\boldsymbol{\gamma}.\tau]\!]_{m_\Omega}$ where $\boldsymbol{\gamma}$ is the canonical type context of τ. If τ does not contain any type variables, the meaning $[\![\tau]\!]_{m_\Omega}$ of τ is a set in $\mathcal{U}$, and $[\![\tau]\!]_{m_\Omega}$ is given by:

- $[\![c]\!]_{m_\Omega} = m_\Omega(c, 0)$
- $[\![(\tau_1, \ldots, \tau_k)\nu]\!]_{m_\Omega} = m_\Omega(\nu, k)([\![\tau_1]\!]_{m_\Omega}, \ldots, [\![\tau_k]\!]_{m_\Omega})$
- $[\![\tau_1 \to \tau_2]\!]_{m_\Omega} = [\![\tau_1]\!]_{m_\Omega} \longrightarrow [\![\tau_2]\!]_{m_\Omega}$.

A model $m = \langle m_\Omega, m_C \rangle$ of a standard signature $\Sigma = \langle \Omega, C \rangle$ is *standard* if

- $m_\Omega(o, 0) = \mathbf{2}$
- $m_\Omega(\iota, 0) = \mathit{Ind}$
- $m_C(\mathbf{Q}_{\gamma \to \gamma \to o})$ is *the equality test function.*

- $m_C(\varepsilon_{(\gamma\to o)\to\gamma})$ is *Hilbert's description operator*, i.e., the function that takes a function f of type $\tau \to o$ and returns an element v of type τ such that f applied to v is 1, if such an element exists[2].

If Σ has product, then it must also be the case that

- $m_\Omega(\times, 2) = (\lambda\,(S_1, S_2) \in \mathfrak{U}^2 \,.\, S_1 \times S_2)$ of type $\mathfrak{U}^2 \longrightarrow \mathfrak{U}$
- $m_C(pair_{\gamma\to\delta\to\gamma\times\delta}) : \Pi\,(S_1, S_2) \in \mathfrak{U}^2 \bullet S_1 \longrightarrow S_2 \longrightarrow S_1 \times S_2$ is the function that for $x \in S_1$ and $y \in S_2$ gives the element $(x, y) \in S_1 \times S_2$
- $m_C(fst_{\gamma\times\delta\to\gamma}) : \Pi\,(S_1, S_2) \in \mathfrak{U}^2 \bullet S_1$ is the *projection function* to the first element of a pair
- $m_C(snd_{\gamma\times\delta\to\delta}) : \Pi\,(S_1, S_2) \in \mathfrak{U}^2 \bullet S_2$ is the *projection function* to the second element of a pair.

The interpretation of a type $\tau \in Types_\Omega$ in the context of a model $m = \langle m_\Omega, m_C\rangle$ is $[\![\tau]\!]_{m_\Omega}$. The notation $[\![\tau]\!]_m$ is also used, and if τ only contains standard features so that the interpretation is the same for all standard models, we may write $[\![\tau]\!]$.

Similar to types, the interpretation of a term t in the context of a model m is denoted by $[\![t]\!]_m$ to be explained in Section 5.4.

For each standard signature $\Sigma = \langle\Omega, C\rangle$, the category $Mod(\Sigma)$ is the discrete category of standard models.

Our notion of a Σ-model $\langle m_\Omega, m_C\rangle$ is denoted a Σ-*structure* in [12] and corresponds to the models presented in [27]. In these models, only full function-spaces containing all functions are considered. Another kind of models, in which function spaces need not contain all functions, was introduced by Henkin in [35]. Henkin proved that simple type theory is complete when using these *general models* which are often called *Henkin models*. However, Henkin models are not considered here, nor in [12, 27].

5.3.3 The Reduct Functor

In the following, let $\Sigma = \langle\Omega, C\rangle$ and $\Sigma' = \langle\Omega', C'\rangle$ be signatures, let $\sigma : \Sigma \longrightarrow \Sigma'$ be a signature morphism, and let $m' = \langle m'_{\Omega'}, m'_{C'}\rangle$ be a Σ'-model.

Definition 5.12 For a Σ'-model m', the σ-*reduct* $m'|_\sigma$ is the Σ-model $m = \langle m_\Omega, m_C\rangle$ where $m_\Omega(\nu, n) = m'_{\Omega'}(\sigma_\Omega(\nu, n))$ for $(\nu, n) \in \Omega$ and $m_C(c_\tau) = m'_{C'}(\sigma_\Omega(c_\tau))$ for $c_\tau \in C$.

The σ-reduct construction gives rise to a functor.

[2] If $\varepsilon_{(\gamma\to o)\to\gamma}$ is applied to a function $g_{\tau\to o}$ that always returns 0, it still returns an element of type τ. As we shall see in Section 5.6, this may be used to define existence (the third definition, `Ex_def`, in Table 5.4): there exists an x that satisfies P if and only if (εP) satisfies P.

Definition 5.13 The *reduct functor* $Mod(\sigma) : Mod(\Sigma') \longrightarrow Mod(\Sigma)$ is the functor that maps each Σ'-model m' to the σ-reduct $m'|_\sigma$, and maps each morphism $id_{m'} : m' \longrightarrow m'$ in $Mod(\Sigma')$ to the morphism $id_{m'|_\sigma} : m'|_\sigma \longrightarrow m'|_\sigma$ in $Mod(\Sigma)$.

5.3.4 The Functor Mod_{HOL}

The concepts of signatures and models are used to define a functor.

Definition 5.14 $Mod_{HOL} : Sign_{HOL} \longrightarrow Cat^{op}$ is the functor that maps each signature Σ in $Sign_{HOL}$ to the discrete category $Mod(\Sigma)$ of models over the signature, and maps each signature morphism $\sigma : \Sigma \longrightarrow \Sigma'$ in $Sign_{HOL}$ to the reduct functor $Mod(\sigma) : Mod(\Sigma') \longrightarrow Mod(\Sigma)$ between the categories $Mod(\Sigma')$ and $Mod(\Sigma)$ of models.

5.4 Satisfaction Relation

Given a model $m = \langle m_\Omega, m_C \rangle$, the meaning of a type-in-context $\boldsymbol{\gamma}.\tau$, where $\boldsymbol{\gamma} = (\gamma_1, \ldots, \gamma_n)$, is the function $[\![\boldsymbol{\gamma}.\tau]\!]_{m_\Omega} : \mathfrak{U}^n \longrightarrow \mathfrak{U}$ which was described in Section 5.3.2. The notation $[\![\boldsymbol{\gamma}.\tau]\!]_m$ is used for $[\![\boldsymbol{\gamma}.\tau]\!]_{m_\Omega}$.

Given a model $m = \langle m_\Omega, m_C \rangle$, the meaning of a type τ containing $n > 0$ type variables is the function $[\![\tau]\!]_{m_\Omega} : \mathfrak{U}^n \longrightarrow \mathfrak{U}$ described in Section 5.3.2. If a type τ does not contain any type variables, its meaning is $[\![\tau]\!]_{m_\Omega}$ which was also described in Section 5.3.2. The notation $[\![\tau]\!]_m$ is used in both cases.

Given a model $m = \langle m_\Omega, m_C \rangle$, the interpretation of constants, m_C, is a function that maps each constant c_τ in C to an element in $\Pi\, \boldsymbol{S} \in \mathfrak{U}^n \bullet [\![\tau]\!]_m(\boldsymbol{S})$ if τ contains n type variables, and to an element in $[\![\tau]\!]_m$ if τ does not contain any type variables. In other words, if τ contains n type variables, then $m_C(c_\tau)$ is a function that takes an element $\boldsymbol{S} \in \mathfrak{U}^n$ and yields an element of the set $[\![\tau]\!]_m(\boldsymbol{S})$. If τ contains no type variables, then $m_C(c_\tau)$ is an element in the set $[\![\tau]\!]_m$.

Given a model $m = \langle m_\Omega, m_C \rangle$, the meaning of a term-in-context $\boldsymbol{\gamma}, \boldsymbol{x}.t_\tau$, where $\boldsymbol{\gamma} = (\gamma_1, \ldots, \gamma_n)$ and $\boldsymbol{x} = (x_1^{\tau_1}, \ldots, x_k^{\tau_k})$, is an element in

$$\Pi\, \boldsymbol{S} \in \mathfrak{U}^n \bullet ([\![\boldsymbol{\gamma}.\tau_1]\!]_m(\boldsymbol{S}) \times \ldots \times [\![\boldsymbol{\gamma}.\tau_k]\!]_m(\boldsymbol{S})) \longrightarrow [\![\boldsymbol{\gamma}.\tau]\!]_m(\boldsymbol{S}).$$

The vector $\boldsymbol{S}$ represents the interpretation of type variables in $\boldsymbol{\gamma}$, the Cartesian product represents the interpretation of variables in $\boldsymbol{x}$, and the result type is the type of the term (in the given interpretation of type variables).

If $n = 0$ and $k > 0$, i.e., there are no type variables, then $[\![\boldsymbol{\gamma}, \boldsymbol{x}.t_\tau]\!]_m$ is an element in

$$([\![\tau_1]\!]_m, \ldots, [\![\tau_k]\!]_m) \longrightarrow [\![\tau]\!]_m.$$

If $n > 0$ and $k = 0$, i.e., there are no variables, then $[\![\boldsymbol{\gamma}, \boldsymbol{x}.t_\tau]\!]_m$ is an element in

$$\Pi\, \boldsymbol{S} \in \mathfrak{U}^n \bullet [\![\boldsymbol{\gamma}.\tau]\!]_m(\boldsymbol{S}).$$

If $n = 0$ and $k = 0$, i.e., there are no type variables and no variables, then $[\![\gamma, \boldsymbol{x}.t_\tau]\!]_m$ is an element in $[\![\tau]\!]_m$. In this case, the meaning $[\![t_\tau]\!]_m$ of the term t is defined by:

- $[\![c_\tau]\!]_m = m_C(c_\tau)$
- $[\![(t_{\tau_1 \to \tau_2} t'_{\tau_1})_{\tau_2}]\!]_m = [\![t_{\tau_1 \to \tau_2}]\!]_m [\![t'_{\tau_1}]\!]_m$
- $[\![(\lambda x^{\tau_1}.t_{\tau_2})_{\tau_1 \to \tau_2}]\!]_m = (\lambda y \in [\![\tau_1]\!]_m.[\![(x^{\tau_1}).t_{\tau_2}]\!]_m(y))$. The variable x^{τ_1} is free in t_{τ_2}, so $[\![(x^{\tau_1}).t_{\tau_2}]\!]_m$ is an element in $[\![\tau_1]\!]_m \longrightarrow [\![\tau_2]\!]_m$, i.e., a function that takes an element in $[\![\tau_1]\!]_m$ and yields an element in $[\![\tau_2]\!]_m$. Consequently, $[\![(\lambda x^{\tau_1}.t_{\tau_2})_{\tau_1 \to \tau_2}]\!]_m$ is a function that takes an element in $[\![\tau_1]\!]_m$ and yields an element in $[\![\tau_2]\!]_m$.

The satisfaction relation is defined next.

Definition 5.15 Let Σ be a HOL signature, let m be a Σ-model, and let φ be a Σ-sentence. Then $m \models_\Sigma \varphi$ if and only if $[\![\varphi]\!]_m = 1$.

The satisfaction relation obeys the satisfaction condition for an institution.

Proposition 5.16 *For all signatures Σ and Σ', signature morphisms $\sigma : \Sigma \longrightarrow \Sigma'$, Σ'-models m', and Σ-sentences e:*

$$m' \models_{\Sigma'} Sen_{HOL}(\sigma)(e) \qquad \textit{iff} \qquad m'|_\sigma \models_\Sigma e.$$

The satisfaction condition is proved in [12].

5.5 The Institution *HOL*

The above definitions are now used to give a summarizing definition of an institution for higher-order logic.

Definition 5.17 The institution HOL is given by the quadruple

$$\langle Sign_{HOL}, Sen_{HOL}, Mod_{HOL}, \models \rangle$$

where $Sign_{HOL}$ is given by Definition 5.8, Sen_{HOL} is given by Definition 5.10, Mod_{HOL} is given by Definition 5.14, and $\models = (\models_\Sigma)_{\Sigma \in |Sign_{HOL}|}$ is given by Definition 5.15.

Proposition 5.16 states that the satisfaction condition holds for the institution HOL.

The institution in Definition 5.17 is essentially the target institution for the light institution comorphism presented in Chapter 6. However, in Section 5.6.3, we shall consider adding some datatypes and constants to the institution for HOL for convenience. The added datatypes and constants correspond to those that have been declared and developed in Isabelle/HOL, i.e., they build on the basic HOL institution.

5.6 Isabelle/HOL

In the above, the formulation of an institution for higher-order logic was considered. We now consider Isabelle/HOL which is the higher-order logic instance of the generic proof assistant Isabelle.

After a brief introduction of Isabelle in Section 5.6.1, Isabelle/HOL is introduced in Section 5.6.2, and incorporating Isabelle/HOL with the institution HOL is considered in Section 5.6.3. Then Section 5.6.4 shows how Isabelle/HOL provides concrete syntax for writing specifications and implements a proof system for higher-order logic. Furthermore, developing proofs using Isabelle is briefly considered.

5.6.1 Isabelle

Isabelle [49, 50, 51] is a generic proof assistant that supports a variety of logics. To instantiate Isabelle to an *object-logic*, the object-logic is encoded in Isabelle's *meta-logic* which is intuitionistic higher-order logic with equality. The encoding involves defining syntax (types, connectives, and operators) and an axiomatic semantics in the form of proof rules.

The Meta-Logic

Isabelle has polymorphic types involving type variables and type constructors. We use `'a`, `'b`, `'c`, etc. for type variables, and we use `_ => _` for function types. Curried function types `t1 => (t2 => t3)` may be abbreviated `[ t1, t2 ] => t3`, and the type `prop` represents truth values at the meta-level.

Type classes are sets of types, and a type variable may be restricted to be of a certain type class. The universal sort `{}` covers all types, and `logic` is a built-in type class for the types we want to reason about in an object-logic.

The notation `'a::tc` restricts the type variable `'a` to the type class `tc`, and likewise the notation `c::t` express that the constant `c` has the type `t`.

The meta-level connectives are as follows:

- `==>` of type `[ prop, prop ] => prop` is implication. The term `P ==> Q` reads "P implies Q".
- `!!` of type `('a::logic => prop) => prop` is universal quantification. The quantifier takes a predicate on an object-logic type and returns a truth value, and the term `!!x. P` reads "P is true for all x".
- `==` of type `['a::{}, 'a ] => prop` is equality. Meta-level equality works on all types, and the term `P == Q` reads "P equals Q".

For further introduction to Isabelle's meta-logic, we refer to [49, 51].

Proof Support

Isabelle provides a *simplifier* that tries to solve proof goals by performing rewriting, i.e., by substituting equals for equals. The simplifier uses a set of rules (known as a *simpset*) which may be modified by the user. Moreover, the simplifier performs automatic case-splits.

The simplifier needs to be set up for the object-logic of discourse. However, this has already been done for a number of object-logics (FOL, ZF, HOL, etc.). In Chapter 7, we use the simplifier as it is set up for HOL. However, rules corresponding to definitions from the encoding of mRSL in Isabelle/HOL and the specifications are added to the simpset when the simplifier is invoked.

Isabelle also includes a *classical reasoner* which provides a suite of automatic tactics when certain information about an object-logic is given. Like the simplifier, the classical reasoner has already been set up for a number of object-logics.

Moreover, Isabelle features a tableaux prover, automatic tactics that combine simplification and classical reasoning, and tactics for finding proofs by exhaustive search.

5.6.2 Isabelle/HOL

This section concerns the encoding of HOL in Isabelle which is described in [47]. The encoding comprises declarations of datatypes, type classes, constants, binders, and operators, as well as basic rules and definitions. Furthermore, a wide variety of rules are derived from the basic rules, and the simplifier and the classical reasoner are set up to provide proof support for HOL.

Encoding of Syntax

The syntax of HOL in Isabelle encompasses a series of types, type classes, constants, binders, and infix operators.

Some of the types provided are:

- `bool` (the type of truth values)
- `int` (the type of integer numbers)
- `real` (the type of real numbers)
- `unit` (the singleton type).

Furthermore, Isabelle/HOL includes a formulation of set theory which introduces the unary type-constructor `_ set`, and there is a binary infix type-constructor `_ * _` for products.

A series of type classes are defined; one of them is `ord` which contains partially ordered types.

The Isabelle/HOL constants are:

- `Trueprop` of type `bool => prop` is the coercion from the HOL type of truth values to the meta-level type of truth values.
- `Not` of type `bool => bool` is the negation operator.
- `True` of type `bool` denotes tautology.
- `False` of type `bool` denotes absurdity.
- `If` of type `[ bool, 'a, 'a ] => 'a` is a conditional operator. The usual notation is: `if` *term* `then` *term* `else` *term*.
- `Let` of type `[ 'a, 'a => 'b ] => 'b` denotes let binding. The usual notation is: `let` *id* `=` *term*`; ...;` *id* `=` *term* `in` *term*.

Compared with the HOL institution, the coercion `Trueprop` is a technicality due to the use of Isabelle, and `Let` is a convenient addition. The other constants are known from Table 5.1.

The binders are:

- `Eps` of type `('a => bool) => 'a` is Hilbert's description operator which selects an element of a type such that the element satisfies the given predicate. The symbols `SOME` and `@` are used.
- `All` of type `('a => bool) => bool` is universal quantification. The symbols `ALL` and `!` are used.
- `Ex` of type `('a => bool) => bool` is existential quantification. The symbols `EX` and `?` are used.
- `Ex1` of type `('a => bool) => bool` is unique existence. The symbols `EX!` and `?!` are used.
- `Least` of type `('a::ord => bool) => 'a` selects the least element of an ordered type such that the element satisfies the given predicate. The symbol `LEAST` is used.

Most of the binders are also recognized from Table 5.1. Unique existence and least choice on ordered types are convenient additions.

The infix operators are:

- `o` of type `[ 'b => 'c, 'a => 'b ] => ('a => 'c)` is function composition.
- `=` of type `[ 'a, 'a ] => bool` is the equality test operator.
- `<` of type `[ 'a::ord, 'a ] => bool` is "less than" (defined for types with an ordering of the elements).
- `<=` of type `[ 'a::ord, 'a ] => bool` is "less than or equal to" (defined for types with an ordering of the elements).

- `&` of type `[ bool, bool ] => bool` is conjunction.
- `|` of type `[ bool, bool ] => bool` is disjunction.
- `-->` of type `[ bool, bool ] => bool` is implication.

In Table 5.2, it is shown how the abbreviations for terms presented in Table 5.1 in Section 5.2.1 are written in Isabelle/HOL syntax. However, $IF_{o\rightarrow\tau\rightarrow\tau\rightarrow\tau\rightarrow o}$ from Table 5.1 is left out – it is not in Isabelle/HOL and it was primarily used for introducing $(if\, t_o\, then\, t'_\tau\, else\, t''_\tau)_\tau$.

Table 5.2: Isabelle/HOL syntax for HOL abbreviations.

HOL abbreviation	Isabelle/HOL term or operator
$t_\tau = t'_\tau$	`t = t'`
$True_o$	`True`
$False_o$	`False`
$\forall x^\tau . t_o$	`ALL x::tau . t`
$\wedge_{o\rightarrow o\rightarrow o}$	`&`
$\Rightarrow_{o\rightarrow o\rightarrow o}$	`-->`
$\neg_{o\rightarrow o}$	`Not`
$\exists x^\tau . t_o$	`EX x::tau . t`
$(if\, t_o\, then\, t'_\tau\, else\, t''_\tau)_\tau$	`(if t then t' else t'').`

Functions are expressed by λ-abstraction with the syntax `(% x. P)` and function application is expressed by juxtapositioning.

Axiomatic Semantics – Proof Rules

Various proof calculi for higher-order logic have been presented in the literature [1, 27]. Isabelle/HOL has eight basic inference rules and a wide variety of definitions and derived rules.

Using the Isabelle notation `[| P; Q |] ==> R` for `P ==> (Q ==> R)`, the basic HOL rules are shown in Table 5.3

Table 5.3: Basic HOL rules in Isabelle/HOL.

Name	Rule
`refl`	`t = (t::'a)`
`subst`	`[| s = t; P s |] ==> P (t::'a)`
`ext`	`(!!x::'a. (f x :: 'b) = g x) ==> (%x. f x) = (%x. g x)`
`impI`	`(P ==> Q) ==> P-->Q`
`mp`	`[| P -->Q; P |] ==> Q`
`iff`	`(P-->Q) --> (Q-->P) --> (P=Q)`
`someI`	`P(x::'a) ==> P(@x. P x)`
`True_or_False`	`(P=True) | (P=False)`

Constants and binders are specified by a series of definitions shown in Table 5.4. Note that `op` turns an infix operator into a function. Although the definitions are named, they are not intended to be used in proofs – a set of derived rules is provided instead.

Table 5.4: Basic HOL definitions in Isabelle/HOL.

Name	Definition	
`True_def`	`True`	`== ((%x::bool. x) = (%x. x))`
`All_def`	`All`	`== (%P. P = (%x. True))`
`Ex_def`	`Ex`	`== (%P. P(@x. P x))`
`False_def`	`False`	`== (!P. P)`
`not_def`	`not`	`== (%P. P --> False)`
`and_def`	`op &`	`== (%P Q. !R. (P --> Q --> R) --> R)`
`or_def`	`op |`	`== (%P Q. !R. (P --> R) --> (Q --> R) --> R)`
`Ex1_def`	`Ex1`	`== (%P. ? x. P x & (! y. P y --> y=x))`
`o_def`	`op o`	`== (%(f::'b=>'c) g x::'a. f(g x))`
`if_def`	`If P x y`	`== (%P x y. @z::'a.`
		`(P=True --> z=x) & (P=False --> z=y))`
`Let_def`	`Let s f`	`== f s`
`Least_def`	`Least P`	`== (@x. P(x) & (ALL y. P(y) --> x <= y))`

The formulation of the definitions is different from that of [1, 12]. However, we assume it to be sound with the model-theoretic semantics presented in [27].

The formulation of set theory in Isabelle/HOL provides additional constants, binders, definitions, and rules not shown here. See [47] for a full account.

5.6.3 Isabelle/HOL and the Institution HOL

After having presented the definition of the institution HOL and the syntax of Isabelle/HOL, a few remarks on their relation are appropriate.

Regarding types, the type `bool` in Isabelle/HOL is o in the HOL institution. Furthermore, the types `int`, `real`, and `unit` have been developed on the primitive basis of HOL. Consequently, the types are assumed to be part of HOL without further ado. In Chapter 6, syntax for these types will therefore be used in sentences of the HOL institution. This includes numerals and well-known operators for `int` and `real`, and () for `unit`.

The semantics of the types is defined such that $[\![unit]\!]_m = \mathbf{1}$ for all models m. Likewise, $[\![int]\!]_m$ and $[\![real]\!]_m$ denotes the integers and the reals, respectively.

Definition 5.18 For all standard models m, the semantics of the additional types are:

- $[\![bool]\!]_m = [\![o]\!]_m = \mathbf{2}$,
- $[\![int]\!]_m = \mathbb{Z}$,

- $[\![real]\!]_m = \mathbb{R}$,
- $[\![unit]\!]_m = \mathbf{1}$.

Since the interpretation of the types is independent of the model-context, the notation $[\![bool]\!]$, $[\![int]\!]$, $[\![real]\!]$, and $[\![unit]\!]$ is also used for $[\![bool]\!]_m$, $[\![int]\!]_m$, $[\![real]\!]_m$, and $[\![unit]\!]_m$, respectively.

Numerals and operators have the usual interpretations (as they are defined in the Isabelle/HOL encoding), and $[\![()]\!]_m = 0$.

5.6.4 Specifications and Proofs

This section concerns the concrete syntax used for writing Isabelle/HOL specifications and proofs.

Specifications

In the following, the Isabelle/Isar format [64] is used.

Using Isabelle/HOL terminology, a specification is also referred to as a *theory*. This is not to be confused with a theory over an institution as in Definition 2.15. An *Isabelle/HOL theory* is a named specification (Definition 2.13) consisting of a signature (with types and constants) and a set of sentences (axioms). In other words, an Isabelle/HOL theory (or Isabelle/HOL specification) is concrete syntax for a specification over the institution HOL.

An Isabelle/HOL theory is of the form

```
theory A = B :
  ...
end
```

where `A` is the name of the theory being defined, `B` is the name of an existing theory, and the ellipsis symbolizes a sequence of declarations. The theory of `A` is the theory of `B` extended with the declarations symbolized by the ellipsis.

In the following, the various forms of declarations are explained.

Abstract types may be declared as follows:

```
typedecl s
```

and types may be defined as names of existing types:

```
types T = type_expr
```

where `T` is introduced as a name for the type `type_expr`. Previously declared types and type constructors can occur in `type_expr`, i.e., types and type constructors declared in Isabelle/HOL (`bool`, `int`, `real`, `unit`, `_ set`, etc.) and user-declared types and type constructors. Type variables can also occur in `type_expr`.

Furthermore, inductive datatypes[3] may be declared. As an example, the type `newtype` may be introduced by:

[3] For details on inductive datatypes in Isabelle/HOL, see [6, 5].

```
datatype newtype == Con1 | Con2 bool
```

such that elements of the type are constructed by the constructors `Con1` and `Con2`, where the second constructor takes a Boolean argument. The images of constructors are disjoint, and constructors are injective. In this case, it means that `Con1` $\neq$ `Con2 b` for all Booleans `b`, and `Con2 true` $\neq$ `Con2 false`. Moreover, all elements of `newtype` can be generated by the constructors. In other words, `newtype` contains three elements corresponding to the terms `Con1`, `Con2 true`, and `Con2 false`.

A constant c_t is declared by:

```
consts
  c :: t
```

which declares c as a constant of type t. Here, t represents an arbitrary type expression which needs to be enclosed within citation marks when not atomic.

An arbitrary axiom may be declared by:

```
axioms
  axiom_name : "expression"
```

where `expression` is a term of type `prop`.

Declaring arbitrary axioms may lead to inconsistent specifications. Using Isabelle/HOL, it is therefore adviced to define constants using definitions that are simple definitions, definitions of primitive recursive functions, or generally recursive functions for which measure functions must be provided to ensure termination.

An example of a simple definition is:

```
constdefs
  xor :: "bool => bool => bool"
  "xor b1 b2 == b1 & (Not b2) | (Not b1) & b2"
```

where `b1` and `b2` are variables that can be instantiated when the definition is used in a proof.

Proofs

In classical Isabelle, a *proof script* is an ML-file containing a sequence of ML-commands that establish a proof goal and manipulate it by applying proof rules. In Isabelle/Isar [64], *proofs* are placed in theory files. Consequently, properties of constants declared in the theory file may be proved within the theory file.

A proof goal may be established by:

```
lemma id : "expression"
```

where `id` is the name of the lemma, and `expression` is a term of type `prop`. The proof goal can also be established by `theorem` or `corollary` instead of `lemma`.

Using Isar, the simplifier may be invoked by:

```
apply (simp add : rule_1 rule_2)
```

in which case the simplifier tries to solve the current goal using the current simpset and the rules `rule_1` and `rule_2`.

The command

```
apply auto
```

attempts to prove the current goal by an automatic method which combines simplification and classical reasoning.

A single rule – such as `eq_reflection` – can be applied by:

```
apply (rule eq_reflection)
```

If `e` is a variable of a free datatype `t`, the command:

```
apply (cases e)
```

invokes case analysis based on the definition of `t`.

The current goal may be replaced by subgoals during a proof, i.e., there is a list of subgoals to be proved. When all subgoals have been proved, the proof can be ended with `done`. Consequently, a proof has the following form:

```
lemma id : "expression"
  apply meth1
    ...
  done
```

where `id` is the name of the lemma to be proved, `expression` is the actual lemma, and `meth1` is a proof method.

CHAPTER 6

A Light Institution Comorphism from mRSL to HOL

In this chapter, we define a light institution comorphism $\varrho = \langle \varrho, \alpha, \beta \rangle$ from the institution for mRSL to the institution of HOL. The institution for mRSL was described in Chapter 4, and the institution for HOL was described in Chapter 5.

The light institution comorphism serves as a formal foundation for translating mRSL specifications (basic schemes) to Isabelle/HOL specifications. In other words, mRSL declarations are translated to Isabelle/HOL declarations by means of the mappings of signatures and sentences in the light institution comorphism.

The signature Σ of an mRSL specification stems from declarations of types and values. These declarations are translated to declarations of types and constants that make up part of the HOL signature $Sig(\varrho(\Sigma))$.

Axiom declarations in mRSL are translated to axiom declarations in Isabelle/HOL. This amounts to translating mRSL sentences to HOL sentences – as done by α.

EXAMPLE 6.1 *Recall the mRSL specification from Example 4.1:*

scheme C1 =
  class
    type Colour
    value black : Colour,
        white : Colour
    axiom [black_not_white] (black $\neq$ white) $\equiv$ **true**

end

This is translated to the following Isabelle/HOL specification:

```
theory C1 = RSL :
  typedecl Colour
  consts black :: "Colour"
         white :: "Colour"
  axioms black_not_white : "((Proc black) ~=rsl (Proc white)) = true"
end
```

where `RSL` *is a HOL theory containing declarations and definitions of constants, such as* `~=rsl`*, that facilitate the translation, i.e., are used to express semantics of mRSL value expressions in Isabelle/HOL. The translation of the occurrences of black and white in the axiom will be described later in this chapter.*

The mapping β of models is not found in the translation of mRSL specifications. However, it serves the important purpose of ensuring that the translation is defined in such a way that the use of Isabelle/HOL is sound.

In the following, we describe the components of a light institution comorphism: a mapping of signatures, a mapping of sentences, and a mapping of models. Moreover, the representation condition is proved and issues regarding the implementation of the light institution comorphism as a translation from mRSL syntax to Isabelle/HOL syntax are considered.

6.1 Mapping Signatures

The function ϱ maps an mRSL signature $\Sigma = \langle S, A, \Psi, V \rangle$ to a HOL presentation $\langle\langle \Omega_{RSL} \cup \Omega_\Sigma, C_{RSL} \cup C_\Sigma \rangle, E_{RSL} \rangle$ where

- Ω_{RSL} is a type structure containing the type constant $(Process, 1)$ for lifting value domains to process domains
- C_{RSL} is a set of typed constants containing:
  - constants for expressing the mRSL semantics in HOL
  - constants representing mRSL constructs in HOL (e.g., value literals, basic expressions, and infix operators)
- E_{RSL} is a set of sentences (definitions/axioms) specifying the constants for expressing mRSL semantics and mRSL constructs

and

- Ω_Σ is a type structure containing type constants $(s, 0)$ for all sort types $s \in S$
- C_Σ is a set of typed constants containing constants for values in V.

Notice that Ω_{RSL}, C_{RSL}, and E_{RSL} are fixed contributions, whereas Ω_Σ and C_Σ depend on the signature. Furthermore, this mapping of signatures serves as a theoretic foundation of a translation from RSL to Isabelle/HOL syntax.

Stricly speaking, $\langle\langle\Omega_{RSL} \cup \Omega_\Sigma, C_{RSL} \cup C_\Sigma\rangle, E_{RSL}\rangle$ is not a HOL presentation in the sense of Definition 5.17, but an "Isabelle/HOL presentation" based on an HOL institution in which the types and constants/operators already defined in Isabelle/HOL are available as described in Section 5.6.3. This is reflected by the translation, in which we rely on types and operators already present in Isabelle/HOL. For instance, we use the Isabelle/HOL types `bool`, `int`, `real`, and `unit` for representing the corresponding mRSL types without further ado. Moreover, miscellaneous Isabelle/HOL operators on integers and reals are used for expressing mRSL constructs.

6.1.1 Fixed Contributions

The constants in C_{RSL} and the axioms in E_{RSL} are now described. Since C_{RSL} and E_{RSL} together with Ω_{RSL} so to speak represents the encoding of the mRSL language, the presentation is organized according to the concepts of mRSL.

Process Domains

The type structure Ω_{RSL} provides a type constant $(Process, 1)$ so that RSL process domains $Procs_t$ can be represented by $(\tau)Process$ where τ is the translation of t.

The set C_{RSL} contains a family of constants that are *constructors* of the $(\gamma)Process$ types representing process domains:

$$(Proc,\ \gamma \to (\gamma)Process)$$

$$(Chaos,\ (\gamma)Process).$$

The following sentences in E_{RSL} express that the images of the constructors are disjoint, the constructors are injective, and all elements of a type $(\gamma)Process$ can be generated by the constructors:

$$\forall x^\gamma.\ \neg(Chaos = Proc\ x)$$

$$\forall x^\gamma.\ (\forall y^\gamma.\ (Proc\ x = Proc\ y) \Rightarrow (x = y))$$
$$\forall P^{(\gamma)Process \to o}.\ ((P\ Chaos \wedge (\forall\ x^\gamma\ .\ P\ (Proc\ x))) \Rightarrow (\forall\ x^{(\gamma)Process}\ .\ P\ x)).$$

Semantic Operators

The semantic operators *pipe* and *v2p* (from Section 4.3.2) are represented by the constants

$$(pipe,\ (\gamma)Process \to (\gamma \to (\delta)Process) \to (\delta)Process)$$

and

$$(v2p,\ \gamma \to (\gamma)Process)$$

in C_{RSL}. The constants are defined by the following sentences in E_{RSL}:

$$\forall\ x^{\gamma}\ .\ \forall\ f^{\gamma \to (\gamma)Process}\ .\ pipe\ (Proc\ x)\ f = f\ x$$

$$\forall\ f^{\gamma \to (\delta)Process}\ .\ pipe\ Chaos\ f = Chaos$$

$$\forall\ x^{\gamma}\ .\ v2p\ x = Proc\ x.$$

Value Literals

The Boolean value literals are represented in C_{RSL} by the constants

$$(true,\ (bool)Process)$$

$$(false,\ (bool)Process)$$

defined by the following sentences in E_{RSL}:

$$true = Proc\ True$$

$$false = Proc\ False.$$

Basic Expressions

The basic expressions are represented in C_{RSL} by the constants

$$(chaos,\ (\gamma)Process)$$

$$(skip,\ (unit)Process)$$

defined by the following sentences in E_{RSL}:

$$chaos = Chaos$$

$$skip = Proc\ ().$$

Product Expressions

C_{RSL} contains the following constant used for representing products:

$$(prod,\ (\gamma)Process \to (\delta)Process \to (\gamma \times \delta)Process).$$

E_{RSL} contains the following defining sentence:

$$\forall\ ve_1^{(\gamma)Process}.\ \forall\ ve_2^{(\delta)Process}.$$
$$prod\ ve_1\ ve_2 = pipe\ ve_1\ (\lambda dv_1^{\gamma}\ .\ pipe\ ve_2\ (\lambda dv_2^{\delta}\ .\ v2p\ (dv_1, dv_2))).$$

Application Expressions

An operator for function application is represented in C_{RSL} by the constant:

$$(appl,\ (\gamma \rightarrow (\delta)Process)Process \rightarrow (\gamma)Process \rightarrow (\delta)Process).$$

It is defined by the following sentence in E_{RSL}:

$$\begin{array}{l}\forall\ f^{(\gamma \rightarrow (\delta)Process}.\ \forall\ x^{(\gamma)Process}.\\ \quad appl\ f\ x = pipe\ f\ (\lambda dv_1^{\gamma \rightarrow (\delta)Process}\ .\ pipe\ x\ (\lambda dv_2^{\gamma}\ .\ dv_1\ dv_2)).\end{array}$$

Infix Expressions

Infix expressions are represented by constants for the infix operators and connectives together with constants for application of infix operators and connectives.

Application of infix operators is represented in C_{RSL} by the constant

$$(infix\_appl,\ (\gamma)Process \rightarrow (\gamma \rightarrow \delta \rightarrow (\theta)Process) \rightarrow (\delta)Process \rightarrow Process(\theta))$$

which is defined in E_{RSL} by the axiom

$$\begin{array}{l}\forall\ x^{(\gamma)Process}.\ \forall\ iop^{\gamma \rightarrow \delta \rightarrow (\theta)Process}.\ \forall\ y^{(\delta)Process}.\\ \quad infix\_appl\ x\ iop\ y = pipe\ x\ (\lambda dv_1^{\gamma}\ .\ pipe\ y\ (\lambda dv_2^{\delta}\ .\ iop\ dv_1\ dv_2)).\end{array}$$

The infix operators are represented by a series of constants:

$$(RSLeq,\ \gamma \rightarrow \gamma \rightarrow (bool)Process)$$

$$(RSLneq,\ \gamma \rightarrow \gamma \rightarrow (bool)Process)$$

$$(RSLl,\ int \rightarrow int \rightarrow (bool)Process)$$

$$(RSLg,\ int \rightarrow int \rightarrow (bool)Process)$$

$$(RSLgeq,\ int \rightarrow int \rightarrow (bool)Process)$$

$$(RSLleq,\ int \rightarrow int \rightarrow (bool)Process)$$

$$(RSLplus,\ int \rightarrow int \rightarrow (int)Process)$$

$$(RSLminus,\ int \rightarrow int \rightarrow (int)Process)$$

$$(RSLmod,\ int \rightarrow int \rightarrow (int)Process)$$

$$(RSLmult,\ int \rightarrow int \rightarrow (int)Process)$$

$$(RSLdiv,\ int \rightarrow int \rightarrow (int)Process)$$

$$(RSLl\_real,\ real \rightarrow real \rightarrow (bool)Process)$$

$$(RSLg\_real,\ real \rightarrow real \rightarrow (bool)Process)$$

$$(RSLgeq\_real,\ real \rightarrow real \rightarrow (bool)Process)$$

$$(RSLleq\_real,\ real \rightarrow real \rightarrow (bool)Process)$$
$$(RSLplus\_real,\ real \rightarrow real \rightarrow (real)Process)$$
$$(RSLminus\_real,\ real \rightarrow real \rightarrow (real)Process)$$
$$(RSLmult\_real,\ real \rightarrow real \rightarrow (real)Process)$$

which are defined by the axioms:

$$\forall x^{\gamma}.\forall y^{\gamma}.\ RSLeq\ x\ y = v2p\ (x = y)$$
$$\forall x^{\gamma}.\forall y^{\gamma}.\ RSLneq\ x\ y = v2p\ (x \neq y)$$
$$\forall x^{int}.\forall y^{int}.\ RSLl\ x\ y = v2p\ (x < y)$$
$$\forall x^{int}.\forall y^{int}.\ RSLg\ x\ y = v2p\ (y < x)$$
$$\forall x^{int}.\forall y^{int}.\ RSLgeq\ x\ y = v2p\ (y \leq x)$$
$$\forall x^{int}.\forall y^{int}.\ RSLleq\ x\ y = v2p\ (x \leq y)$$
$$\forall x^{int}.\forall y^{int}.\ RSLplus\ x\ y = v2p\ (x + y)$$
$$\forall x^{int}.\forall y^{int}.\ RSLminus\ x\ y = v2p\ (x - y)$$
$$\forall x^{int}.\forall y^{int}.\ RSLmod\ x\ y = v2p\ (x\ mod\ y)$$
$$\forall x^{int}.\forall y^{int}.\ RSLmult\ x\ y = v2p\ (x * y)$$
$$\forall x^{int}.\forall y^{int}.\ RSLdiv\ x\ y = v2p\ (x\ div\ y)$$
$$\forall x^{real}.\forall y^{real}.\ RSLl\_real\ x\ y = v2p\ (x < y)$$
$$\forall x^{real}.\forall y^{real}.\ RSLg\_real\ x\ y = v2p\ (y < x)$$
$$\forall x^{real}.\forall y^{real}.\ RSLgeq\_real\ x\ y = v2p\ (y \leq x)$$
$$\forall x^{real}.\forall y^{real}.\ RSLleq\_real\ x\ y = v2p\ (x \leq y)$$
$$\forall x^{real}.\forall y^{real}.\ RSLplus\_real\ x\ y = v2p\ (x + y)$$
$$\forall x^{real}.\forall y^{real}.\ RSLminus\_real\ x\ y = v2p\ (x - y)$$
$$\forall x^{real}.\forall y^{real}.\ RSLmult\_real\ x\ y = v2p\ (x * y).$$

Infix notation is provided by means of the following abbreviations:

$$p_1 =rsl\ p_2 = infix\_appl\ p_1\ RSLeq\ p_2$$
$$p_1 \sim\!=rsl\ p_2 = infix\_appl\ p_1\ RSLneq\ p_2$$
$$p_1 <rsl\ p_2 = infix\_appl\ p_1\ RSLl\ p_2$$
$$p_1 >=rsl\ p_2 = infix\_appl\ p_1\ RSLg\ p_2$$
$$p_1 >=rsl\ p_2 = infix\_appl\ p_1\ RSLgeq\ p_2$$
$$p_1 <=rsl\ p_2 = infix\_appl\ p_1\ RSLleq\ p_2$$

$$p_1 \; +rsl \; p_2 = infix\_appl \; p_1 \; RSLplus \; p_2$$
$$p_1 \; -rsl \; p_2 = infix\_appl \; p_1 \; RSLminus \; p_2$$
$$p_1 \; modrsl \; p_2 = infix\_appl \; p_1 \; RSLmod \; p_2$$
$$p_1 \; *rsl \; p_2 = infix\_appl \; p_1 \; RSLmult \; p_2$$
$$p_1 \; divrsl \; p_2 = infix\_appl \; p_1 \; RSLdiv \; p_2$$
$$p_1 \; <rrsl \; p_2 = infix\_appl \; p_1 \; RSLl\_real \; p_2$$
$$p_1 \; >rrsl \; p_2 = infix\_appl \; p_1 \; RSLg\_real \; p_2$$
$$p_1 \; >=rrsl \; p_2 = infix\_appl \; p_1 \; RSLgeq\_real \; p_2$$
$$p_1 \; <=rrsl \; p_2 = infix\_appl \; p_1 \; RSLleq\_real \; p_2$$
$$p_1 \; +rrsl \; p_2 = infix\_appl \; p_1 \; RSLplus\_real \; p_2$$
$$p_1 \; -rrsl \; p_2 = infix\_appl \; p_1 \; RSLminus\_real \; p_2$$
$$p_1 \; *rrsl \; p_2 = infix\_appl \; p_1 \; RSLmult\_real \; p_2.$$

Application of infix connectives is represented in C_{RSL} by the constant:

$$(infix\_conn\_appl, \quad \begin{array}{l} (bool)Process \rightarrow \\ (bool \rightarrow (bool)Process \rightarrow (bool)Process) \rightarrow \\ (bool)Process \rightarrow (bool)Process \end{array})$$

which is defined in E_{RSL} by the axiom:

$$\forall \; ve_1^{(bool)Process}. \; \forall \; iconn^{bool \rightarrow (bool)Process \rightarrow (bool)Process}. \; \forall \; ve_2^{(bool)Process}.$$
$$infix\_conn\_appl \; ve_1 \; iconn \; ve_2 = pipe \; ve_1 \; (\lambda dv_1^{bool} \; . \; iconn \; dv_1 \; ve_2).$$

The infix connectives ($\wedge$, $\vee$, and $\Rightarrow$) are represented in C_{RSL} by the constants:

$$(RSLand, \; bool \rightarrow (bool)Process \rightarrow (bool)Process)$$
$$(RSLor, \; bool \rightarrow (bool)Process \rightarrow (bool)Process)$$
$$(RSLimp, \; bool \rightarrow (bool)Process \rightarrow (bool)Process)$$

which are defined in E_{RSL} by the axioms

$$\forall b^{bool}.\forall p^{(bool)Process}. \; RSLand \; b \; p = if \; b \; then \; p \; else \; (v2p \; False)$$
$$\forall b^{bool}.\forall p^{(bool)Process}. \; RSLor \; b \; p = if \; b \; then \; (v2p \; True) \; else \; p$$
$$\forall b^{bool}.\forall p^{(bool)Process}. \; RSLimp \; b \; p = if \; b \; then \; p \; else \; (v2p \; True).$$

Moreover, infix notation is provided:

$$bp_1 \; AND \; bp_2 = infix\_conn\_appl \; bp_1 \; RSLand \; bp_2$$
$$bp_1 \; OR \; bp_2 = infix\_conn\_appl \; bp_1 \; RSLor \; bp_2$$
$$bp_1 \; IMP \; bp_2 = infix\_conn\_appl \; bp_1 \; RSLimp \; bp_2.$$

Prefix Expressions

Prefix expressions are represented by constants for the prefix operator (**abs**) and connective ($\sim$) together with constants for application of prefix operators and connectives.

Application of a prefix operator is represented in C_{RSL} by the constant

$$(prefix\_appl,\ (\gamma \rightarrow (\delta)Process) \rightarrow (\gamma)Process \rightarrow (\delta)Process)$$

which is defined in E_{RSL} by the axiom

$$\forall pop^{\gamma\rightarrow(\delta)Process}.\forall x^{(\gamma)Process}.\ prefix\_appl\ pop\ x = pipe\ x\ (\lambda dv_1^{\gamma}\ .\ pop\ dv_1).$$

The prefix operator (**abs**) is represented in C_{RSL} by the constants:

$$(RSLabs,\ int \rightarrow (int)Process)$$

$$(RSLabs,\ real \rightarrow (real)Process)$$

which are defined in E_{RSL} by the axioms:

$$\forall x^{int}.\ RSLabs\ x = v2p\ (abs\ x)$$

$$\forall x^{real}.\ RSLabs\ x = v2p\ (abs\ x)$$

where *abs* is a pre-defined function available in Isabelle/HOL. Prefix notation is provided:

$$absrsl\ p = prefix\_appl\ RSLabs\ p.$$

Application of a prefix connective is represented in C_{RSL} by the constant:

$$(prefix\_conn\_appl,\ (bool \rightarrow (bool)Process) \rightarrow (bool)Process \rightarrow (bool)Process)$$

which is defined in E_{RSL} by the axiom:

$$\begin{aligned}&\forall\ pconn^{bool\rightarrow(bool)Process}.\ \forall ve^{(bool)Process}.\\ &\quad prefix\_conn\_appl\ pconn\ ve = pipe\ ve\ (\lambda dv_1^{bool}\ .\ pconn\ dv_1).\end{aligned}$$

The infix connective ($\sim$) is represented in C_{RSL} by the constant:

$$(RSLnot,\ bool \rightarrow (bool)Process)$$

which is defined in E_{RSL} by the axiom:

$$\forall b^{bool}.\ RSLnot\ b = if\ b\ then\ (v2p\ False)\ else\ (v2p\ True).$$

Prefix notation is provided:

$$NOT\ bp = prefix\_conn\_appl\ RSLnot\ bp.$$

Structured Expressions

Structured expressions are represented in C_{RSL} by the constant

$$(ifexpr,\ (bool)Process \to (\gamma)Process \to (\gamma)Process \to (\gamma)Process).$$

Using the infix notation $if\ \_\ then\ \_\ else\ \_\ end$, the constant $ifexpr$ is defined in E_{RSL} by the axiom

$$\begin{array}{l}\forall\ ve_b^{(bool)Process}.\ \forall\ ve_1^{(\gamma)Process}.\ \forall ve_2^{(\gamma)Process}.\\ \quad if\ ve_b\ then\ ve_1\ else\ ve_2\ end\\ \quad\quad =\\ \quad pipe\ ve_b\ (\lambda dv_1^{bool}\ .\ if\ dv_1\ then\ (pipe\ ve_1\ v2p)\ else\ (pipe\ ve_2\ v2p)).\end{array}$$

6.1.2 Signature-dependent Contributions

The elements of C_Σ stems from the value names in V. A value name v in V_t, $t \in T(S)$, yields a constant c of type $A^{st}_\Sigma(t)$ where A^{st}_Σ is the function defined below.

Definition 6.1 Let $\Sigma = \langle S, A, \Psi, V\rangle$ be an mRSL signature. Let the function

$$A^{st}_\Sigma : T(S \cup A) \longrightarrow Types_{(\Omega_{RSL} \cup \Omega_\Sigma)}$$

be defined by:

$$\begin{array}{lcl}A^{st}_\Sigma(s) & \doteq & s \text{ for } s \in S\\ A^{st}_\Sigma(a) & \doteq & A^{st}_\Sigma(\Psi^*(a)) \text{ for } a \in A\\ A^{st}_\Sigma(s\_bool) & \doteq & bool\\ A^{st}_\Sigma(s\_int) & \doteq & int\\ A^{st}_\Sigma(s\_real) & \doteq & real\\ A^{st}_\Sigma(s\_unit) & \doteq & unit\\ A^{st}_\Sigma(s\_prod(t_1, t_2)) & \doteq & A^{st}_\Sigma(t_1) \times A^{st}_\Sigma(t_2)\\ A^{st}_\Sigma(s\_fct(t_1, t_2)) & \doteq & A^{st}_\Sigma(t_1) \to (A^{st}_\Sigma(t_2))Process.\end{array}$$

Below, the functions A^{te}_Σ, $\Sigma \in |Sign_{mRSL}|$, are introduced to translate mRSL type expressions (occurring in mRSL sentences) to Isabelle/HOL types, i.e., a translation at the level of syntax. The definition of A^{te}_Σ conforms with that of A^{st}_Σ.

Definition 6.2 Let $\Sigma = \langle S, A, \Psi, V\rangle$ be an mRSL signature. Let the function

$$A^{te}_\Sigma : TypeExpr(\Sigma) \longrightarrow Types_{(\Omega_{RSL} \cup \Omega_\Sigma)}$$

be the function defined by:

$$\begin{array}{lcl}A^{te}_\Sigma(s) & \doteq & s \text{ for } s \in S\\ A^{te}_\Sigma(a) & \doteq & A^{st}_\Sigma(\Psi^*(a)) \text{ for } a \in A\\ A^{te}_\Sigma(\mathbf{Bool}) & \doteq & bool\\ A^{te}_\Sigma(\mathbf{Int}) & \doteq & int\\ A^{te}_\Sigma(\mathbf{Real}) & \doteq & real\\ A^{te}_\Sigma(\mathbf{Unit}) & \doteq & unit\\ A^{te}_\Sigma(t_1 \times t_2) & \doteq & A^{te}_\Sigma(t_1) \times A^{te}_\Sigma(t_2)\\ A^{te}_\Sigma(t_1 \stackrel{\sim}{\to} t_2) & \doteq & A^{te}_\Sigma(t_1) \to (A^{te}_\Sigma(t_2))Process.\end{array}$$

6.1.3 Mapping Signature Morphisms

In the definition of an institution comorphism $\langle \varrho, \alpha, \beta \rangle$, it is required that ϱ is a functor. Consequently, ϱ must map signatures in $Sign_{mRSL}$ to presentations in $Pres_{HOL}$ and map signature morphisms in $Sign_{mRSL}$ to presentation morphisms in $Pres_{HOL}$. However, the naive definition of a mapping of signature morphisms is problematic as we shall see in the following.

The mRSL signatures in $Sign_{mRSL}$ contain a notion of abbreviation types, and a similar notion is not found in the HOL signatures in $Sign_{HOL}$. So when an RSL signature Σ is mapped to a HOL presentation, the abbreviation types of Σ are forgotten.

Let $\Sigma = \langle S, A, \Psi, V \rangle$ and $\Sigma' = \langle S', A', \Psi', V' \rangle$ be mRSL signatures. We should then define a functor ϱ such that $\varrho(\Sigma) = \langle \langle \Omega_{RSL} \cup \Omega_{\Sigma}, C_{RSL} \cup C_{\Sigma} \rangle, E_{RSL} \rangle$ and $\varrho(\Sigma') = \langle \langle \Omega_{RSL} \cup \Omega_{\Sigma'}, C_{RSL} \cup C_{\Sigma'} \rangle, E_{RSL} \rangle$ are HOL presentations.

A signature morphism $\sigma : \Sigma \longrightarrow \Sigma'$ in $Sign_{mRSL}$ should be mapped to a presentation morphism $\varrho(\sigma) : \varrho(\Sigma) \longrightarrow \varrho(\Sigma')$, i.e., a signature morphism with the properties given in Definition 2.23.

When $\sigma = \langle \sigma_S, \sigma_A, \sigma_V \rangle$ should be mapped to a HOL signature-morphism $\langle \sigma_\Omega, \sigma_C \rangle$, the immediate idea is to make σ_Ω correspond to σ_S and make σ_C correspond to σ_V. The mapping σ_Ω should then be from $\Omega_{RSL} \cup \Omega_\Sigma$ to $\Omega_{RSL} \cup \Omega_{\Sigma'}$.

As long as $\sigma_S(s) \in S'$ for all $s \in S$, we can define the mapping σ_Ω based on σ_S: let σ_Ω map Ω_Σ to $\Omega_{\Sigma'}$ by mapping $(s, 0)$ to $(s', 0)$ if $\sigma_S(s) = s' \in S'$. However, if $\sigma_S(s) \in A'$ for an $s \in S$, i.e., σ_S maps a sort in Σ to an abbreviation type in Σ', then the mapping σ_Ω turns out to be partial if we try to define it this way. Unfortunately, the pair $\langle \sigma_\Omega, \sigma_C \rangle$ is not a HOL signature morphism unless both σ_Ω and σ_C are total functions. Hence, the straigthforward attempt to define $\langle \sigma_\Omega, \sigma_C \rangle$ fails.

There are different ways to overcome this problem. By changing the definitions of the institutions $mRSL$ and HOL, it will be possible to define $\varrho(\sigma)$ such that it is a signature morphism.

The institution for mRSL could be changed accordingly, e.g., signature morphisms and sentence morphisms could be ignored. However, we would like to keep the notion of abbreviation types in the institution for mRSL since abbreviation types are part of the language and signature morphisms describe refinement and structured specifications.

Another approach is to change the definition of the institution for HOL such that abbreviation types are included in signatures and signature morphisms. Since Isabelle/HOL does include a notion of abbreviation types, one could argue that it would be natural to include this in the institution for HOL. However, the abbreviation types in Isabelle/HOL are expanded in the internal Isabelle representation, and it has been our intention to consider a HOL institution describing higher-order logic in more general terms than the implementation in Isabelle. The latter has been compromised by making assumptions about additional features in the institution for HOL, cf. Section 5.6.3, but adding these features did not require structural changes to the HOL signatures, as it would be the case when adding abbreviation types.

Instead of changing the definitions of the institutions for mRSL and HOL, we have chosen to introduce the concept of a light institution comorphism which is sufficient to describe the mappings that are needed to translate mRSL specifications to HOL specifications such that a theorem prover for HOL can be reused.

A light institution comorphism only calls for a mapping of mRSL signatures to HOL presentations; mRSL signature morphisms need not be mapped.

6.2 Mapping Sentences

Axioms in mRSL are translated to axioms in Isabelle/HOL, and translating axioms amounts to translating sentences. This translation is carried out by a family α of functions in *Set*. The family is indexed by signatures, i.e., the translation depends on the signature. In the concrete translation, this is recognized by the fact that identifiers (type names and value names) must also be translated.

The translation of sentences is based on a translation of value expressions.

6.2.1 Mapping Type Expressions

Value expressions in mRSL may contain mRSL type expressions which are mapped accordingly to HOL type expressions by the functions in A^{te} from Definition 6.2.

6.2.2 Mapping Value Expressions

We now define a family of functions that map value expressions.

Definition 6.3 Let $\Sigma = \langle S, A, \Psi, V \rangle$ be an mRSL signature, and let $\Sigma \vdash te \rhd st$. Then A_Σ is the function that maps value expressions over Σ to HOL terms over the signature $Sig(\varrho(\Sigma))$ such that value expressions of the static type st are mapped to terms of type $(A^{st}_\Sigma(st))Process$. The definition of A_Σ is as follows:

$A_\Sigma(()) \doteq (Proc\ ())$

$A_\Sigma(\mathbf{true}) \doteq true$

$A_\Sigma(\mathbf{false}) \doteq false$

$A_\Sigma(i) \doteq (Proc\ i)$

$A_\Sigma(r) \doteq (Proc\ r)$

$A_\Sigma(id) \doteq (Proc\ id)$

$A_\Sigma(\mathbf{chaos}) \doteq chaos$

$A_\Sigma(\mathbf{skip}) \doteq skip$

$A_\Sigma((expr_1, expr_2)) \doteq$
$\quad((A_\Sigma(expr_1))\ pipe\ (\lambda\ dv_1^\gamma\ .\ ((A_\Sigma(expr_2))\ pipe\ (\lambda\ dv_2^\delta\ .\ v2p\ ((dv_1, dv_2))))))$

$A_\Sigma(\lambda\ x : te \bullet expr) \doteq (v2p\ (\lambda\ x^{A_\Sigma^{te}(te)}\ .\ A_{\Sigma\dagger(x,st)}(expr)))$

$A_\Sigma(function\_expr\ param\_expr) \doteq appl\ A_\Sigma(function\_expr)\ A_\Sigma(param\_expr)$

$A_\Sigma(\exists\ x : te \bullet expr) \doteq (v2p\ (\exists\ x^{A_\Sigma^{te}(te)}\ .\ A_{\Sigma\dagger(x,st)}(expr) = true))$

$A_\Sigma(\forall\ x : te \bullet expr) \doteq (v2p\ (\forall\ x^{A_\Sigma^{te}(te)}\ .\ A_{\Sigma\dagger(x,st)}(expr) = true))$

$A_\Sigma(expr_1 \equiv expr_2) \doteq (v2p\ ((A_\Sigma(expr_1)) = (A_\Sigma(expr_2))))$

$A_\Sigma((expr)) \doteq (A_\Sigma(expr))$

$A_\Sigma(expr_1\ op\ expr_2) \doteq A_\Sigma(expr_1)\ A_\Sigma^{io}(op, expr_1, expr_2)\ A_\Sigma(expr_2)$

$A_\Sigma(expr_1\ conn\ expr_2) \doteq A_\Sigma(expr_1)\ A_\Sigma^{ic}(conn)\ A_\Sigma(expr_2)$

$A_\Sigma(op\ expr) \doteq A_\Sigma^{po}(op)\ A_\Sigma(expr)$

$A_\Sigma(conn\ expr) \doteq A_\Sigma^{pc}(conn)\ A_\Sigma(expr)$

$A_\Sigma(\textbf{if}\ expr_1\ \textbf{then}\ expr_2\ \textbf{else}\ expr_3\ \textbf{end}) \doteq$
$\quad if\ A_\Sigma(expr_1)\ then\ A_\Sigma(expr_2)\ else\ A_\Sigma(expr_3)\ end.$

The infix connectives are mapped by A_Σ^{ic} as follows:

$A_\Sigma^{ic}(\wedge) \doteq AND$

$A_\Sigma^{ic}(\vee) \doteq OR$

$A_\Sigma^{ic}(\Rightarrow) \doteq IMP.$

The prefix connective is mapped by A_Σ^{pc} as follows:

$A_\Sigma^{pc}(\sim) \doteq NOT.$

The infix operators are mapped by A_Σ^{io} as follows:

$A_\Sigma^{io}(=, expr_1, expr_2) \doteq\ =rsl$

$A_\Sigma^{io}(\neq, expr_1, expr_2) \doteq\ \sim=rsl$

$$A_\Sigma^{io}(<, expr_1, expr_2) \doteq \begin{cases} <rsl & \text{if } \Sigma \vdash expr_1 \rhd s\_int \text{ and } \Sigma \vdash expr_2 \rhd s\_int \\ <rrsl & \text{if } \Sigma \vdash expr_1 \rhd s\_real \text{ and } \Sigma \vdash expr_2 \rhd s\_real \end{cases}$$

$$A_\Sigma^{io}(>, expr_1, expr_2) \doteq \begin{cases} >rsl & \text{if } \Sigma \vdash expr_1 \rhd s\_int \text{ and } \Sigma \vdash expr_2 \rhd s\_int \\ >rrsl & \text{if } \Sigma \vdash expr_1 \rhd s\_real \text{ and } \Sigma \vdash expr_2 \rhd s\_real \end{cases}$$

$$A_\Sigma^{io}(\geq, expr_1, expr_2) \doteq \begin{cases} >=rsl & \text{if } \Sigma \vdash expr_1 \rhd s\_int \text{ and } \Sigma \vdash expr_2 \rhd s\_int \\ >=rrsl & \text{if } \Sigma \vdash expr_1 \rhd s\_real \text{ and } \Sigma \vdash expr_2 \rhd s\_real \end{cases}$$

$$A^{io}_{\Sigma}(\leq, expr_1, expr_2) \doteq \begin{cases} <=rsl & \text{if } \Sigma \vdash expr_1 \rhd s\_int \text{ and } \Sigma \vdash expr_2 \rhd s\_int \\ <=rrsl & \text{if } \Sigma \vdash expr_1 \rhd s\_real \text{ and } \Sigma \vdash expr_2 \rhd s\_real \end{cases}$$

$$A^{io}_{\Sigma}(+, expr_1, expr_2) \doteq \begin{cases} +rsl & \text{if } \Sigma \vdash expr_1 \rhd s\_int \text{ and } \Sigma \vdash expr_2 \rhd s\_int \\ +rrsl & \text{if } \Sigma \vdash expr_1 \rhd s\_real \text{ and } \Sigma \vdash expr_2 \rhd s\_real \end{cases}$$

$$A^{io}_{\Sigma}(-, expr_1, expr_2) \doteq \begin{cases} -rsl & \text{if } \Sigma \vdash expr_1 \rhd s\_int \text{ and } \Sigma \vdash expr_2 \rhd s\_int \\ -rrsl & \text{if } \Sigma \vdash expr_1 \rhd s\_real \text{ and } \Sigma \vdash expr_2 \rhd s\_real \end{cases}$$

$$A^{io}_{\Sigma}(\backslash, expr_1, expr_2) \doteq modrsl$$

$$A^{io}_{\Sigma}(*, expr_1, expr_2) \doteq \begin{cases} *rsl & \text{if } \Sigma \vdash expr_1 \rhd s\_int \text{ and } \Sigma \vdash expr_2 \rhd s\_int \\ *rrsl & \text{if } \Sigma \vdash expr_1 \rhd s\_real \text{ and } \Sigma \vdash expr_2 \rhd s\_real \end{cases}$$

$$A^{io}_{\Sigma}(/, expr_1, expr_2) \doteq divrsl.$$

The prefix operator is mapped by A^{po}_{Σ} as follows:

$$A^{po}_{\Sigma}(\mathbf{abs}) \doteq absrsl.$$

6.2.3 Mapping Sentences

The mapping of sentences is based on the mapping of value expressions. The components of the family $\alpha = (\alpha_\Sigma)_{\Sigma \in |Sign_{mRSL}|}$ of functions are defined below.

Definition 6.4 For a given mRSL signature Σ, the function α_Σ maps mRSL Σ-sentences (equivalence expressions) into HOL $Sig(\varrho(\Sigma))$-sentences as follows:

$$\alpha_\Sigma(ve_1 \equiv ve_2) \doteq A_\Sigma(ve_1) = A_\Sigma(ve_2)$$

where A_Σ is the function from Definition 6.3 that maps mRSL Σ-value expressions into HOL $Sig(\varrho(\Sigma))$-terms that represent the semantics of the value expressions.

6.3 Mapping Models

For a given mRSL signature Σ, the function β_Σ maps HOL $Sig(\varrho(\Sigma))$-models into mRSL Σ-models. In order to map HOL models to mRSL models, a relationship between the semantic domains of mRSL and the universe $\mathcal{U}$ used in HOL must be established.

In Section 6.3.1, the value domains that form the basis for mRSL models are mapped into the universe $\mathcal{U}$ that forms the basis for HOL models. Section 6.3.2 then considers the definition of functions β_Σ that map HOL models to mRSL models.

6.3.1 Mapping Value Domains

The definition of β_Σ is based on a function B that maps types (value domains) to sets in $\mathcal{U}$ and a family of functions D_t that maps values to elements of sets in $\mathcal{U}$, i.e.,

$$B : Types \longrightarrow \mathcal{U} \quad \text{and} \quad D_t : Value_t \longrightarrow B(Value_t) \quad \text{for } t \in T(\emptyset).$$

The functions B and D_t are *injective* and defines a *standard encoding* of RSL types in HOL. We let B^{-1} and D_t^{-1} denote the *left inverses* of B and D_t, respectively.

In the following, we assume a fixed interpretation of the type constant $(Process, 1)$ and the constants $Proc$ and $Chaos$. We let $[\![Process]\!] : \mathcal{U} \longrightarrow \mathcal{U}$ be the function in $\mathcal{U}$ that lifts sets to process domains, cf. Section 6.1.1. Due to the properties of $\mathcal{U}$ such a function exists, and we assume that the function lifts in conformance with the functions $[\![Proc]\!]$ and $[\![Chaos]\!]$.

Definition 6.5 The function $B : Types \longrightarrow \mathcal{U}$ is defined by:

- $B(Value_{s\_bool}) \doteq \mathbf{2}$
- $B(Value_{s\_int}) \doteq [\![int]\!]$
- $B(Value_{s\_real}) \doteq [\![real]\!]$
- $B(Value_{s\_unit}) \doteq \mathbf{1}$
- $B(Value_{s\_prod(t_1,t_2)}) \doteq B(Value_{t_1}) \times B(Value_{t_2})$
- $B(Value_{s\_fct(t_1,t_2)}) \doteq B(Value_{t_1}) \longrightarrow [\![Process]\!](B(Value_{t_2}))$.

Definition 6.6 The family of functions $D = (D_t : Value_t \longrightarrow B(Value_t))_{t \in T(\emptyset)}$ is defined by:

- $D_{s\_bool}(ff) \doteq 0 \in \mathbf{2}$
- $D_{s\_bool}(tt) \doteq 1 \in \mathbf{2}$
- $D_{s\_int}(i) \doteq i \in [\![int]\!]$
- $D_{s\_real}(r) \doteq r \in [\![real]\!]$
- $D_{s\_unit}(d\_unit) \doteq 0 \in \mathbf{1}$
- $D_{s\_prod(t_1,t_2)}((x,y)) \doteq (D_{t_1}(x), D_{t_2}(y))$ for $(x,y) \in Value_{s\_prod(t_1,t_2)}$
- $D_{s\_fct(t_1,t_2)}(f) \in B(Value_{s\_fct(t_1,t_2)})$ is the function F that satisfies $F(D_{t_1}(x)) = D_{Procs_{t_2}}(fx)$ for $f \in Value_{s\_fct(t_1,t_2)}$ and $x \in Value_{t_1}$.

The family of functions $D = (D_{Procs_t} : Procs_t \longrightarrow [\![Process]\!](B(Value_t)))_{t \in T(\emptyset)}$ is defined by:

- $D_{Procs_t}(\perp) \doteq [\![Chaos]\!]$ for $t \in T(\emptyset)$
- $D_{Procs_t}(v2p(x)) \doteq [\![Proc]\!]\, D_t(x)$ for $t \in T(\emptyset)$ and $x \in Value_t$.

6.3.2 Mapping Models

We now define a family $\beta = (\beta_\Sigma)_{\Sigma \in |Sign_{mRSL}|}$ of partial functions that map HOL models to mRSL models.

Definition 6.7 Let $\Sigma = \langle S, A, \Psi, V \rangle$ be an mRSL signature and let $m = \langle m_\Omega, m_C \rangle$ be a $Sig(\varrho(\Sigma))$-model with standard encoding of RSL types so that

- for all $s \in S$ there exists an mRSL type (value domain) vt such that $m_\Omega(s, 0) = B(vt)$,
- for all $v \in V_t$ there exists an element dv in $Value_t$ such that $m_C(v_{A^{st}_\Sigma(t)}) = D_t(dv)$.

Then $\beta_\Sigma(m)$ is defined as $\langle m_S, m_A, m_V \rangle$ where

- $m_S(s) = B^{-1}(m_\Omega(s, 0))$ for $s \in S$
- m_A is defined by m_S and Ψ such that $m_A(a) = m_{T(S)}(\Psi^*(a))$ for $a \in A$
- $m_V = (m_{V_t} : V_t \longrightarrow m_{T(S)}(t))_{t \in T(S)}$ where $m_{V_t}(v) = D^{-1}(m_C(v_{A^{st}_\Sigma(t)}))$ for $t \in T(S)$, $v \in V_t$.

Notice that β_Σ is partial. It is only defined for arguments $m = \langle m_\Omega, m_C \rangle$ for which m_Ω interprets type constants $(s, 0)$, $s \in S$, as sets in the range of B.

Since the HOL type $A^{st}_\Sigma(t)$ does not contain type variables, $m_C(c_{A^{st}_\Sigma(t)})$ is an element in $[\![A^{st}_\Sigma(t)]\!]_m$.

We introduce a lemma regarding the function B and the semantics of type expressions.

Lemma 6.8 *For all mRSL signatures Σ, mRSL Σ-type expressions te, and HOL $Sig(\varrho(\Sigma))$-models m', if $\beta_\Sigma(m')$ is defined, then*

$$B(M(\Sigma)(\beta_\Sigma(m'))(te)) = [\![A^{te}_\Sigma(te)]\!]_{m'}.$$

Proof. By induction over the structure of the type expression te.

For type literals, the result follows easily from the definition of M, Definition 6.5, and Definition 5.18:

$$\begin{aligned}
& B(M(\Sigma)(\beta_\Sigma(m'))(\mathbf{Bool})) \\
= \; & B(Value_{s_bool}) \\
= \; & \mathbf{2} \\
= \; & [\![bool]\!]_{m'} \\
= \; & [\![A^{te}_\Sigma(\mathbf{Bool})]\!]_{m'}
\end{aligned}$$

And similarly for **Int**, **Real**, and **Unit**.

Composite type expressions are either product type expressions or function type expressions.

In the following, let $Value_{t_1}$ and $Value_{t_2}$ be value domains such that

$$M(\Sigma)(\beta_\Sigma(m'))(te_1) = Value_{t_1}$$

and

$$M(\Sigma)(\beta_\Sigma(m'))(te_2) = Value_{t_2}.$$

For binary products:

$$\begin{aligned}
& B(M(\Sigma)(\beta_\Sigma(m'))(te_1 \times te_2)) \\
= \; & B(M(\Sigma)(\beta_\Sigma(m'))(te_1) \times M(\Sigma)(\beta_\Sigma(m'))(te_2)) \\
= \; & B(Value_{t_1} \times Value_{t_2}) \\
= \; & B(Value_{s_prod(t_1,t_2)}) \\
= \; & B(Value_{t_1}) \times B(Value_{t_2}) \text{ by definition of } B \\
= \; & B(M(\Sigma)(\beta_\Sigma(m'))(te_1)) \times B(M(\Sigma)(\beta_\Sigma(m'))(te_2)) \\
= \; & [\![A^{te}_\Sigma(te_1)]\!]_{m'} \times [\![A^{te}_\Sigma(te_2)]\!]_{m'} \text{ by the induction hypothesis} \\
= \; & [\![A^{te}_\Sigma(te_1) \times A^{te}_\Sigma(te_2)]\!]_{m'} \\
= \; & [\![A^{te}_\Sigma(te_1 \times te_2)]\!]_{m'} \text{ by the definition of } A^{te}_\Sigma.
\end{aligned}$$

For function type expressions:

$$\begin{aligned}
& B(M(\Sigma)(\beta_\Sigma(m'))(te_1 \stackrel{\sim}{\rightarrow} te_2)) \\
= \; & B(M(\Sigma)(\beta_\Sigma(m')(te_1)) \longrightarrow Procs(M(\Sigma)(\beta_\Sigma(m'))(te_2))) \\
= \; & B(Value_{t_1} \longrightarrow Procs(Value_{t_2})) \\
= \; & B(Value_{t_1} \longrightarrow Procs_{t_2}) \\
= \; & B(Value_{s_fct(t_1,t_2)}) \\
= \; & B(Value_{t_1}) \longrightarrow [\![Process]\!](B(Value_{t_2})) \\
= \; & B(M(\Sigma)(\beta_\Sigma(m'))(te_1)) \longrightarrow [\![Process]\!](B(M(\Sigma)(\beta_\Sigma(m'))(te_2))) \\
= \; & [\![A^{te}_\Sigma(te_1)]\!]_{m'} \longrightarrow [\![Process]\!]([\![A^{te}_\Sigma(te_2)]\!]_{m'}) \text{ by the induction hypothesis} \\
= \; & [\![A^{te}_\Sigma(te_1) \to (A^{te}_\Sigma(te_2))Process]\!]_{m'} \\
= \; & [\![A^{te}_\Sigma(te_1 \stackrel{\sim}{\rightarrow} te_2)]\!]_{m'}.
\end{aligned}$$

□

In Chapter 4, notions of signature fragments, extensions of signatures, model fragments, and extensions of models were introduced. Similar notions are introduced for HOL signatures and HOL models.

Definition 6.9 Let $\Sigma = \langle \Omega, C \rangle$ be a HOL signature. A Σ-*signature fragment* is a pair (x, τ) where x is the name of a variable and τ is a type in $Types_\Omega$.

We say that (x, τ) is the signature fragment that maps x to τ. The intuitive idea is to consider the name of a variable as a name of a constant; to move x^τ from $VarNames$ to a new x_τ in $Names$, so to speak.

Definition 6.10 Let $\Sigma = \langle \Omega, C \rangle$ be a HOL signature, and let (x, τ) be a Σ-signature fragment. Then $\Sigma \dagger (x, \tau) = \langle \Omega, C \cup (x, \tau) \rangle$.

Model fragments correspond to signature fragments.

Definition 6.11 Let $\Sigma = \langle \Omega, C \rangle$ be a HOL signature, let $\Delta = (x, \tau)$ be a Σ-signature fragment, and let $m = \langle m_\Omega, m_C \rangle$ be a Σ-model. A Δ-fragment of m is a pair (x, dv) where dv is an element of $[\![\tau]\!]_m$.

We say that (x, dv) is the model fragment that interprets x as dv.

Definition 6.12 Let $\Sigma = \langle \Omega, C \rangle$ be a HOL signature, let $\Delta = (x, \tau)$ be a Σ-signature fragment, let $m = \langle m_\Omega, m_C \rangle$ be a Σ-model, and let (x, dv) be a Δ-fragment of m. Then $m \dagger (x, dv) = \langle m_\Omega, m_{C^\dagger} \rangle$ where $m_{C^\dagger}(c_\tau) = m_C(c_\tau)$ for $c_\tau \in C$ and $m_{C^\dagger}(x_\tau) = dv$.

Lemma 6.8 is used for proving the following proposition regarding the function D and the semantics of value expressions:

Proposition 6.13 *For all mRSL signatures Σ, mRSL Σ-value expressions ve, and HOL $Sig(\varrho(\Sigma))$-models m', if $\beta_\Sigma(m')$ is defined, then*

$$D(M(\Sigma)(\beta_\Sigma(m'))(ve)) = [\![A_\Sigma(ve)]\!]_{m'}.$$

Proof. By induction on the structure of ve.

In mRSL, value literals have fixed interpretations that do not depend on the model context. When value literals are translated to HOL, we assume that they also have a standard interpretation. The desired result then follows from definition of M, Definition 6.6 (D), Definition 6.3 (A_Σ), and the definition of $[\![\_]\!]$. We show the details for the Boolean literal **true** and the unit literal ():

- $D(M(\Sigma)(\beta_\Sigma(m'))(\mathbf{true})) = D(v2p(tt)) = [\![Proc]\!]_{m'}(D(tt)) = [\![Proc]\!]_{m'}(1)$
  $= [\![Proc]\!]_{m'}([\![True]\!]_{m'}) = [\![(Proc\ True)]\!]_{m'} = [\![true]\!]_{m'} = [\![A_\Sigma(\mathbf{true})]\!]_{m'}$

- $D(M(\Sigma)(\beta_\Sigma(m'))(())) = D(v2p(d\_unit)) = [\![Proc]\!]_{m'}(D(d\_unit))$
  $= [\![Proc]\!]_{m'}(0) = [\![Proc]\!]_{m'}([\![()]\!]_{m'}) = [\![(Proc\ ())]\!]_{m'} = [\![A_\Sigma(())]\!]_{m'}$.

The details for the rest of the literals are similar.

For identifiers of values, $v \in V_t$:

$$
\begin{aligned}
& D(M(\Sigma)(\beta_\Sigma(m'))(v)) \\
= \; & D(v2p((\beta_\Sigma(m'))_{V_t}(v))) \\
= \; & [\![Proc]\!]_{m'}\ D((\beta_\Sigma(m'))_{V_t}(v)) \\
= \; & [\![Proc]\!]_{m'}\ D(D^{-1}(m'_C(v_{A^{st}_\Sigma(t)}))) \\
= \; & [\![Proc]\!]_{m'}\ (m'_C(v_{A^{st}_\Sigma(t)})) \\
= \; & [\![(Proc\ v)]\!]_{m'} \\
= \; & [\![A_\Sigma(v)]\!]_{m'}.
\end{aligned}
$$

If $\beta_\Sigma(m')$ is defined, then $\beta_\Sigma(m')$ is a Σ-model such that $(\beta_\Sigma(m'))_{V_t}(v)$ is defined; and since $m'_C(v_{A^{st}_\Sigma(t)})$ is in the range of D, it is the case that $D(D^{-1}(m'_C(v_{A^{st}_\Sigma(t)}))) = m'_C(v_{A^{st}_\Sigma(t)})$.

For basic value expressions **chaos** and **skip**:

- $D(M(\Sigma)(\beta_\Sigma(m'))(\mathbf{chaos})) = D(\bot) = [\![Chaos]\!]_{m'} = [\![A_\Sigma(\mathbf{chaos})]\!]_{m'}$
- $D(M(\Sigma)(\beta_\Sigma(m'))(\mathbf{skip})) = D(v2p(d\_unit)) = [\![Proc]\!]_{m'}(D(d\_unit))$
  $= [\![Proc]\!]_{m'}(0) = [\![Proc]\!]_{m'}([\![()]\!]_{m'}) = [\![(Proc\ ())]\!]_{m'} = [\![skip]\!]_{m'} = [\![A_\Sigma(\mathbf{skip})]\!]_{m'}$.

For function expressions (assuming that $\Sigma \vdash te \rhd st$):

$$
\begin{aligned}
& D(M(\Sigma)(\beta_\Sigma(m'))(\lambda\ x : te \bullet ve) \\
= \; & D(v2p(\lambda dv \in M(\Sigma)(\beta_\Sigma(m'))(te).M(\Sigma \dagger (x, st))(\beta_\Sigma(m') \dagger (x, dv))(ve))) \\
= \; & [\![Proc]\!]_{m'}\ D((\lambda dv \in M(\Sigma)(\beta_\Sigma(m'))(te).M(\Sigma \dagger (x, st))(\beta_\Sigma(m') \dagger (x, dv))(ve))) \\
= \; & [\![Proc]\!]_{m'}\ (\lambda dv' \in B(M(\Sigma)(\beta_\Sigma(m'))(te)). \\
& \qquad\qquad D(M(\Sigma \dagger (x, st))(\beta_\Sigma(m') \dagger (x, D^{-1}(dv')))(ve))) \\
= \; & [\![Proc]\!]_{m'}\ (\lambda dv' \in [\![A^{te}_\Sigma(te)]\!]_{m'}.[\![A_{\Sigma\dagger(x,st)}(ve)]\!]_{m'\dagger(x,dv')}) \text{ by Lemma 6.8 and the} \\
& \text{induction hypothesis} \\
= \; & [\![Proc]\!]_{m'}\ (\lambda dv' \in [\![A^{te}_\Sigma(te)]\!]_{m'}.[\![(x^{A^{te}_\Sigma(te)}).A_{\Sigma\dagger(x,st)}(ve)]\!]_{m'}(dv')) \\
= \; & [\![Proc]\!]_{m'}\ [\![(\lambda x^{A^{te}_\Sigma(te)}.A_{\Sigma\dagger(x,st)}(ve)]\!]_{m'} \\
= \; & [\![v2p(\lambda x^{A^{te}_\Sigma(te)}.A_{\Sigma\dagger(x,st)}(ve))]\!]_{m'} \\
= \; & [\![A_\Sigma(\lambda\ x : te \bullet ve)]\!]_{m'}.
\end{aligned}
$$

Likewise for quantified expressions.

A composite value expression is of the general form $\mathcal{F}_{mRSL}(ve_1, \ldots, ve_n)$ where $\mathcal{F}_{mRSL}$ is the mRSL syntax, and ve_i, $1 \leq i \leq n$, are value expressions. F_{mRSL} denotes the mRSL semantics that corresponds to $\mathcal{F}_{mRSL}$. This is essentially a function. F_{HOL} is the semantics of the HOL term that corresponds to the mRSL semantics. $\mathcal{F}_{HOL}$ is the HOL syntax, i.e., the HOL term that corresponds to the mRSL semantics.

$$\begin{aligned}
& D(M(\Sigma)(\beta_\Sigma(m'))(\mathcal{F}_{mRSL}(ve_1, \ldots, ve_n))) \\
= \ & D(F_{mRSL}(M(\Sigma)(\beta_\Sigma(m'))(ve_1), \ldots, M(\Sigma)(\beta_\Sigma(m'))(ve_n))) \\
= \ & F_{HOL}(D(M(\Sigma)(\beta_\Sigma(m'))(ve_1)), \ldots, D(M(\Sigma)(\beta_\Sigma(m'))(ve_n))) \\
= \ & F_{HOL}([\![A_\Sigma(ve_1)]\!]_{m'}, \ldots, [\![A_\Sigma(ve_n)]\!]_{m'}) \text{ by the induction hypothesis} \\
= \ & [\![\mathcal{F}_{HOL}(A_\Sigma(ve_1), \ldots, A_\Sigma(ve_n))]\!]_{m'} \text{ by the compositional definition of the HOL semantics} \\
= \ & [\![A_\Sigma(\mathcal{F}_{mRSL}(ve_1, \ldots, ve_n))]\!]_{m'} \text{ by the definition of } A_\Sigma.
\end{aligned}$$

□

In order to use Theorem 2.30, the functions β_Σ (where $\Sigma \in |Sign_{mRSL}|$) are required to be surjective. This requirement is met:

Proposition 6.14 *The function β_Σ, from Definition 6.7, is surjective for $\Sigma \in |Sign_{mRSL}|$.*

Proof. For each mRSL signature $\Sigma = \langle S, A, \Psi, V\rangle$ and mRSL Σ-model $m = \langle m_S, m_A, m_V\rangle$, there is a HOL $Sig(\varrho(\Sigma))$-model $m' = \langle m'_\Omega, m'_C\rangle$ such that $m'_\Omega(s, 0) = B(m_S(s))$ for $s \in S$ and $m'_C(v_{A^{st}_\Sigma(t)}) = D(m_{V_t}(v))$ for $v \in V_t$, $t \in T(S)$. Since $\beta_\Sigma(m') = m$, the mapping β_Σ is surjective. □

In Chapter 2, we introduced the concept of looser semantics together with an equivalence result regarding semantic consequence of a specification with respect to loose semantics and looser semantics (Theorem 2.21).

Although the semantics of RSL specifications is best described by the concept of looser semantics, we have chosen to assume loose semantics of mRSL specification in this chapter. Theorem 2.21 justifies this decision: the set of sentences that are semantic consequences of a specification SP with respect to loose semantics equals the set of sentences that are semantic consequences of SP with respect to looser semantics.

As shown above by Definition 6.7 and Proposition 6.14, it is possible to define a surjective mapping β_Σ of HOL models of $Sig(\varrho(\Sigma))$ to mRSL models of Σ.

If mRSL specifications had been assumed to have looser semantics, this should have been considered in the reusing theorem (Theorem 2.30). Then β_Σ should not only map to Σ-models but to Θ-models were Θ represents all possible extensions of Σ. Since HOL models only interpret the names of constants that occur in the signature and not all possible extensions of the signature, a mapping to RSL models with extended signatures would involve a fixed, chosen interpretation of the entities extending the signature. However, such a mapping would not be surjective.

6.4 The Representation Condition

We can now state and prove the representation condition for the light institution comorphism.

Proposition 6.15 *For all mRSL signatures Σ, HOL models m of $\varrho(\Sigma)$, and mRSL Σ-sentences φ:*

$$m \models_{Sig(\varrho(\Sigma))} \alpha_\Sigma(\varphi) \qquad \textit{iff} \qquad \beta_\Sigma(m) \models_\Sigma \varphi.$$

Proof. The sentence φ is of the form $ve_1 \equiv ve_2$. Then by definitions of $\models$, Proposition 6.13, and injectivity of the function D:

$$\begin{aligned}
& m \models_{Sig(\varrho(\Sigma))} \alpha_\Sigma(\varphi) \\
\text{iff } & m \models_{Sig(\varrho(\Sigma))} \alpha_\Sigma(ve_1 \equiv ve_2) \\
\text{iff } & [\![A_\Sigma(ve_1)]\!]_m = [\![A_\Sigma(ve_2)]\!]_m \\
\text{iff } & D(M(\Sigma)(\beta_\Sigma(m))(ve_1)) = D(M(\Sigma)(\beta_\Sigma(m))(ve_2)) \\
\text{iff } & M(\Sigma)(\beta_\Sigma(m))(ve_1) = M(\Sigma)(\beta_\Sigma(m))(ve_2) \\
\text{iff } & \beta_\Sigma(m) \models_\Sigma \varphi.
\end{aligned}$$

□

6.5 The Light Institution Comorphism from $mRSL$ to HOL

The definitions from the previous sections are summarized in the definition of a light institution comorphism from $mRSL$ to HOL.

Definition 6.16 Let $\varrho = \langle \varrho, \alpha, \beta \rangle$ be the light institution comorphism from $mRSL$ to HOL such that

- ϱ is the function that maps mRSL signatures to HOL presentations as described in Section 6.1

- $\alpha = (\alpha_\Sigma)_{\Sigma \in |Sign_{mRSL}|}$ is the family of functions from Definition 6.4
- $\beta = (\beta_\Sigma)_{\Sigma \in |Sign_{mRSL}|}$ is the family of functions from Definition 6.7.

Proposition 6.15 states that the representation condition is satisfied.

6.6 Applying the Reusing Theorem

By Proposition 6.14, Theorem 2.30, and soundness of Isabelle/HOL, we can use the light institution comorphism $\langle \varrho, \alpha, \beta \rangle$ to translate mRSL specifications SP_{mRSL} and proof obligations P_{mRSL} to HOL and reuse Isabelle/HOL as illustrated below:

$$
\begin{tikzcd}
mRSL & & HOL \\
SP_{mRSL} \overset{?}{\models} P_{mRSL} \arrow[rr, "\varrho, \alpha"] & & SP_{HOL} \overset{?}{\models} P_{HOL} \arrow[d, "\text{Isabelle}"'] \\
SP_{mRSL} \models P_{mRSL} & & Yes \arrow[ll, "\text{Reusing theorem}"]
\end{tikzcd}
$$

Example 6.1 touched upon translating mRSL specifications to Isabelle/HOL theories. In the following, it is described how the light institution comorphism is used to translate basic specifications and declarations in mRSL to theories and declarations in Isabelle/HOL.

6.6.1 Translating Specifications

Only basic (i.e., not structured) specifications are considered in mRSL. In other words, we restrict our attention to specifications of the form:

scheme A =
  class
    ...
  end

where the ellipsis symbolizes a sequence of declarations.

The Isabelle/HOL theory `RSL` contains all the definitions needed to facilitate the translation. Essentially, it is the Isabelle/HOL formulation of the HOL presentation $\langle\langle \Omega_{RSL}, C_{RSL}\rangle, E_{RSL}\rangle$, i.e., it contains declarations for the fixed contributions. The Isabelle/HOL theory `RSL` is found in Appendix B, and as shown in Example 6.1, it is used as a stub such that the result of translating an mRSL scheme is an Isabelle/HOL theory of the form:

```
theory A = RSL :
  ...
end
```

where the ellipsis symbolizes a sequence of declarations.

In RSL, the order of the declarations in a specification does not matter. However, this is not the case in Isabelle/HOL where entities must be defined before they are used. Consequently, declarations are translated in the following order: (i) type declarations, (ii) value declarations, (iii) axiom declarations.

In the following sections, it is explained how the three kinds of declarations are translated.

6.6.2 Translating Type Declarations

Two kinds of type declarations are considered in mRSL: declarations of abbreviation types and declarations of sort types.

Since the order of declarations is important in Isabelle but not in RSL, the type declarations of a specification must be translated with care. Consider the following type declarations:

> **type** T = type_expr
> **type** s

where T is introduced as another name for the type denoted by the type expression type_expr, and s is an abstract sort type.

If s occurs in type_expr, the sort type declaration must be translated before the abbreviation type declaration. Consequently, declarations of sort types are always translated first. The abbreviation type declarations are then translated in a loop: if the right-hand side only contains names of types that are already translated (or are built-in), then the declaration is translated. Since abbreviation type declarations are assumed not to be cyclic, this process of translation terminates.

The type declarations above are translated to:

```
typedecl s
types T = type_expr
```

where `type_expr` is the result of translating the mRSL type expression as prescribed by the function A^{te}_{Σ} defined in Definition 6.2, where Σ is the signature of the considered specification.

Notice that $A^{te} = (A^{te}_{\Sigma})_{\Sigma \in |Sign_{mRSL}|}$ is a family of functions indexed by mRSL-signatures, i.e., the translation of a type expression depends on the signature since the signature contains information about declared types.

Example 6.2 *An mRSL specification with declarations of types for representing colours, e.g., on a computer screen:*

scheme $RGB =$
class
type
Colour $=$ *Red* $\times$ *Green* $\times$ *Blue,*
Red $=$ *Intensity,*
Green $=$ *Intensity,*
Blue $=$ *Intensity,*
Intensity
end

translates to the Isabelle/HOL theory:

```
theory RGB = RSL :
  typedecl Intensity
  types Red = Intensity
        Green = Intensity
        Blue = Intensity
        Colour = "Red * Green * Blue"
end
```

6.6.3 Translating Value Declarations

A value declaration of the form

value id : type_expr

is translated to a declaration of a constant:

```
consts id :: "type_expr"
```

where `type_expr` is the result of translating type_expr as prescribed by the function A^{te}_{Σ} defined in Definition 6.2, where Σ is the signature of the considered specification.

Explicit function definitions are expanded to value declarations and axiom declarations and then translated. This is further discussed in Section 7.1.3.

6.6.4 Translating Axiom Declarations

An axiom declaration consists of a named equivalence expression:

axiom
[axiom_id] value_expr_1 $\equiv$ value_expr_2

This is translated to:

```
axioms
  axiom_id : "value_expr_1 = value_expr_2"
```

where `value_expr_1` and `value_expr_2` are the results of translating the value expressions value_expr_1 and value_expr_2, respectively, as prescribed by the function A_{Σ} from Definition 6.3, where Σ is the signature of the considered specification.

6.7 Discussion

First, we briefly discuss a technical issue regarding the interpretations of Isabelle/HOL theories that are results of translating mRSL specifications. Then, the encoding of C_{RSL}, Ω_{RSL}, and E_{RSL} in Isabelle/HOL is discussed. Finally, expanding mRSL is discussed.

6.7.1 Restrictions on Interpretations

The mRSL signatures are mapped to HOL presentations to ensure certain interpretations of the built-in mRSL operators. In principle, it is also needed to ensure a certain interpretation of sort types, and means of constraining the interpretation of abstract types are therefore needed.

An abstract sort type should be constrained in such a way that it could only be implemented using the built-in mRSL types, the product type, and the function and process type-constructors (i.e., `'a => 'b Process`). However, the mapping of mRSL signatures to HOL presentations does not impose these restrictions. Fortunately, too many interpretations of abstract types in Isabelle/HOL does not interfere with the soundness of the light institution comorphism. The mapping β_Σ becomes partial since it cannot account for unintended interpretations of sort types, but it is still surjective which is the property that is needed to ensure sound use of Isabelle/HOL.

6.7.2 HOL Presentations and Isabelle/HOL Specifications

The light institution comorphism $\langle \varrho, \alpha, \beta \rangle$ serves as a formal foundation for translating mRSL specifications to Isabelle/HOL specifications.

When using the light institution comorphism for translation, the fixed contributions described in Section 6.1.1 are placed in an Isabelle/HOL specification `RSL` which is found in Appendix B. The encoding of the fixed contributions in `RSL` is not a direct encoding of C_{RSL}, Ω_{RSL}, and E_{RSL}; it contains the same elements, but the exact formulation of the declarations follows Isabelle/HOL tradition.

In the Isabelle/HOL specification, the type constant $(Process, 1)$, the constants $Proc$ and $Chaos$, and the defining axioms in E_{RSL} are represented by a datatype declaration:

```
datatype 'a Process = Proc 'a | Chaos
```

The reason for using the datatype package in Isabelle/HOL for this is that it provides better proof support than a direct encoding.

Constants are defined using definitions in Isabelle/HOL. Likewise, this is for better proof support. The difference between the sentences in E_{RSL} shown in Section 6.1.1 and the definitions in Appendix B is that the universally quantified variables in the sentences are replaced by free variables in the definitions. Moreover, the definitions are equalities at the Isabelle meta-level.

6.7.3 Expanding mRSL

Translating mRSL specifications to Isabelle/HOL is also described in [38] which not only accounts for the translation based on the mapping of sentences described in Section 6.2, but also discusses possibilities for expanding the considered subset of RSL.

As mentioned in Section 3.2.2, a value expression denotes a function from a store to a process. However, attempts to represent RSL value expressions as functions in Isabelle/HOL had little success when it came to proof support.

When translating from mRSL to Isabelle/HOL the approach taken has been to encode the mRSL semantics in Isabelle/HOL. So when the semantics of mRSL (or RSL) is complex, then the encoding in Isabelle/HOL will also be complex. One may choose not to see this complexity as a problem, but on the other hand see it as a good reason for using a proof assistant instead of pen and paper. However, if the proofs in Isabelle/HOL become even more difficult and time-consuming than their counterparts in RSL, we have not gained anything by translating to Isabelle/HOL.

One attempt to remedy the complexity inherited from the semantics is to translate an mRSL value expression ve to a relatively simple Isabelle/HOL term that is equivalent to the direct encoding of ve in Isabelle/HOL. An advantage of this approach is that the equivalence between the Isabelle/HOL terms can be proved using Isabelle/HOL. In fact, we used this approach when defining the translation A_Σ on quantified expressions – Isabelle/HOL was used to prove equivalence between the translation shown by A_Σ in Definition 6.3 and a direct encoding of the semantics from Section 4.4.2.

If the RSL datatypes for sets and lists should be included in mRSL, it should be investigated to which extent the set theory and lists already found in Isabelle/HOL can be used. The denotational semantics of RSL list expressions is expressed using maps, and it is therefore desirable to find a simpler representation in Isabelle/HOL than the direct encoding.

CHAPTER 7

Applications

This chapter reports on experiences with the use of the light institution comorphism presented in Chapter 6 for translating mRSL specifications and proof obligations to HOL specifications and proof goals, and subsequently using Isabelle for proving the proof goals.

First, we discuss general issues and experiences regarding the use of the light institution comorphism and Isabelle/HOL. Then, three applications are considered: logic circuits, a generalized railway crossing, and an encoding of Duration Calculus in RSL.

The applications are not intended to be full-fledged case studies of formal developments, but to serve as examples that illustrate strengths and weaknesses of this approach to proof support and show how proof support plays a role in the RAISE development method.

7.1 General Considerations

In this section, we shall discuss some general issues concerning the use of Isabelle/HOL for applications.

We discuss Isabelle proof scripts, encoding RSL proof rules in Isabelle/HOL, explicit function definitions in mRSL, and a theorem due to the use of the semantic operator $v2p$. Moreover, issues regarding terminating functions are considered before Isabelle/HOL is compared with the eden editor.

7.1.1 Development of Isabelle Proofs

In Chapter 5, we briefly introduced Isabelle and Isabelle/HOL. Based on experiences from the applications, we now comment on the style of the proof scripts and the development process.

Proof Scripts

Isabelle is a tactical theorem prover in which a proof is in the form of a script – a sequence of applications of tactics that transform the goal that is to be proved.

In classical Isabelle, proof scripts are separate ML files, and the tactics are ML functions that change the proof state. Isabelle/Isar provides a new scripting language for structuring proofs in a way that makes them more readable. In Isar, specifications and proofs are mixed, i.e., goals and proofs may be placed inside a specification. In Section 5.6.4, we saw how to establish proof goals and apply rules and methods in an Isabelle/HOL specification.

Although we used the Isar language for the applications to be discussed in this chapter, the proof scripts turned out to be rather unstructured, and they could just as well have been classical Isabelle proof scripts. If it had been our intention to develop structured proof scripts, e.g., on the basis of an informal pen-and-paper proof, we could have used the original RSL proof tools. Our intention, however, was to explore the use of the Isabelle simplifier for making proofs with a high degree of automation.

Simplification

The Isabelle simplifier rewrites a proof goal by substituting equals for equals. It uses a set of rules (a simplifier set) which may be modified by the user. Hence, we can add axioms that define relevant functions to the simplifier set. When developing Isabelle proofs (especially unstructured proof scripts), designing the appropriate simplifier sets is an important task. If the simplifier set contains rules that can rewrite a goal in a cyclic way, the simplifier may not terminate (it depends on the goal).

When the simplifier was used for rewriting a goal in the proofs for the applications described later in this chapter, the axioms for the simplifier set were selected by inspecting the goal. Axioms that defined the constants occuring in the goal were added to the simplifier set when the simplifier was applied. This analysis and selection was done manually, but it would be an interesting future task to mechanize it.

7.1.2 Encoding RSL Proof Rules

RSL has an associated proof system with a large set of proof rules. We have translated some of the proof rules to Isabelle/HOL and proved them as theorems. The purpose was two-fold: when the translated proof rules have been proved as theorems/lemmas of Isabelle/HOL, they may be used in the further development of proofs; and proving the proof rules as theorems shows that the RSL proof system is sound with respect to the denotational semantics (since the translation essentially is a direct encoding of the denotational semantics). The rules are listed in Appendix B.

The RSL proof rules fall into different categorizations: inference rules and equivalence rules. Moreover, proof rules may have side-conditions. In the following, the encoding of inference rules, equivalence rules, and side-conditions is considered.

An account of the proof rules implemented in the RAISE tools (the eden editor) from CRI [28] is given in [21]. A proof system for RSL is also presented in [55].

Inference Rules

Inference rules are used to replace a goal with a series of subgoals (new goals to be proved). Generally, inference rules are of the form:

$$\frac{S_1, \ldots, S_n}{C}.$$

The inference rule reads: if the premises S_1 through S_n are true, then the conclusion C is true. Consequently, the conclusion can be proved by proving the premises.

RSL inference rules are encoded as Isabelle meta-level implications of the form:

```
[| S_1 = true; ...; S_n = true |] ==> C = true.
```

When encoding an inference rule, the RSL value expression C of type **Bool** becomes an Isabelle/HOL term `C` of type `bool Process`. To take `C` from the "process level" to the "logic level", we compare with `true`. Likewise for the premises S_1 through S_n. The translation A_Σ from Definition 6.3 describes how the mRSL value expressions are represented as HOL terms which are easily expressed in Isabelle/HOL syntax.

Example 7.1 *The inference rule* and_split_inf*:*

$$\frac{eb, \quad eb'}{eb \wedge eb'}$$

is encoded as

```
[| eb = true; eb' = true |] ==> (eb AND eb') = true
```

where `eb` *and* `eb'` *are Isabelle/HOL terms corresponding to the respective mRSL value expressions.*

The inference rule in the above example is known in natural deduction systems as $\wedge$-introduction.

Equivalence Rules

Equivalence rules are of the form:

$$expr_1 \simeq expr_2$$

and are used to substitute $expr_2$ for $expr_1$ (or $expr_1$ for $expr_2$). The translation A_Σ of value expressions (see Definition 6.3) is used to translate RSL equivalence rules to Isabelle equalities of the form:

$$A_\Sigma(expr_1) = A_\Sigma(expr_2).$$

Example 7.2 *The equivalence rule* if_not2*:*

$$\textbf{if } eb \textbf{ then } {\sim}eb' \textbf{ else } {\sim}eb'' \textbf{ end} \simeq {\sim}\textbf{if } eb \textbf{ then } eb' \textbf{ else } eb'' \textbf{ end}$$

is encoded as

```
  if eb then (NOT eb') else (NOT eb'') end
= NOT (if eb then eb' else eb'' end)
```

which is easily proved by case-analysis (`eb`*,* `eb'`*, and* `eb''`*) and simplification.*

Side-Conditions

RSL proof rules may have side-conditions which, e.g., require expressions to be convergent.

Example 7.3 *The rule* if_idempotence3*:*

$$\textbf{if } eb \textbf{ then } e \textbf{ else } e \textbf{ end} \simeq e$$

has the side-condition **when readonly**$(eb) \wedge$ **convergent**(eb)*. Since imperative aspects of RSL are disregarded in mRSL, the expression eb will trivially be read-only. The side-condition regarding convergence may be expressed by representing eb as* `(Proc eb)` *in the translated rule:*

```
if (Proc eb) then e else e end = e
```

or by introducing a premise:

```
(eb = Proc b) ==> (if eb then e else e end = e).
```

Side-conditions regarding imperative aspects of RSL (variables and channels) are not relevant for mRSL, and are therefore ignored as done in the above example. Only convergence is considered.

In an inference rule of the form

$$\frac{S_1, \ldots, S_n}{C}$$

it is understood that the conclusion C is true if all premises S_1 through S_n are true. Hence, the premises S_1 through S_n are implicitly required to be convergent. This is also reflected in the encoding of the proof rule in Isabelle/HOL:

```
[| S_1 = true; ...; S_n = true |] ==> C = true.
```

A premise cannot equal `true` unless it is convergent.

7.1.3 Function Definitions

In the description of mRSL in Chapter 4, value declarations and axiom declarations were kept apart. In full RSL, however, *explicit function definitions* are often used to combine the declaration of a value with an axiom that defines the value.

An example of an explicit function definition is:

value
  $\mathrm{f} : \mathbf{Int} \overset{\sim}{\rightarrow} \mathbf{Int}$
  $\mathrm{f(x)} \equiv \mathrm{x} + 4$

which is a shorthand for a value declaration and an axiom declaration:

value
  $\mathrm{f} : \mathbf{Int} \overset{\sim}{\rightarrow} \mathbf{Int}$
axiom
  $[\mathrm{f\_def}]\ \forall\ \mathrm{x} : \mathbf{Int} \bullet \mathrm{f(x)} \equiv \mathrm{x} + 4.$

Translating the value declaration and the axiom to Isabelle/HOL yields:

```
consts
 f :: "int => (int Process)"
axioms
 f_def : "v2p(ALL x :: int . (v2p((appl (Proc f) (Proc x))
                                  = ((Proc x) +rsl (Proc 4))))
                              = true)
          = true".
```

The essence of the definition is wrapped in applications of `v2p`, a universal quantifier, and the applications of `appl` and `Proc`. As a consequence, the axiom `f_def` cannot be used directly to rewrite the term `(Proc f)`.

The function `f` may alternatively be specified by the following axiom:

axiom
  $[\mathrm{f\_def}]\ \mathrm{f} \equiv (\lambda\ \mathrm{x} : \mathbf{Int} \bullet \mathrm{x} + 4)$

which translates to:

```
axioms
 f_def : "(Proc f) = (v2p(% x :: int . ((Proc x) +rsl (Proc 4))))".
```

This axiom can be used by the simplifier to rewrite the term `(Proc f)`. Furthermore, applications of λ-abstractions are automatically removed by β-conversion.

Simplification may take us to a goal in which the function `f` occurs (not "`(Proc f)`" but "`f`"). To handle this situation, a lemma of the following form may be useful:

```
lemma
 f_def_lemma : "f == (% x :: int . ((Proc x) +rsl (Proc 4)))".
```

Lemmas of this form may be derived systematically from the function definitions and are easily proved using the defining axioms (`f_def` in the above case). Both the original and the derived form may be useful in a proof.

7.1.4 Auxiliary Theorems

In the encoding of mRSL semantics in Isabelle/HOL, a free datatype was used for representing a simplified notion of processes such that the HOL term `Proc v` represents a process that terminates with the result value `v`. At the same time, the function `v2p` is used to lift a value `w` to the process with the result value `w`. The motivation for using `v2p` was to make it easier to expand the subset later on by modifying the encoding of the process domain together with the definition of `v2p`. However, proofs could have been simpler if we had used the constructor `Proc` instead of the function `v2p` in the translation and written `Proc x` instead of `v2p x`.

The following theorem relates `Proc` and `v2p` and turned out to be useful in proofs:

```
theorem Proc_v2p_thm : "((Proc x) = (v2p y)) ==> (x = y)".
```

7.1.5 Terminating Functions

In full RSL, there is a type operator for total functions (functions that terminate deterministically for all values of the arguments). In mRSL, on the other hand, there is no distinct type operator for total functions – there is one type operator for (potentially) partial functions. Nevertheless, total functions are considered in some applications and proofs may rely on the termination of the functions involved. Consequently, we may need to explicitly state that a function terminates for all values of its arguments.

We deal with different situations:

- A function may be declared as a value (in a value declaration) and implicitly defined by one or more axioms. In this case, an axiom may be used to specify that the function is in fact total.

- A function may be declared as a value and defined by an axiom that explicitly defines the function as it was shown in the previous section. In this case, a lemma may express that the function is total. Such a lemma can be proved using the axiom that explicitly defines the function (provided that the function does terminate for all argument values).

- A function may be a bound variable (e.g., in a quantified expression). In this case, the nice solution is to rephrase proof goals. However, a pragmatic solution in which an axiom assumes termination for all functions of the relevant type may be considered. This is not without problems, and we shall return to this issue in Section 7.4 where we consider the third application.

7.1.6 Comparing Isabelle/HOL and eden

The Isabelle proof assistant and the eden justification editor take different approaches to making proofs.

Isabelle provides a scripting language making it possible to rerun proofs (proof scripts). The eden justification editor saves proofs in a binary format and it is not possible to rerun proofs.

The eden editor, however, has some user-friendly features:

- The user can select (highlight) a subexpression in the current goal and rewrite it by applying the eden simplifier or various equivalence rules. The editor can list the rules that are applicable for the selection – a feature of that makes it easier for non-experienced users to develop proofs.

- The proofs contain the intermediate goals – this usually makes proofs more readable, but displaying all intermediate goals may make proofs very long, decreasing the readability. However, it is possible to hide intermediate goals such that only the important intermediate goals are shown.

The eden editor would benefit from a scripting language. This should at least make it possible to rerun proofs, and perhaps also make it possible to express proof strategies. Moreover, a higher degree of automation in the eden editor is desired.

Comparing eden and Isabelle, the latter gives us a valuable scripting language and a powerful simplifier. However, developing the proofs in Isabelle/HOL for the applications shown in this chapter required knowledge and training with respect to using Isabelle (and the ProofGeneral user-interface [2]).

7.2 Combinational Logic Circuits

As our first application example, we consider the specification of simple logic circuits built from basic logic gates. Basic logic gates and two circuits are introduced and described in mRSL specifications. Then, the circuits are proved equivalent and the proof support is discussed.

7.2.1 Logic Gates and Circuits

We consider circuits built from the basic gates depicted in Figure 7.1.

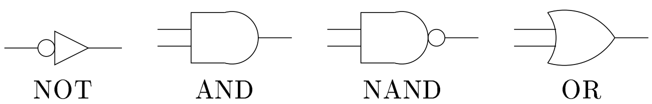

Figure 7.1: Basic logic gates.

The functions that are implemented by the gates are defined in the truth tables in Figure 7.2.

A circuit consists of a set of connected gates. Two examples of circuits are shown in Figure 7.4 and Figure 7.5.

NOT

0	1
1	0

AND

	0	1
0	0	0
1	0	1

NAND

	0	1
0	1	1
1	1	0

OR

	0	1
0	0	1
1	1	1

Figure 7.2: Truth tables for basic logic gates.

	0	1
0	0	1
1	1	0

XOR

Figure 7.3: Truth table for the exclusive-or function.

The two circuits are taken from [63], and in the following, we shall verify that they both implement an exclusive-or (XOR) function with two inputs. The truth table in Figure 7.3 defines the 2-input XOR function.

7.2.2 Specification

Logic gates and circuits are now described in mRSL.

Electrical Signals

An electrical signal is assumed to be either high or low and it may therefore be represented by a Boolean value. This leads to the following declaration of a type S for electrical signals:

type
  S = **Bool**

This is the usual digital abstraction.

Logic Gates

The logical gates are then specified. For each kind of gate, we declare a value that represents the function implemented by that particular kind of gate. Axioms that define these values are also declared:

value
  andgate : $\mathrm{S} \xrightarrow{\sim} \mathrm{S} \xrightarrow{\sim} \mathrm{S}$,
  orgate : $\mathrm{S} \xrightarrow{\sim} \mathrm{S} \xrightarrow{\sim} \mathrm{S}$,
  notgate : $\mathrm{S} \xrightarrow{\sim} \mathrm{S}$,

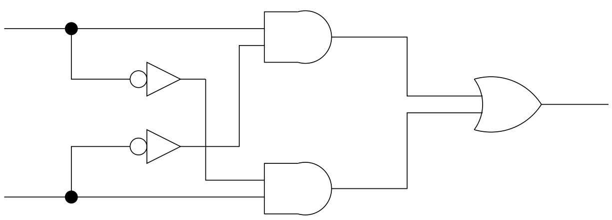

Figure 7.4: Multigate design for the 2-input XOR function: AND-OR.

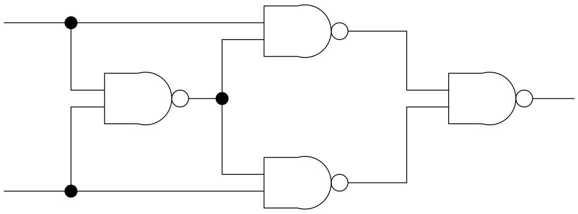

Figure 7.5: Multigate design for the 2-input XOR function: three-level NAND.

nandgate : $S \xrightarrow{\sim} S \xrightarrow{\sim} S$
axiom
[andgate_def] andgate $\equiv (\lambda\ s1 : S \bullet (\lambda\ s2 : S \bullet s1 \wedge s2))$,
[orgate_def] orgate $\equiv (\lambda\ s1 : S \bullet (\lambda\ s2 : S \bullet s1 \vee s2))$,
[notgate_def] notgate $\equiv (\lambda\ s : S \bullet \sim s)$,
[nandgate_def] nandgate $\equiv (\lambda\ s1 : S \bullet (\lambda\ s2 : S \bullet \sim (s1 \wedge s2)))$

The built-in connectives are used, and the values are defined by axioms written in the style previously discussed in Section 7.1.3.

Circuits

Values that represent the functions implemented by the circuits shown in Figure 7.4 and Figure 7.5 are declared and defined with axioms:

value
cir1 : $S \xrightarrow{\sim} S \xrightarrow{\sim} S$,
cir2 : $S \xrightarrow{\sim} S \xrightarrow{\sim} S$
axiom
[cir1_def]
cir1 $\equiv$
(λ s1 : S $\bullet$
(λ s2 : S $\bullet$ orgate
(andgate(s1)(notgate(s2)))
(andgate(notgate(s1))(s2))

```
        )
      ),
    [cir2_def]
      cir2 ≡
        (λ s1 : S •
          (λ s2 : S • nandgate
                          (nandgate(s1)(nandgate(s1)(s2)))
                          (nandgate(nandgate(s1)(s2))(s2))
          )
        )
```

An occurrence of a gate in a circuit is represented by an application of the corresponding function. The input signals are the arguments, and the result value of the application expression is the single output-signal.

7.2.3 Proving Equivalence

The two circuits are claimed to implement the same function. This claim is expressed by the following condition:

$$(\forall\ s1 : S \bullet (\forall\ s2 : S \bullet (cir1(s1)(s2) = cir2(s1)(s2))))$$

which becomes our proof obligation.

To prove that the circuits truly implement the same function, the specification of gates and circuits and the proof obligation are translated to Isabelle/HOL, after which the proof obligation is proved using Isabelle.

Translating the proof obligation yields:

```
theorem
 cir1_cir2 :
  "(v2p (ALL s1 :: S .
     ((v2p (ALL s2 :: S .
       (((appl (appl (Proc cir1) (Proc s1)) (Proc s2))
         =rsl (appl (appl (Proc cir2) (Proc s1)) (Proc s2)))
       = true)))
     = true)))
   = (true)"
```

which is proved by a single application of the simplifier:

```
  apply (simp add : appl_def infix_appl_def infix_conn_appl_def
    prefix_appl_def prefix_appl_def prefix_conn_appl_def false_def
    true_def v2p_def RSL_eq_def RSLeq_def RSL_not_def RSLnot_def
    RSL_and_def RSLand_def RSL_or_def RSLor_def cir1_def andgate_def
    orgate_def notgate_def nandgate_def cir1_def cir2_def true_def
    false_def)
```

Notice how the definitions of constants for expressing mRSL in Isabelle/HOL are added to the set of rules used by the simplifier together with the axioms that specify the gates and the circuits.

7.2.4 Discussion

The simplifier turned out to be very effective for this application. However, the circuits were small, and only a small number of rules were added to the simplifier set.

The proof obligation was also proved using the eden justification editor. When the mRSL specification shown above was used, the proof contained 30 manual applications of rules. The proof was carried out by applying the axioms that defined the functions and using the simplifier in eden – each function application that modelled to a two-input gate required two applications of proof rules to bring it into a form that could be handled by the simplifier in eden.

The mRSL specification was also reformulated as an RSL specification in which the functions modelling gates were defined by explicit function definitions. Since the explicit function definitions effect generation of rules for function applications, the proof obligation could then be proved by 19 manual applications of rules.

Increasing the number of gates in a circuit will, of course, increase the length of a justification in the eden editor. The Isabelle/HOL proof script that invokes the simplifier, however, is of the same length.

In the specification, the built-in type for Booleans was used to model electrical signals and the built-in connectives were used to describe the functionality of the gates. Since RSL deals with non-termination, it is fair to ask if this may be used to model undefined signals. Here, we must observe that function application is strict due to the semantics of RSL. Even though we can specify functions that do not depend on all inputs and do not need to be strict, they will always act as strict functions when applied.

7.3 Generalized Railway Crossing

We shall now consider a generalized railway crossing. The problem of designing a controller for a railway crossing has been considered in [9, 34].

After a brief introduction to the problem, an initial specification is presented. The initial specification is refined in two steps, and proof support is discussed.

7.3.1 The Railway Crossing Problem

A railway track crosses a road. We assume that the railway track is used in only one direction. Before the crossing, there is a signal which is either set or cleared. If the signal is set, trains are not allowed to pass it. If it is cleared, trains may pass the signal and enter the crossing. There is a gate at the crossing. The crossing is set, when the gate is closed such that road traffic does not enter the crossing. The crossing is cleared, when the gate is open and road traffic may enter the crossing. We abstract from the fact that it may take some time for the gate to close – we consider it closed when it starts closing.

We present a discrete model of the control mechanisms at the railway crossing, disregarding real-time properties. Although we have simplified the problem, it serves the purpose of illustrating the specification of a state-transition system using the RAISE development method.

7.3.2 Initial Specification

The railway crossing is now described in an initial, abstract, algebraic RSL specification in which the railway crossing is modelled as a state-transition system. We present the declarations of the specification step by step, and we shall later refer to the whole specification as $SP1$.

An abstract sort type for representing the global state of the system is declared:

type State

Two pieces of state information are considered in the specification: the state of the signal and the state of the crossing. Booleans are used to denote whether the signal and the crossing are set or cleared.

An *observer* is a function that selects a piece of state information. The following observers are declared:

value
 /* observers */
 signal : State $\stackrel{\sim}{\rightarrow}$ **Bool**,
 crossing : State $\stackrel{\sim}{\rightarrow}$ **Bool**

A *generator* is a function that changes state information – in the applicative setting, it yields a new state. Generators for setting and clearing the signal and the crossing are declared:

value
 /* generators */
 setsignal : State $\stackrel{\sim}{\rightarrow}$ State,
 setcrossing : State $\stackrel{\sim}{\rightarrow}$ State,
 clearsignal : State $\stackrel{\sim}{\rightarrow}$ State,
 clearcrossing : State $\stackrel{\sim}{\rightarrow}$ State

The observers and generators are specified by a series of axioms combining each observer with each generator. The axioms specify how the observable state information is changed by the state generators. The axioms are as follows:

axiom
 [signal_setsignal]

$(\forall$ s : State • signal(setsignal(s))) $\equiv$ **true**,
[signal_setcrossing]
$(\forall$ s : State • signal(setcrossing(s)) = signal(s)) $\equiv$ **true**,
[signal_clearsignal]
$(\forall$ s : State • signal(clearsignal(s)) = **false**) $\equiv$ **true**,
[signal_clearcrossing]
$(\forall$ s : State • signal(clearcrossing(s)) = signal(s)) $\equiv$ **true**,
[crossing_setsignal]
$(\forall$ s : State • crossing(setsignal(s)) = crossing(s)) $\equiv$ **true**,
[crossing_setcrossing]
$(\forall$ s : State • crossing(setcrossing(s))) $\equiv$ **true**,
[crossing_clearsignal]
$(\forall$ s : State • crossing(clearsignal(s)) = crossing(s)) $\equiv$ **true**,
[crossing_clearcrossing]
$(\forall$ s : State • crossing(clearcrossing(s)) = **false**) $\equiv$ **true**

The idea behind these specifications is to provide a domain model that can be used to capture the safety-requirements of a controller. To avoid dangerous situations, either the signal or the crossing must be set. In other words, the signal and the crossing must not be cleared at the same time. The safety-requirements are formalized as a derived observer, i.e., a state-predicate:

value
/* derived observer */
safe : State $\xrightarrow{\sim}$ **Bool**

axiom
[safe_def]
safe $\equiv$ (λ s : State • signal(s) $\vee$ crossing(s))

To keep the railway crossing safe, its state must only be changed when certain conditions are present. A *guard* is a state-predicate that checks if the conditions for a safe state change (modelled by a generator) are present. Guards for the generators are declared:

value
/* guards */
can_clearsignal : State $\xrightarrow{\sim}$ **Bool**,
can_clearcrossing : State $\xrightarrow{\sim}$ **Bool**

The following axioms express that safety is preserved by state changes when the conditions modelled by the guards are present:

axiom
[safe_clearsignal]
$(\forall$ s : State •

safe(s) ∧ can_clearsignal(s) ⇒
safe(clearsignal(s))) ≡
true,
[safe_clearcrossing]
(∀ s : State •
safe(s) ∧ can_clearcrossing(s) ⇒
safe(clearcrossing(s))) ≡
true

These axioms specify requirements to the guards.

7.3.3 Refining the Specification

We now present a model-oriented specification that refines the initial specification. The declarations of the specification are presented individually, and we shall later refer to the whole specification as $SP2$.

In the model-oriented specification, the global state is represented by an abbreviation type:

type State = **Bool** × **Bool**

For a state (b_1, b_2), the first component b_1 is true if and only if the signal is set, and the second component b_2 is true if and only if the crossing is set.

This leads to the following definition of the observers:

value
/* observers */
signal : State $\xrightarrow{\sim}$ **Bool**,
crossing : State $\xrightarrow{\sim}$ **Bool**
axiom
[signal_def]
signal ≡ (λ (s1, s2) : State • s1),
[crossing_def]
crossing ≡ (λ (s1, s2) : State • s2)

Usually, observers, guards, and generators would be specified by explicit function definitions at this stage. However, separate declarations of values and axioms are used here to conform with mRSL. The functions are specified with axioms using λ-abstractions (cf. the discussion of explicit function definitions in Section 7.1.3).

The generators are defined as follows:

value
/* generators */
setsignal : State $\xrightarrow{\sim}$ State,
setcrossing : State $\xrightarrow{\sim}$ State,
clearsignal : State $\xrightarrow{\sim}$ State,

clearcrossing : State $\xrightarrow{\sim}$ State
axiom
[setsignal_def]
setsignal ≡ (λ (s1, s2) : State • (**true**, s2)),
[setcrossing_def]
setcrossing ≡ (λ (s1, s2) : State • (s1, **true**)),
[clearsignal_def]
clearsignal ≡ (λ (s1, s2) : State • (**false**, s2)),
[clearcrossing_def]
clearcrossing ≡ (λ (s1, s2) : State • (s1, **false**)),

The definition of the derived observer $safe$ is the same as in the initial specification:

value
/* derived observer */
safe : State $\xrightarrow{\sim}$ **Bool**
axiom
[safe_def]
safe ≡ (λ s : State • signal(s) ∨ crossing(s))

The guards are given explicit definitions as follows:

value
/* guards */
can_clearsignal : State $\xrightarrow{\sim}$ **Bool**,
can_clearcrossing : State $\xrightarrow{\sim}$ **Bool**
axiom
[can_clearsignal_def]
can_clearsignal ≡ (λ s : State • crossing(s)),
[can_clearcrossing_def]
can_clearcrossing ≡ (λ s : State • signal(s))

They express that the signal can be cleared when the crossing is set, and that the crossing can be cleared when the signal is set.

The model-oriented specification, $SP2$, statically implements the initial specification, $SP1$, since it declares the same types and functions. Notice that the sort type State in SP1 is implemented by the abbreviation type State in SP2.

The axioms of SP2 are different from those of SP1 – they provide explicit definitions of the observers, generators, and guards. To prove that SP2 is a refinement of SP1, the axioms of SP1 must be proved as theorems of SP2. To do so, we translate SP2 into an Isabelle/HOL theory, and translate the axioms of SP1 into proof goals in Isabelle/HOL.

Proving the axioms of SP1 in the context of SP2 was done by simplification. Definitions of the relevant functions and constants used to express RSL constructs in Isabelle/HOL were added to the simplifier set and the proofs were easily carried out.

As an example, the axiom signal_setsignal was translated to a proof goal which was proved by the following:

```
theorem
 signal_setsignal_dev :
   "(v2p (ALL s :: State .
        (appl (Proc signal) (appl (Proc setsignal) (Proc s)))
      = true))
    = true"
 apply (simp add : signal_def crossing_def setsignal_def
   setcrossing_def)
 apply (simp add : v2p_def true_def appl_def)
done
```

The first application of the simplifier with the definitions for the observers and generators rewrites the goal to:

```
 1. v2p (ALL a b.
            appl (v2p (%s1s2. Proc (fst s1s2)))
             (appl (v2p (%s1s2. true pipe (%dv1. v2p (dv1, snd s1s2))))
               (Proc (a, b))) =
            true) =
    true
```

This is proved by a second application of the simplifier with the definitions of v2p, true, and appl.

The next observer-generator axiom was also proved by simplification:

```
theorem
 signal_setcrossing_dev :
   "(v2p (ALL s :: State .
        ((appl (Proc signal) (appl (Proc setcrossing) (Proc s)))
        =rsl (appl (Proc signal) (Proc s)))
      = true))
    = true"
 apply (simp add : signal_def crossing_def setsignal_def
   setcrossing_def)
 apply (simp add : v2p_def true_def appl_def RSL_eq_def
   infix_appl_def RSLeq_def)
done
```

Compared to the previous proof script, the occurrence of the RSL equality operator `=rsl` required that more axioms were used by the simplifier.

All the axioms of SP1 were proved in the context of SP2 by simplification as out-lined in the above examples. Thus, we proved that SP2 implements SP1.

7.3.4 Extending the Specification

By proving the axioms safe_clearsignal and safe_clearcrossing, we prove that safe is an invariant of the system. Whether it is possible to prove this depends on the definitions of the guards and safe. We were able to prove it for this application, but for some systems (see, e.g., [39, 30]), it is not possible to prove safe to be an invariant. Instead, another predicate, consistent, is proved to be an invariant, and it is proved that consistent implies safe[1].

To illustrate the introduction of such a state-predicate which is more restrictive (i.e., holds for less states) than safe, we extend the specification with the following declarations:

value
  consistent : State $\xrightarrow{\sim}$ **Bool**
axiom
  [consistent_def]
    consistent ≡
      (λ s : State •
          (signal(s) ∧ ∼ crossing(s)) ∨
          (∼ signal(s) ∧ crossing(s)))

A consistent state must also be a safe state. The following axiom expresses that consistency implies safety:

axiom
  [consistent_safe]
    (∀ s : State • consistent(s) ⇒ safe(s)) ≡ **true**

This is translated to the following goal:

```
theorem
 consistent_safe :
   "(v2p (ALL s :: State .
        ((appl (Proc consistent) (Proc s))
          IMP (appl (Proc safe) (Proc s)))
      = true))
    = true"
```

Proving this goal involves simplification, case-splitting, and use of the classical reasoner.

Case-splitting is done based on the results of signal and crossing in the sense that a result is either successful termination with a value or divergence (chaos). It must be assured that both signal and crossing are total functions. This property is easily derived from the specifications thanks to the explicit definitions of the functions.

[1]In this example, consistent is not an invariant – using one of the generators to change a consistent state yields a state that is not consistent. The predicate consistent is only introduced for illustrative purposes.

7.3.5 Discussion

Proof of the refinement relation between SP1 and SP2 was supported by the translation and Isabelle/HOL. Proving the axioms of SP1 in the context of SP2 was surprisingly easy – simplification was sufficient.

Using the eden editor to prove that SP2 refines SP1, 101 manual applications of proof rules were used to prove 11 proof obligations.

For this example, our translation and Isabelle indeed provided a basis for proof support with at high degree of automation. Furthermore, the proofs for this application only needed little knowledge of Isabelle/HOL.

Only the features of mRSL were available for this example, and a few points about datatypes should be made:

- The variant and union definitions are not part of mRSL. Although it was possible to formulate SP2 using a product, the first choice for experienced users of RSL is most likely a variant definition.

- The variant and union definitions in RSL are short ways of writing more elaborate algebraic specifications. Since it is possible to write algebraic specifications in mRSL, variant and union definitions may be expressed in mRSL by their elaborate specifications. However, the elaborate specifications are expected to become difficult to read and reason about.

- Since free datatypes have been incorporated in Isabelle/HOL, it would be possible to translate variant types for RSL to free datatypes in Isabelle/HOL. However, this deviates from the approach of using institutions and the semantic differences must be taken into account.

  In [17], RSL variant types are translated directly to datatypes in PVS. In some cases, this may cause a consistent RSL specification to be translated into an inconsistent PVS specification – this problem is accounted for by issuing a warning when the situation may occur.

7.4 DC/RJ$^-$

As our third application, we consider an encoding of Duration Calculus (DC) in RSL ([65] which is based on original ideas of Haxthausen; see also [31]).

DC is a formalism for design of real-time systems (systems that must meet timing constraints). A brief introduction to DC is given in the following, and we refer to [14, 13, 29] for a thorough introduction.

We adapt the RSL specifications in [65] to mRSL so that we can use the light institution comorphism to translate the specifications to Isabelle/HOL and use Isabelle to prove theorems of DC. Our objective is to show how a set of given RSL specifications can be adapted to mRSL such that Isabelle/HOL can be used, rather than providing proof support for DC.

After a brief introduction to DC, the encoding in RSL/mRSL is presented and a series of theorems are considered. The section ends with a discussion.

7.4.1 Duration Calculus

In this section, a brief introduction to DC is given, and encoding DC in RSL, as well as incorporating real time in RSL, is briefly touched upon.

Duration Calculus

DC is an interval (temporal) logic, i.e., a logic for time intervals.

In DC, points in time are represented by real numbers, and the truth value of a formula is relative to an interval of time-points. System states and events are represented by Boolean-valued functions.

In the following, $Time$ is the type of time-points, *Bool* is the type of Booleans, and *Real* is the type of reals.

A *Boolean state model* of a real-time system consists of a set of functions,

$$P_i : Time \rightarrow Bool$$

where P_i is a *characteristic function* of a specific aspect of the system behaviour.

Some aspects of a system are best described by real-valued functions over time which are usually denoted *state expressions*:

$$R_i : Time \rightarrow Real.$$

Time intervals are represented by their end points, and the set of time intervals is:

$$Interval = \{[b, e] \mid b \in Time \wedge e \in Time \wedge b \leq e\}.$$

The notion of a *state duration* is used in DC to specify behaviour of real-time systems. The duration of a Boolean state over a time interval is the accumulation of the time in the interval in which the state has been present. In other words, the duration of a Boolean state represented by the function $P : Time \rightarrow Bool$ over the interval $[b, e]$ is the integral

$$\int_b^e P(t)\mathrm{d}t.$$

In interval logics, properties may be expressed as functions of time intervals, known as *interval variables*:

$$v_i : Interval \rightarrow Real.$$

The *length* of an interval is an interval variable denoted l:

$$l : Interval \rightarrow Real.$$

Furthermore, the duration of a state P is an interval variable:

$$\int P : Interval \rightarrow Real.$$

Duration Calculus in RSL

The DC/RJ$^-$ system described in [65] is an encoding of axioms, inference rules, and theorems of interval logic and DC in RSL. This makes it possible to use the eden justification editor for verifying properties of DC specifications. As an example, the verification of a gas burner is considered in [65] – an existing proof is ported to the RSL encoding.

The RSL specifications in [65] provided us with a suite of RSL specifications that we transformed into mRSL specifications which we translated to Isabelle/HOL specifications. A number of DC theorems were also translated to goals in Isabelle/HOL and proved using Isabelle.

Duration Calculus and RSL combined

RSL (as described in [54] and [43]) has no features dedicated to modelling real-time properties, but work on providing means of including real-time properties in RSL specifications has been done.

The purpose of [31, 65] is two-fold. Not only does DC/RJ$^-$ provide a proof tool for DC, i.e., the eden justification editor can be used for making DC proofs. DC/RJ$^-$ also provides abstract syntax for DC formulas in RSL such that DC formulas, and thereby real-time properties, can be included in RSL specifications.

It should be mentioned that RSL has been extended to Timed RSL (TRSL) which includes real-time features. TRSL is described in [22, 58]; the syntax is RSL with an additional *wait* expression of the form **wait** r, specifying a delay of r seconds.

In the following, only DC/RJ$^-$ is considered.

7.4.2 RSL Specifications

The RSL specifications encoding DC are now presented and it is explained how they are adapted to mRSL. Overloading is resolved manually as the RSL specifications are adapted to mRSL.

Duration Calculus in RSL

The representation of time-points and intervals of time-points in RSL is straightforward (from [65]):

$$\begin{array}{l}\textbf{type}\ \mathrm{Time} = \textbf{Real},\\ \qquad \mathrm{Interval} = \{|\ (\mathrm{t1}, \mathrm{t2}) : \mathrm{Time} \times \mathrm{Time} \bullet \mathrm{t1} \leq \mathrm{t2}\ |\}\end{array}$$

However, subtypes are not part of mRSL. The type declarations are adapted to mRSL by ignoring subtypes:

type Time = **Real**,
  Interval = Time × Time

Consequently, an Interval is a pair (r_1, r_2) of reals. The restriction $r_1 \leq r_2$ is not imposed. However, this property may be needed, as we shall see in the proof of A0 in Section 7.4.3, in which case it must be explicitly assumed.

Characteristic functions of aspects of system behaviour, state assertions, are represented by (from [65]):

type StateAssertion = {| e : Time → **Bool** • is_finitely_variable(e) |},
  StateExpression = {| e : Time → **Real** • is_finitely_variable(e) |}

This is adapted to mRSL:

type StateAssertion = Time $\stackrel{\sim}{\rightarrow}$ **Bool**,
  StateExpression = Time $\stackrel{\sim}{\rightarrow}$ **Real**

We only consider finitely variable functions, i.e., functions that only have finitely many transitions within a finite time period. This is, however, not enforced in the mRSL specifications.

Basic state assertions and operations on state assertions are defined in mRSL as in [65] except that the total function arrow → has been replaced by $\stackrel{\sim}{\rightarrow}$:

value
  /∗ state assertion constructors ∗/
  tt : StateAssertion = (λ t : Time • **true**),

  ff : StateAssertion = (λ t : Time • **false**),

  not : StateAssertion $\stackrel{\sim}{\rightarrow}$ StateAssertion
  not(e) ≡ (λ t : Time • ∼ e(t)),

  or : StateAssertion × StateAssertion $\stackrel{\sim}{\rightarrow}$ StateAssertion
  or(e1, e2) ≡ (λ t : Time • e1(t) ∨ e2(t)),

  and : StateAssertion × StateAssertion $\stackrel{\sim}{\rightarrow}$ StateAssertion
  and(e1, e2) ≡ (λ t : Time • e1(t) ∧ e2(t)),

  implies : StateAssertion × StateAssertion $\stackrel{\sim}{\rightarrow}$ StateAssertion
  implies(e1, e2) ≡ (λ t : Time • e1(t) ⇒ e2(t)),

  = : StateExpression × StateExpression $\stackrel{\sim}{\rightarrow}$ StateAssertion
  se1 = se2 ≡ (λ t : Time • se1(t) = se2(t))

Real numbers may be lifted to state expressions by the operator lift (from [65]):

$$
\begin{array}{l}
\textbf{value} \\
\quad \text{lift} : \textbf{Real} \to \text{StateExpression} \\
\quad \text{lift(r)} \equiv (\lambda\ \text{t} : \text{Time} \bullet \text{r})
\end{array}
$$

The definition is adapted to:

$$
\begin{array}{l}
\textbf{value}\ \text{lift} : \textbf{Real} \xrightarrow{\sim} \text{StateExpression} \\
\textbf{axiom}\ [\text{lift\_def}] \\
\quad \text{lift} \equiv (\lambda\ \text{r} : \textbf{Real} \bullet (\lambda\ \text{t} : \text{Time} \bullet \text{r}))
\end{array}
$$

Likewise, an operator lifts functions on real numbers to functions on state expressions (from [65]):

$$
\begin{array}{l}
\textbf{value} \\
\quad \text{lift} : (\textbf{Real} \times \textbf{Real} \to \textbf{Real}) \to \\
\qquad\qquad (\text{StateExpression} \times \text{StateExpression} \to \text{StateExpression}) \\
\quad \text{lift(f)(se1, se2)} \equiv (\lambda\ \text{t} : \text{Time} \bullet \text{f(se1(t), se2(t))})
\end{array}
$$

which is easily adapted to mRSL.

Some standard operations on state expressions are defined. One of them is addition with the following definition in mRSL:

$$
\begin{array}{l}
\textbf{value} \\
\quad \text{DCplus} : \text{StateExpression} \times \text{StateExpression} \xrightarrow{\sim} \text{StateExpression} \\
\quad \text{DCplus(se1, se2)} \equiv (\lambda\ \text{t} : \text{Time} \bullet \text{se1(t)} + \text{se2(t)})
\end{array}
$$

Operators for substraction and multiplication are defined in a similar way.

In [65], state variables are introduced as terms:

$$
\textbf{type}\ \text{Term} = \text{Interval} \xrightarrow{\sim} \textbf{Real}
$$

The type definition is accompanied by declarations of a term giving the length of an interval, operators for lifting reals and functions, and operators for manipulating terms (from [65]):

$$
\begin{array}{l}
\textbf{value} \\
\quad \text{length} : \text{Term} = (\lambda\ \text{(t1, t2)} : \text{Interval} \bullet \text{t2} - \text{t1}), \\
\\
\quad \text{lift} : \textbf{Real} \to \text{Term} \\
\quad \text{lift(r)} \equiv (\lambda\ \text{i} : \text{Interval} \bullet \text{r}), \\
\\
\quad \text{lift} : (\textbf{Real} \times \textbf{Real} \to \textbf{Real}) \to (\text{Term} \times \text{Term} \to \text{Term}) \\
\quad \text{lift(f)(te1, te2)} \equiv \\
\qquad (\lambda\ \text{(t1, t2)} : \text{Time} \times \text{Time} \bullet \text{f(te1(t1, t2), te2(t1, t2))}), \\
\\
\quad + : \text{Term} \times \text{Term} \to \text{Term} \\
\quad \text{te1} + \text{te2} \equiv (\lambda\ \text{i} : \text{Interval} \bullet \text{te1(i)} + \text{te2(i)}),
\end{array}
$$

$- : \text{Term} \times \text{Term} \rightarrow \text{Term}$
$\text{te1} - \text{te2} \equiv (\lambda\ \text{i} : \text{Interval} \bullet \text{te1(i)} - \text{te2(i)}),$

$* : \text{Term} \times \text{Term} \rightarrow \text{Term}$
$\text{te1} * \text{te2} \equiv (\lambda\ \text{i} : \text{Interval} \bullet \text{te1(i)} * \text{te2(i)})$

Moreover, formulas are introduced:

type Formula = Interval $\stackrel{\sim}{\rightarrow}$ **Bool**

A series of formula constructors and operators are defined. Some of them are shown here (from [65]):

value
/*DC formula constructors */
tt : Formula = (λ i : Interval • **true**),

ff : Formula = (λ i : Interval • **false**),

lift : (**Real** × **Real** → **Bool**) → (Term × Term → Formula)
lift(f)(te1, te2) ≡ (λ i : Interval • f(te1(i), te2(i))),

$= : \text{Term} \times \text{Term} \rightarrow \text{Formula}$
$\text{te1} = \text{te2} \equiv (\lambda\ \text{i} : \text{Interval} \bullet \text{te1(i)} = \text{te2(i)}),$

$\geq : \text{Term} \times \text{Term} \rightarrow \text{Formula}$
$\text{te1} \geq \text{te2} \equiv (\lambda\ \text{i} : \text{Interval} \bullet \text{te1(i)} \geq \text{te2(i)}),$

not : Formula → Formula
not(f) ≡ (λ i : Interval • $\sim$ f(i)),

or : Formula × Formula → Formula
or(f1, f2) ≡ (λ i : Interval • f1(i) $\vee$ f2(i)),

and : Formula × Formula → Formula
and(f1, f2) ≡ (λ i : Interval • f1(i) $\wedge$ f2(i)),

implies : Formula × Formula → Formula
implies(f1, f2) ≡ (λ i : Interval • f1(i) $\Rightarrow$ f2(i)),

$\frown$: Formula × Formula → Formula
f1 $\frown$ f2 ≡
(
λ (t1, t2) : Interval •

```
            ∃ t : Time • t1 ≤ t ∧ t ≤ t2 ∧ f1(t1, t) ∧ f2(t, t2)
      ),

   hold : Formula → Bool
   hold(f) ≡ (∀ i : Interval • f(i))
```

In [65], the duration operator is specified implicitly by a signature and axioms. The signature is:

```
value dur : StateAssertion → Term
```

And the axioms are:

```
axiom
   [DC_A1] (dur(tt) = length) ≡ tt,

   [DC_A2] (dur(ff) = lift(0.0)) ≡ tt,

   [DC_A3] ∀ e : StateAssertion • (dur(e) ≥ lift(0.0)) ≡ tt,

   [DC_A4]
      ∀ e1, e2 : StateAssertion •
         (dur(e1) + dur(e2) = dur(or(e1, e2)) + dur(and(e1, e2))) ≡ tt,

   [DC_A5]
      ∀ x, y : Real, e : StateAssertion •
         (
            implies
               (
                  (dur(e) = lift(x)) ⌢ (dur(e) = lift(y)), dur(e) = lift(x + y)
               )
         ) ≡
         tt,

   [DC_A6]
      ∀ e1, e2 : StateAssertion •
         shold(and(implies(e1, e2), implies(e2, e1))) ⇒
            hold(dur(e1) = dur(e2))
```

where shold is defined by:

```
value
   shold : StateAssertion → Bool
   shold(e) ≡ (∀ t : Time • e(t))
```

The duration of a state assertion a is a term dur(a) that given an interval i yields the accumulated time in i for which a has been true.

These axioms constitute an encoding of the proof system for DC given in [14].

7.4.3 Proving Properties

We translated and proved a selection of DC theorems and an axiom of DC.

Theorem AND_ASSOC

The DC theorem AND_ASSOC expresses that conjunction between formulas is associative. Here it is shown as an RSL proof goal with the scheme DC as context:

[AND_ASSOC]
  in DC ⊢
    ∀ F1, F2, F3 : Formula •
      and(and(F1,F2),F3) ≡ and(F1,and(F2,F3))

The proof of the theorem was conducted by a relatively short proof script (8 lines) in Isabelle/HOL. The strategy was to unfold/expand everything and then simplify. However, it required some ingenuity – the key was to use the HOL rule `ext` (see Section 5.6.2) and case-splitting on the possible outcomes of applying F1 and F2. This is a good example of how one may need in-depth knowledge about the target theorem prover when translating between logical systems.

Using the RSL specifications presented in Section 7.4.2 and the eden justification editor, AND_ASSOC was proved by 7 manual applications of proof rules.

Theorem AND_DIS_OR2

The theorem AND_DIS_OR2:

[AND_DIS_OR2]
  in DC ⊢
    ∀ F1, F2, F3 : Formula •
      and(F1,or(F2,F3)) ≡ or(and(F1,F2),and(F1,F3))

has a structure similar to that of AND_ASSOC. Consequently, the proof of the theorem was conducted by a proof script which essentially was identical to the one used for proving AND_ASSOC.

Using the RSL specifications from Section 7.4.2 and the eden justification editor, AND_DIS_OR2 was proved by 10 manual applications of proof rules.

Theorem OR_IMPLY

The proof of

[OR_IMPLY]
  in DC ⊢
    ∀ F1, F2, F3 : Formula •
      implies(or(F1,F2),F3) ≡ and(implies(F1,F3),implies(F2,F3))

relied on the fact that F2 and F3 are total functions (i.e., they terminate for any argument values).

Formulas are total functions, but when the RSL specifications of DC/RJ$^-$ are adapted to mRSL, this information is lost. Consequently, the non-terminating case must also be considered when case-splitting is conducted on the results of applying a formula to an interval.

When developing a proof of OR_IMPLY, we end up having three subgoals that are implications in which it is assumed in the premises that a formula does not terminate.

Since we want formulas to be terminating, we may add the following axiom which makes it easy to prove the remaining subgoals:

```
axioms
 DCFormula_convergence : "(F::DCFormula) (x, y) ~= Chaos"
```

The axiom states that a formula always terminates. Strictly speaking, adding this axiom is unsound. Type abbreviations are expanded in Isabelle/HOL, so the axiom states that all functions (of a certain type) return a value different from Chaos. This is of course not true, but since we only consider functions of that certain type when they denote DC formulas, we can argue that adding the axiom temporarily is acceptable when we only use it for discarding the subgoals that consider divergent formulae.

The spotless solution, however, is to rephrase the goal as an implication: if the functions F1, F2, and F3 terminate for all argument values, then the value expression implies(or(F1,F2),F3) is equivalent to and(implies(F1,F2),implies(F2,F3)).

If F1, F2, and F3 had been values instead of bounded variables, their termination could have been assumed by axioms regarding the specific DCFormulas:

```
axioms
 F1_convergent : "F1 (x, y) ~= Chaos"
 F2_convergent : "F2 (x, y) ~= Chaos"
 F3_convergent : "F3 (x, y) ~= Chaos"
```

Adding these axioms would not have caused any controversies with respect to soundness.

The Isabelle/HOL proof script used to prove OR_IMPLY was 17 lines long, while an RSL proof required only 10 manual applications of proof rules in the eden justification editor.

Theorem HOLD_AND

The theorem HOLD_AND,

[HOLD_AND]
  in DC $\vdash$
    $\forall$ F1, F2 : Formula •
      hold(and(F1,F2)) $\equiv$ (hold(F1) $\wedge$ hold(F2)),

was also proved by a relatively short proof script (14 lines). During the proof, simplification introduced existential quantifiers which were removed by instantiation, i.e., the rule `exI` was used to replace the bounded variables in the quantifications with entities defined outside the quantifications. Consequently, knowledge of proof techniques in Isabelle/HOL turned out to be, if not crucial, then very useful.

Using the eden justification editor, HOLD_AND was proved by 10 manual applications of proof rules.

Theorem IMPLY_DBL

The theorem IMPLY_DBL:

[IMPLY_DBL]
  in DC ⊢
    ∀ F1, F2, F3 : Formula •
      implies(and(F1,F2),F3) ≡ implies(F1,implies(F2,F3))

was easily proved by simplification and case-splitting.

The Isabelle/HOL proof script was 7 lines long, while an RSL proof required 9 manual applications of proof rules.

Axiom A0

Proving axioms of DC ensures that the encoding of DC in RSL conforms with the axiomatic formulation.

An axiom of DC expresses that lengths of intervals are non-negative:

$$\models l \geq 0$$

This is expressed in mRSL as

hold(length ≥ lift(0.0))

which is translated to the following proof obligation in Isabelle/HOL:

```
theorem
 A0_thm :
   "(appl (Proc DChold) (appl (Proc DCgteq) ((Proc DClength)
      pipe (% dv1 . ((appl (Proc DClift3) (Proc 0))
             pipe (% dv2 . v2p ((dv1,dv2))))))))
    = true"
```

Proving this theorem is problematic since it relies on a property of intervals: the left endpoint is less than or equal to the right endpoint. When we adapted the encoding of DC to mRSL, we represented the subtype Interval with the binary product of reals. Consequently, the subtype property of intervals is not available in the proof. It is not possible to conduct the proof without this assumption/property.

However, we may modify the encoding of the function *hold* such that it is taken into account that the subtype of intervals is considered. Essentially, this means that a universal quantification

$$\forall\ x : T \bullet P(x)$$

over a subtype

$$T = \{|\ t : T' \bullet Q(t)\ |\}$$

is represented as

$$\forall\ x : T' \bullet Q(x) \Rightarrow P(x)$$

The definition of hold may be modified when it is translated to Isabelle/HOL such that its representation in Isabelle/HOL becomes:

```
consts
 DChold1 :: "DCFormula => (bool Process)"
axioms
 DChold1_def :
  "(Proc DChold1) =
     (v2p (% f :: DCFormula .
       (v2p (ALL i :: DCInterval .
         ((fst i) <= (snd i))
           --> ((appl (Proc f) (Proc i)) = true)))))"
```

where the Isabelle/HOL implication `-->` is used. Alternatively, we can modify the definition and translate it the usual way:

```
consts
 DChold2 :: "DCFormula => (bool Process)"
axioms
 DChold2_def :
   "(Proc DChold2) =
      (v2p (% f :: DCFormula .
        (v2p (ALL i :: DCInterval .
            (((v2p (fst i)) <=rrsl (v2p (snd i)))
              IMP (appl (Proc f) (Proc i)))
          = true))))"
```

where `IMP` is the result of translating an mRSL implication ⇒. In both cases, the theorem A0 is now easily proved by means of simplification, e.g.,

```
theorem
 A0_new_thm :
   "(appl (Proc DChold1) (appl (Proc DCgteq) ((Proc DClength)
      pipe (% dv1 . ((appl (Proc DClift3) (Proc 0))
```

```
            pipe (% dv2 . v2p ((dv1,dv2)))))))
    = true"
 apply (simp add : DChold1_def DCgteq_def DClength_def DClift3_def
   appl_def v2p_def RSL_geq_real_def RSLgeq_real_def infix_appl_def
   RSL_minus_real_def RSLminus_real_def true_def)
done
```

The RSL proof of A0 given in [65] consists of 17 applications of proof rules. So Isabelle/HOL gave increased proof support for A0, but it was necessary to modify the encoding to deal with subtypes.

7.4.4 Discussion

In the RSL specifications of DC/RJ$^-$, subtypes are used. However, subtypes are not part of mRSL, and the subtype information was lost when the RSL specifications were adapted to mRSL. We came up with pragmatic solutions to the problems inflicted by the disregard of subtyping, but the problems still question the choice of Isabelle/HOL as target language for the translation.

PVS ([48]) features strong subtyping, and it is therefore natural to ask if this would have been a better choice. The RSL tools [19] available from UNU/IIST include a translator [17] from RSL to PVS; it translates RSL subtypes to PVS subtypes.

With respect to subtyping, it would of course have been easier to translate to PVS instead of Isabelle/HOL since PVS features subtyping. However, it was our intention to translate RSL specifications to higher-order logic in general (using the concrete Isabelle/HOL syntax) rather than translating to another specification language which may be seen as a competitor to RSL. Translating to higher-order logic in general is motivated by the possibilities of using other theorem provers than Isabelle and the desire to base the translation on a mapping between institutions.

If subtyping should have been included in mRSL, it should also have been part of the mRSL institution; and if subtyping in the target language was used in the translation, subtyping should have been part of the target institution.

It should be noted that although Isabelle/HOL was chosen as a “clean” implementation of higher-order logic, it moves towards being a specification language.

In contrast to the preceding examples in Section 7.2 and Section 7.3, the use of Isabelle/HOL did not increase the proof support for all the proofs considered in this example. The proof support was increased for the proof of A0, but for the other theorems, the number of lines in the Isabelle/HOL proof scripts either exceeded or were approximately the same as the number of manual applications of proof rules needed in proofs developed using the eden editor.

It should be emphasized that the intention of this example was to show how RSL specifications could be translated to Isabelle/HOL; the primary goal was not to provide proof support for DC by means of Isabelle. Proof support for DC by means of Isabelle has been considered in [33, 56]. In [56], Isabelle is used for an interval logic, i.e., the interval logic is encoded in the meta-logic of Isabelle. However, encoding the interval logic in Isabelle/HOL is also considered, and it is concluded that the latter improves the arithmetic reasoning considerably compared to the encoding in Isabelle's meta-logic.

In the Isabelle/HOL proof scripts, we concentrated on using the simplifier which was effective for the previous example applications. If the classical reasoner is set up properly, it may increase the proof support for this example application, as well as others.

CHAPTER 8

Conclusion

This thesis considered the possibility of providing increased proof support for the RAISE development method by translating RSL specifications and proof obligations to higher-order logic and make proofs with the Isabelle/HOL proof assistant. An applicative, deterministic subset of RSL was considered, and this subset was named mRSL. The translation of mRSL to higher-order logic was based on institutions which provide a framework for formally describing and relating logical systems.

In this concluding chapter, the results are summarized, related work is discussed, and directions for possible future work are given.

8.1 Results

Following the approach of using institutions as a formal foundation, new proof support for RAISE was provided by means of a translation from a subset of RSL to Isabelle/HOL. As a side-effect, new concepts and definitions of general interest emerged.

The results summarized in this section regard the concepts of the formal foundation itself, the definition of an institution and a translation, and experiences with example applications.

8.1.1 Looser Semantics

Considering institutions, specifications are traditionally given loose or initial semantics. The semantics of RSL specifications, however, has traditionally been described using a different terminology and does not correspond to the traditional ways of giving institution-independent semantics. The new concept of *looser semantics* was introduced in Definition 2.17 as an institution-independent semantics of specifications that describes the semantics of RSL specifications.

The following result was stated and proved in Theorem 2.21: A sentence is a semantic consequence of a specification with respect to loose semantics if and only if it is a semantic consequence of a specification with respect to looser semantics. Thus, loose and looser semantics are equivalent when it comes to semantic consequence.

8.1.2 Light Institution Comorphisms

Light institution comorphisms were defined as a variation of the concepts of institution comorphisms [24] and institution representations [45]. Essentially, light institution comorphisms are simple institution comorphisms in which signature morphisms are disregarded. Consequently, the mapping of signatures is a function (not a functor), the mapping α of sentences is a family of functions (not necesarrily a natural transformation), and the mapping β of models is a family of partial functions (not a natural transformation of functors).

An alternative to defining a light institution comorphism would be to remove signature morphisms from the institution of mRSL, i.e., to define $Sign_{mRSL}$ as a discrete category of mRSL signatures. However, this would make the mRSL institution less usable in other contexts since signature morphisms capture important notions regarding structured specifications.

The reusing theorem was stated and proved again for light institution comorphisms in Theorem 2.30.

8.1.3 An Institution for mRSL

A main result is the definition of an institution of a subset of RSL. The definition of the institution was based on the denotational semantics of RSL, and since most proofs in the RAISE method are done for applicative specifications, we concentrated on a deterministic, applicative subset of RSL. The original denotational semantics was simplified to accomodate the features of discourse, and an institution of the subset referred to as mRSL was defined in Definition 4.45.

The institution of mRSL together with the concept of looser semantics provide an accessible introduction to the lore of RSL semantics. Moreover, it is interesting since it combines language features from algebraic specification and model-oriented specification.

8.1.4 A Light Institution Comorphism from mRSL to HOL

Another main result is the definition of a light institution comorphism from the institution of mRSL to the institution of HOL.

The light institution comorphism provides a translation of mRSL signatures and sentences such that mRSL specifications and proof obligations can be translated to

Isabelle/HOL specifications and proof obligations. Furthermore, the light institution comorphism meets the requirements necessary for making it possible to reuse Isabelle/HOL.

8.1.5 Applications

The translation was applied to examples in order to test its usefulness and gain experience about which enhancements may be needed.

The translation to Isabelle/HOL provided increased proof support for most of the example applications. However, for some proofs, the number of lines in the Isabelle proof scripts exceeded the number of manual applications of proof rules needed when using the original RAISE proof tool.

The price that we have to pay for reusing Isabelle/HOL is that the user not only needs to know RSL but also Isabelle/HOL and that it may be difficult to recognize the original specifications and thereby get feedback from a failed proof attempt. However, this is inherent to the translational approach.

The remaining part of this section concerns the applications from Chapter 7.

General Considerations

It turned out to be helpful to adopt a certain style in function definitions. In other words, to get the most out of the Isabelle simplifier, it was necessary to write RSL specifications in a certain way.

A series of RSL proof rules were translated to Isabelle/HOL and proved as theorems using Isabelle. This not only provided ready-to-use theorems but also proved the RSL proof rules sound with respect to the denotational semantics since the translation essentially encodes the denotational semantics of mRSL in Isabelle/HOL.

Combinational Logic Circuits

The Isabelle/HOL simplifier was very effective for the example that considered simple logic circuits built from basic logic gates.

Generalized Railway Crossing

The generalized railway crossing featured an algebraic specification and a refinement step involving a model-oriented specification. It gave an impression of how proof support via Isabelle/HOL could be used in RAISE developments. Again, the Isabelle/HOL simplifier provided a high degree of automation.

DC/RJ$^-$

The third example considered a ready-made encoding of the real-time logic DC in RSL. The original RSL specifications were adapted to mRSL specifications which were translated to Isabelle/HOL specifications. When adapting the RSL specifications, information about subtypes was lost, and some of the proofs became difficult as a result. Furthermore, the encoding of some DC operators was axiomatical, and it was revealed that the use of Isabelle/HOL is much more beneficial when the definitions of operators have a more operational flavour.

8.2 Related Work

A translation from a subset of RSL to the specification language of PVS is presented in [17]. The considered subset is applicative and deterministic, and RSL constructs are translated to similar PVS constructs (the translation does not encode the RSL semantics in PVS). In [17], it is informally argued that the translation from RSL to PVS is correct with respect to the semantics, and the translation is implemented in the UNU/IIST tool suite [19].

The approach of using institutions as a formal foundation for translations from a specification language to Isabelle/HOL has also been considered in [45, 12, 11] where the specification language CASL [4] is translated to Isabelle/HOL. In contrast to RSL, the semantics of CASL is given in terms of an underlying institution from the beginning.

8.3 Future Work

The translation was done more or less manually for the applications in Chapter 7. If the translation is to be used for RAISE case-studies, however, a future task is to implement a parser, type checker, and translator for mRSL specifications. Furthermore, it may be desirable to include more features in the subset.

This thesis has considered the approach of translating RSL specifications and proof obligations to Isabelle/HOL and reusing Isabelle in order to provide increased proof support for RAISE. A different approach is to stay in RSL and implement new proof tools. This might include a proof checker with a scripting language making it possible to express strategies and rerun proofs, and a simplifier that allows the user to modify the set of rules used.

APPENDIX A

Syntax Summary of mRSL

This appendix summarizes the syntax of mRSL. The presentation of the syntax is kept in the style of [54].

Comments are not included in the grammar, but can be put in specifications between /* and */.

Specifications

specification ::= scheme_decl

Declarations

Scheme Declarations

scheme_decl ::= **scheme** id = class_expr

class_expr ::= **class** opt-decl-string **end**

decl ::=
  type_decl |
  value_decl |
  axiom_decl

Type Declarations

type_decl ::= **type** type_def-list

type_def ::= id | id = type_expr

Value Declarations

value_decl ::= **value** value_def-list

value_def ::= typing | explicit_function_def

Explicit Function Definitions

explicit_function_def ::= typing formal_function_application ≡ value_expr

formal_function_application ::= id formal_function_parameter-string

formal_function_parameter ::=
    () |
    (binding) |
    (binding , binding)

Axiom Declarations

axiom_decl ::= **axiom** axiom_def-list

axiom_def ::= opt-axiom_naming equivalence_expr

axiom_naming ::= [id]

Type Expressions

type_expr ::=
    type_literal |
    type_name |
    product_type_expr |
    function_type_expr |
    bracketed_type_expr

Type Literals

type_literal ::= **Unit** | **Bool** | **Int** | **Real**

Product Type Expressions

product_type_expr ::= type_expr × type_expr

Function Type Expressions

function_type_expr ::= type_expr $\xrightarrow{\sim}$ type_expr

Bracketed Type Expressions

bracketed_type_expr ::= (type_expr)

Value Expressions

value_expr ::=
  value_literal |
  value_name |
  basic_expr |
  product_expr |
  function_expr |
  application_expr |
  quantified_expr |
  equivalence_expr |
  bracketed_expr |
  infix_expr |
  prefix_expr |
  structured_expr

Value Literals

value_literal ::=
  unit_literal |
  bool_literal |
  int_literal |
  real_literal

unit_literal ::= ()

bool_literal ::= **true** | **false**

Basic Expressions

basic_expr ::= **chaos** | **skip**

Product Expressions

product_expr ::= (value_expr , value_expr)

Function Expressions

function_expr ::= λ typing • value_expr

Application Expressions

application_expr ::= value_expr actual_function_parameter

actual_function_parameter ::=
  () |
  (value_expr) |
  (value_expr , value_expr)

Quantified Expressions

quantified_expr ::= quantifier typing restriction

quantifier ::= ∀ | ∃

restriction ::= • value_expr

Equivalence Expressions

equivalence_expr ::= value_expr ≡ value_expr

Bracketed Expressions

bracketed_expr ::= (value_expr)

Infix Expressions

infix_expr ::=
  axiom_infix_expr |
  value_infix_expr

Axiom Infix Expressions

axiom_infix_expr ::= value_expr infix_connective value_expr

Value Infix Expressions

value_infix_expr ::= value_expr infix_op value_expr

Prefix expressions

prefix_expr ::=
  axiom_prefix_expr |
  value_prefix_expr

Axiom Prefix Expressions

axiom_prefix_expr ::= prefix_connective value_expr

Value Prefix Expressions

value_prefix_expr ::= prefix_op value_expr

Structured expressions

structured_expr ::= if_expr

if_expr ::= **if** value_expr **then** value_expr **else** value_expr **end**

Bindings

binding ::= id

Typings

typing ::= id : type_expr

Names

Identifiers and Operators

Infix Operators

infix_op ::= $= \mid \neq \mid < \mid > \mid \geq \mid \leq \mid + \mid - \mid \backslash \mid * \mid /$

Prefix Operators

prefix_op ::= **abs**

Connectives

connective ::=
  infix_connective |
  prefix_connective

Infix Connectives

infix_connective ::= $\Rightarrow \mid \wedge \mid \vee$

Prefix Connectives

prefix_connective ::= $\sim$

Miscellaneous

Names

type_name ::= id

value_name ::= id

Lists

type_def-list ::= type_def | type_def , type_def-list
value_def-list ::= value_def | value_def , value_def-list
axiom_def-list ::= axiom_def | axiom_def , axiom_def-list

Strings

decl-string ::= decl | decl decl-string

formal_function_parameter-string ::=
formal_function_parameter |
formal_function_parameter formal_function_parameter-string

Options

opt-decl-string ::= ϵ | decl-string

opt-axiom_naming ::= ϵ | axiom_naming

opt-value_expr ::= ϵ | value_expr

APPENDIX B

Isabelle/HOL Encoding of mRSL

This appendix contains the Isabelle/HOL theory `RSL.thy` which encodes mRSL. The theory is listed in section B.1 without the encoding of RSL proof rules. A listing of the RSL proof rules that has been proved is given in section B.2. The proof scripts have been left out to shorten the presentation.

B.1 Isabelle/HOL Theory

```
theory RSL = Real :

(* Semantic operators *)

  datatype 'a Process = Proc 'a | Chaos

  consts
    pipe :: "'a Process => ('a => 'b Process) => 'b Process"
                                                      (infixl "pipe" 65)

  primrec
    pipe_proc : "Proc x pipe f = f x"
    pipe_chaos : "Chaos pipe f  = Chaos"

  constdefs
    v2p :: "'a => 'a Process"
    "v2p x == Proc x"
```

```
(* Support for translating value expressions *)

(* ---- value literals --- *)

  constdefs
    true :: "bool Process"
    "true == Proc True"

    false :: "bool Process"
    "false == Proc False"

(* ---- basic expressions ---- *)
  constdefs
    chaos :: "'a Process"
    "chaos == Chaos"

    skip :: "unit Process"
    "skip == Proc ()"

(* ---- product expressions ---- *)

  constdefs
    prod :: "'a Process => 'b Process => ('a * 'b) Process"
    "prod ve1 ve2 == ve1 pipe (%dv1. ve2 pipe (%dv2. v2p (dv1,dv2)))"

(* ---- application expressions ---- *)

  constdefs
    appl :: "('a => 'b Process) Process => ('a Process) => ('b Process)"
    "appl f x == f pipe (% dv1 . x pipe (% dv2 . dv1 dv2))"


(* ---- infix expressions ---- *)

  consts
    infix_conn_appl :: "(bool Process) => (bool => (bool Process) =>
                        (bool Process)) => (bool Process) => (bool Process)"

    RSLand :: "bool => (bool Process) => (bool Process)"
    RSLor :: "bool => (bool Process) => (bool Process)"
    RSLimp :: "bool => (bool Process) => (bool Process)"

  defs
    infix_conn_appl_def :
      "infix_conn_appl ve1 iconn ve2 == ve1 pipe (% dv1 . iconn dv1 ve2)"

    RSLand_def : "RSLand b p == if b then p else (v2p False)"
```

```
  RSLor_def : "RSLor b p == if b then (v2p True) else p"
  RSLimp_def : "RSLimp b p == if b then p else (v2p True)"

(* Shorthands *)

 consts
  RSL_and :: "(bool Process) => (bool Process) => (bool Process)"
                                                       (infixl "AND" 60)
  RSL_or :: "(bool Process) => (bool Process) => (bool Process)"
                                                       (infixl "OR" 60)
  RSL_imp :: "(bool Process) => (bool Process) => (bool Process)"
                                                       (infixl "IMP" 60)

 defs
  RSL_and_def : "bp1 AND bp2 == infix_conn_appl bp1 RSLand bp2"
  RSL_or_def : "bp1 OR bp2 == infix_conn_appl bp1 RSLor bp2"
  RSL_imp_def : "bp1 IMP bp2 == infix_conn_appl bp1 RSLimp bp2"

 consts
  infix_appl :: "('a Process) => ('a => 'b => 'c Process) =>
                ('b Process) => ('c Process)"

  RSLeq :: "'a => 'a => bool Process"
  RSLneq :: "'a => 'a => bool Process"
  RSLl :: "int => int => bool Process"
  RSLg :: "int => int => bool Process"
  RSLgeq :: "int => int => bool Process"
  RSLleq :: "int => int => bool Process"
  RSLplus :: "int => int => int Process"
  RSLminus :: "int => int => int Process"
  RSLmod :: "int => int => int Process"
  RSLmult :: "int => int => int Process"
  RSLdiv :: "int => int => int Process"

  RSLl_real :: "real => real => bool Process"
  RSLg_real :: "real => real => bool Process"
  RSLgeq_real :: "real => real => bool Process"
  RSLleq_real :: "real => real => bool Process"
  RSLplus_real :: "real => real => real Process"
  RSLminus_real :: "real => real => real Process"
  RSLmult_real :: "real => real => real Process"

 defs
  infix_appl_def :
    "infix_appl x iop y == x pipe (% dv1 . y pipe (% dv2 . iop dv1 dv2))"

  RSLeq_def : "RSLeq x y == v2p (x = y)"
  RSLneq_def : "RSLneq x y == v2p (x ~= y)"
```

```
   RSLl_def : "RSLl x y == v2p (x < y)"
   RSLg_def : "RSLg x y == v2p (y < x)"
   RSLgeq_def : "RSLgeq x y == v2p (y <= x)"
   RSLleq_def : "RSLleq x y == v2p (x <= y)"
   RSLplus_def : "RSLplus x y == v2p (x + y)"
   RSLminus_def : "RSLminus x y == v2p (x - y)"
   RSLmod_def : "RSLmod x y == v2p (x mod y)"
   RSLmult_def : "RSLmult x y == v2p (x * y)"
   RSLdiv_def : "RSLdiv x y == v2p (x div y)"

   RSLl_real_def : "RSLl_real x y == v2p (x < y)"
   RSLg_real_def : "RSLg_real x y == v2p (y < x)"
   RSLgeq_real_def : "RSLgeq_real x y == v2p (y <= x)"
   RSLleq_real_def : "RSLleq_real x y == v2p (x <= y)"
   RSLplus_real_def : "RSLplus_real x y == v2p (x + y)"
   RSLminus_real_def : "RSLminus_real x y == v2p (x - y)"
   RSLmult_real_def : "RSLmult_real x y == v2p (x * y)"

 (* Shorthands are introduced to provide readable translations *)

consts
   RSL_eq :: "('a Process) => ('a Process) => (bool Process)"
                                                        (infixl "=rsl" 60)
   RSL_neq :: "('a Process) => ('a Process) => (bool Process)"
                                                        (infixl "~=rsl" 60)
   RSL_l :: "(int Process) => (int Process) => (bool Process)"
                                                        (infixl "<rsl" 60)
   RSL_g :: "(int Process) => (int Process) => (bool Process)"
                                                        (infixl ">rsl" 60)
   RSL_geq :: "(int Process) => (int Process) => (bool Process)"
                                                        (infixl ">=rsl" 60)
   RSL_leq :: "(int Process) => (int Process) => (bool Process)"
                                                        (infixl "<=rsl" 60)
   RSL_plus :: "(int Process) => (int Process) => (int Process)"
                                                        (infixl "+rsl" 60)
   RSL_minus :: "(int Process) => (int Process) => (int Process)"
                                                        (infixl "-rsl" 60)
   RSL_mod :: "(int Process) => (int Process) => (int Process)"
                                                        (infixl "modrsl" 60)
   RSL_mult :: "(int Process) => (int Process) => (int Process)"
                                                        (infixl "*rsl" 60)
   RSL_div :: "(int Process) => (int Process) => (int Process)"
                                                        (infixl "divrsl" 60)

   RSL_l_real :: "(real Process) => (real Process) => (bool Process)"
                                                        (infixl "<rrsl" 60)
   RSL_g_real :: "(real Process) => (real Process) => (bool Process)"
                                                        (infixl ">rrsl" 60)
```

```
    RSL_geq_real :: "(real Process) => (real Process) => (bool Process)"
                                                       (infixl ">=rrsl" 60)
    RSL_leq_real :: "(real Process) => (real Process) => (bool Process)"
                                                       (infixl "<=rrsl" 60)
    RSL_plus_real :: "(real Process) => (real Process) => (real Process)"
                                                       (infixl "+rrsl" 60)
    RSL_minus_real :: "(real Process) => (real Process) => (real Process)"
                                                       (infixl "-rrsl" 60)
    RSL_mult_real :: "(real Process) => (real Process) => (real Process)"
                                                       (infixl "*rrsl" 60)

 defs
    RSL_eq_def : "p1 =rsl p2 == infix_appl p1 RSLeq p2"
    RSL_neq_def : "p1 ~=rsl p2 == infix_appl p1 RSLneq p2"
    RSL_l_def : "p1 <rsl p2 == infix_appl p1 RSLl p2"
    RSL_g_def : "p1 >rsl p2 == infix_appl p1 RSLg p2"
    RSL_geq_def : "p1 >=rsl p2 == infix_appl p1 RSLgeq p2"
    RSL_leq_def : "p1 <=rsl p2 == infix_appl p1 RSLleq p2"
    RSL_plus_def : "p1 +rsl p2 == infix_appl p1 RSLplus p2"
    RSL_minus_def : "p1 -rsl p2 == infix_appl p1 RSLminus p2"
    RSL_mod_def : "p1 modrsl p2 == infix_appl p1 RSLmod p2"
    RSL_mult_def : "p1 *rsl p2 == infix_appl p1 RSLmult p2"
    RSL_div_def : "p1 divrsl p2 == infix_appl p1 RSLdiv p2"

    RSL_l_real_def : "p1 <rrsl p2 == infix_appl p1 RSLl_real p2"
    RSL_g_real_def : "p1 >rrsl p2 == infix_appl p1 RSLg_real p2"
    RSL_geq_real_def : "p1 >=rrsl p2 == infix_appl p1 RSLgeq_real p2"
    RSL_leq_real_def : "p1 <=rrsl p2 == infix_appl p1 RSLleq_real p2"
    RSL_plus_real_def : "p1 +rrsl p2 == infix_appl p1 RSLplus_real p2"
    RSL_minus_real_def : "p1 -rrsl p2 == infix_appl p1 RSLminus_real p2"
    RSL_mult_real_def : "p1 *rrsl p2 == infix_appl p1 RSLmult_real p2"

(* ---- prefix expressions ---- *)

  consts
    prefix_conn_appl ::
      "(bool => (bool Process)) => (bool Process) => (bool Process)"

    RSLnot :: "bool => (bool Process)"

    RSL_not :: "(bool Process) => (bool Process)"  ("NOT _")

  defs
    prefix_conn_appl_def :
      "prefix_conn_appl pconn ve == ve pipe (% dv1 . pconn dv1)"

    RSLnot_def : "RSLnot b == if b then (v2p False) else (v2p True)"
```

```
    RSL_not_def : "NOT bp == prefix_conn_appl RSLnot bp"

  consts
    prefix_appl :: "('a => 'b Process) => ('a Process) => ('b Process)"

    RSLabs :: "'a::minus => ('a Process)"

    RSL_abs :: "('a::minus Process) => ('a Process)"   ("absrsl _")

  defs
    prefix_appl_def : "prefix_appl pop x == x pipe (% dv1 . pop dv1)"

    RSLabs_def : "RSLabs x == v2p (abs x)"

    RSL_abs_def : "absrsl p == prefix_appl RSLabs p"

(* ---- structured expressions ---- *)

  constdefs
    ifexpr :: "bool Process => 'a Process => 'a Process => 'a Process"
                                                 ("(if _ then _ else _ end)")
    "if veb then ve1 else ve2 end ==
      veb pipe (%dv1. if dv1 then (ve1 pipe v2p) else (ve2 pipe v2p))"

end
```

B.2 RSL Proof Rules

B.2.1 Infix Operators

```
theorem equality_annihilation : "(Proc e) =rsl (Proc e) == true"

theorem Equality_annihilation : "infix_appl (Proc e) RSLeq (Proc e) == true"

theorem unit_equality : "(Proc eu) =rsl (Proc ()) == true"

theorem Unit_equality : "infix_appl (Proc eu) RSLeq (Proc ()) == true"

theorem true_or_false :
          "((Proc eb) =rsl true) OR ((Proc eb) =rsl false) == true"

theorem True_or_false : "infix_conn_appl (infix_appl (Proc eb) RSLeq true)
                          RSLor (infix_appl (Proc eb) RSLeq false)
                        == true"

theorem inequality_annihilation : "(Proc e) ~=rsl (Proc e) == false"
```

```
theorem Inequality_annihilation :
          "infix_appl (Proc e) RSLneq (Proc e) == false"

theorem inequality_expansion :
          "e ~=rsl e' == prefix_conn_appl RSLnot (e =rsl e')"

theorem Inequality_expansion :
  "infix_appl e RSLneq e' == prefix_conn_appl RSLnot (infix_appl e RSLeq e')"

theorem not_inequality : "NOT (e ~=rsl e') == e =rsl e'"

theorem Not_inequality :
  "prefix_conn_appl RSLnot (infix_appl e RSLneq e') == infix_appl e RSLeq e'"

theorem equality_commutativity : "e =rsl e' = e' =rsl e"

theorem Equality_commutativity :
          "infix_appl e RSLeq e' = infix_appl e' RSLeq e"

theorem inequality_commutativity : "e ~=rsl e' = e' ~=rsl e"

theorem Inequality_commutativity :
          "infix_appl e RSLneq e' = infix_appl e' RSLneq e"
```

B.2.2 If-Expressions

```
theorem if_expansion5 : "if (Proc b) then eb' else eb'' end
                         = (((Proc b) IMP eb') AND ((NOT (Proc b))IMP eb''))"

theorem If_expansion5 :
  "if (Proc b) then eb' else eb'' end
   = (infix_conn_appl
       (infix_conn_appl (Proc b) RSLimp eb')
       RSLand
       (infix_conn_appl (prefix_conn_appl RSLnot (Proc b)) RSLimp eb''))"

theorem if_expansion5' :
  "(eb = Proc b) ==>
     (if eb then eb' else eb'' end = ((eb IMP eb') AND ((NOT eb) IMP eb'')))"

theorem If_expansion5' :
  "(eb = Proc b) ==>
     (if eb then eb' else eb'' end
     = (infix_conn_appl
         (infix_conn_appl eb RSLimp eb')
         RSLand
         (infix_conn_appl (prefix_conn_appl RSLnot eb) RSLimp eb'')))"
```

```
theorem if_reduction1 :
  "(eb = Proc b) ==>
     (if eb then eb else eb' end = if eb then true else eb' end)"

theorem if_reduction2 :
  "(eb = Proc b) ==>
     (if eb then eb' else eb end = if eb then eb' else false end)"

theorem if_annihilation1 :
  "(eb = Proc b) ==> (if eb then eb else true end = true)"

theorem if_annihilation2 :
  "(eb = Proc b) ==> (if eb then false else eb end = false)"

theorem if_true : "if true then e else e' end = e"

theorem if_false : "if false then e else e' end = e'"

theorem if_absorption : "if eb then true else false end = eb"

theorem if_idempotence1 :
  "(eb = Proc b) ==>
     (if eb then
        if eb then e else e' end
      else
        if eb then e'' else e''' end
      end
      = if eb then e else e''' end)"

theorem if_idempotence3 : "(eb = Proc b) ==> (if eb then e else e end = e)"

theorem if_idempotence4 :
  "(eb = Proc b) ==>
     (if eb then if eb then e else e' end else e'' end
      = if eb then e else e'' end)"

theorem if_commutativity :
  "(eb = Proc b) ==>
     (if eb then
        if eb' then e else e' end
      else
        if eb' then e'' else e''' end
      end
      =
      if eb' then
        if eb then e else e'' end
      else
        if eb then e' else e''' end
```

```
      end)"

theorem if_associativity :
  "if if eb then eb' else eb'' end then e else e' end
   =
   if eb then if eb' then e else e' end else if eb'' then e else e' end end"

theorem if_not1 : "if (NOT eb) then e else e' end = if eb then e' else e end"

theorem If_not1 :
  "if (prefix_conn_appl RSLnot eb) then e else e' end
   = if eb then e' else e end"

theorem if_not2 :
  "if eb then (NOT eb') else (NOT eb'') end
   = NOT (if eb then eb' else eb'' end)"

theorem If_not2 :
  "if eb then
     (prefix_conn_appl RSLnot eb')
   else
     (prefix_conn_appl RSLnot eb'')
   end
   = prefix_conn_appl RSLnot (if eb then eb' else eb'' end)"

theorem if_and :
  "(eb = Proc b) ==>
     (((if eb then eb else eb'' end) AND (if eb then eb' else eb''' end))
     = (if eb then (eb AND eb') else (eb'' AND eb''') end))"

theorem If_and :
  "(eb = Proc b) ==>
     ((infix_conn_appl
        (if eb then eb else eb'' end)
        RSLand
        (if eb then eb' else eb''' end))
     = (if eb then
          (infix_conn_appl eb RSLand eb')
        else
          (infix_conn_appl eb'' RSLand eb''')
        end))"

theorem if_or :
  "(eb = Proc b) ==>
     (((if eb then eb else eb'' end) OR (if eb then eb' else eb''' end))
     = (if eb then (eb OR eb') else (eb'' OR eb''') end))"

theorem If_or :
```

```
  "(eb = Proc b) ==>
     ((infix_conn_appl
        (if eb then eb else eb'' end)
        RSLor
        (if eb then eb' else eb''' end))
     = (if eb then
          (infix_conn_appl eb RSLor eb')
        else
          (infix_conn_appl eb'' RSLor eb''')
        end))"

theorem if_implies :
  "(eb = Proc b) ==>
     (((if eb then eb else eb'' end) IMP (if eb then eb' else eb''' end))
     = (if eb then (eb IMP eb') else (eb'' IMP eb''') end))"

theorem If_implies :
  "(eb = Proc b) ==>
     ((infix_conn_appl
        (if eb then eb else eb'' end)
        RSLimp
        (if eb then eb' else eb''' end))
     = (if eb then
          (infix_conn_appl eb RSLimp eb')
        else
          (infix_conn_appl eb'' RSLimp eb''')
        end))"

theorem imply_modus_ponens_inf :
  "[| eb = Proc True; (eb IMP eb') = Proc True |] ==> (eb' = Proc True)"

theorem Imply_modus_ponens_inf :
  "[| eb = Proc True; (infix_conn_appl eb RSLimp eb') = Proc True |] ==>
     (eb' = Proc True)"

theorem and_left_inf : "((eb AND eb') = Proc True) ==> eb = Proc True"

theorem And_left_inf :
  "((infix_conn_appl eb RSLand eb') = Proc True) ==> eb = Proc True"

theorem and_right_inf : "((eb AND eb') = Proc True) ==> (eb' = Proc True)"

theorem And_right_inf :
  "((infix_conn_appl eb RSLand eb') = Proc True) ==> (eb' = Proc True)"

theorem and_split_inf :
  "[| eb = Proc True; eb' = Proc True |] ==> ((eb AND eb') = Proc True)"
```

```
theorem And_split_inf :
  "[| eb = Proc True; eb' = Proc True |] ==>
     ((infix_conn_appl eb RSLand eb') = Proc True)"

theorem or_left_inf : "(eb = Proc True) ==> ((eb OR eb') = Proc True)"

theorem Or_left_inf :
  "(eb = Proc True) ==> ((infix_conn_appl eb RSLor eb') = Proc True)"

theorem or_right_inf :
  "[| eb = Proc b; eb = Proc True |] ==> ((eb OR eb') = Proc True)"

theorem Or_right_inf :
  "[| eb = Proc b; eb = Proc True |] ==>
     ((infix_conn_appl eb RSLor eb') = Proc True)"
```

B.2.3 Connectives

```
theorem implies_expansion1 : "(eb IMP eb') == (if eb then eb' else true end)"

theorem not_not : "(NOT (NOT eb)) == eb"

theorem
 or_commutativity : "((Proc eb) OR (Proc eb')) == ((Proc eb') OR (Proc eb))"

theorem
 Or_commutativity :
   "[| eb = Proc b; eb' = Proc b' |] ==> ((eb OR eb') == (eb' OR eb))"
```

B.2.4 Miscellaneous

```
theorem Proc_v2p_thm1 : "((Proc x) == (v2p y)) ==> (x == y)"

theorem Proc_v2p_thm2 : "((Proc x) = (v2p y)) ==> (x = y)"
```

Bibliography

[1] Peter B. Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof.* Academic Press, 1986.

[2] David Aspinall. Proof General: A Generic Tool for Proof Development. In Susanne Graf and Michael Schwartzbach, editors, *Tools and Algorithms for the Construction and Analysis of Systems TACAS 2000*, volume 1785 of *Lecture Notes in Computer Science*, pages 38–42. Springer, 2000.

[3] E. Astesiano and M. Cerioli. Relationships between Logical Frameworks. In M. Bidoit and C. Choppy, editors, *Proc. 8th WADT workshop*, volume 655 of *Lecture Notes in Computer Science*, pages 126–143. Springer Verlag, 1993.

[4] Egidio Astesiano, Michel Bidoit, Bernd Krieg-Brückner, Peter D. Mosses, Donald Sannella, and Andrzej Tarlecki. CASL: The Common Algebraic Specification Language. *Theoretical Computer Science*, 286(2):153–196, 2002.

[5] Stefan Berghofer. Definitorische Konstruktion induktiver Datentypen in Isabelle/HOL. Master's thesis, Technische Universität München, September 1998. In German.

[6] Stefan Berghofer and Markus Wenzel. Inductive Datatypes in HOL – Lessons Learned in Formal-Logic Engineering. In Y. Bertot et al., editor, *TPHOLs'99*, volume 1690 of *Lecture Notes in Computer Science*, pages 19–36. Springer-Verlag, 1999.

[7] Jan A. Bergstra and Alban Ponse. Kleenes's three-valued logic and process algebra. *Information Processing Letters*, 67:95–103, 1998.

[8] M. Bidoit, M.V. Cengarle, and R. Hennicker. Proof Systems for Structured Specifications and Their Refinements. In E. Astesiano, H.-J. Kreowski, and B. Krieg-Brückner, editors, *Algebraic Foundations of Systems Specification*, chapter 11, pages 385–433. Springer, 1999.

[9] Dines Bjørner, Hans Langmaack, and C.A.R. Hoare, editors. *Provably Correct Systems*, chapter 8, pages 93–107. ProCoS I Partners, 1993.

[10] Dominique Bolignano and Mourad Debabi. On the Foundations of the RAISE Specification Language Semantics. Technical Report LACOS/BULL/DB/27/V1, Bull, June 1992. ESPRIT 5383, LaCoS.

[11] T. Borzyszkowski. Moving specification structures between logical systems. In J. L. Fiadeiro, editor, *Recent Trends in Algebraic Development Techniques, 13th International Workshop, WADT'98, Lisbon, Portugal, April 1998, Selected Papers*, volume 1589 of *Lecture Notes in Computer Science*, pages 16–30. Springer, 1999.

[12] Tomasz Borzyszkowski. Higher-Order Logic and Theorem Proving for Structured Specifications. In Christine Choppy, Didier Bert, and Peter Mosses, editors, *Recent Trends in Algebraic Development Techniques, Selected Papers, 14th International Workshop WADT'99, Bonas, France September 1999*, volume 1827 of *Lecture Notes in Computer Science*, pages 401–418. Springer, 1999.

[13] Zhou Chaochen. Duration Calculii: An Overview. In D. Bjørner, M. Broy, and I.V. Pottosin, editors, *Proceedings of Formal Methods in Programming and Their Applications*, volume 735 of *Lecture Notes in Computer Science*, pages 256–266. Springer-Verlag, 1993.

[14] Zhou Chaochen and Michael R. Hansen. *Duration Calculus: A formal approach to real-time systems.* Springer-Verlag, 2004. To be published.

[15] Alonzo Church. A Formulation of the Simple Theory of Types. *Journal of Symbolic Logic*, 5(2):56–68, June 1940.

[16] CoFI. CASL *Reference Manual* (P. D. Mosses, ed.). Lecture Notes in Computer Science, IFIP Series. Springer-Verlag, 2003. To appear; draft available from `http://www.cofi.info/CASL_RefManual_DRAFT.pdf`.

[17] Aristides Dasso and Chris George. Transforming RSL into PVS. Technical Report 256, UNU/IIST, May 2002.

[18] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1*, volume 6 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, 1985.

[19] Chris George. RAISE Tool User Guide. Technical Report 227, UNU/IIST, January 2001.

[20] Chris George. Introduction to RAISE. Technical Report 249, UNU/IIST, March 2002.

[21] Chris George and Søren Prehn. *The RAISE Justification Handbook*, April 1993.

[22] Chris George and Xia Yong. An Operational Semantics for Timed RAISE. Technical Report 149, UNU/IIST, November 1998.

[23] J.A. Goguen and R.M. Burstall. Introducing Institutions. In Edmund Clarke and Dexter Kozen, editors, *Logic of Programs, Proceedings 1983*, volume 164 of *Lecture Notes in Computer Science*, pages 221–256. Springer-Verlag, 1984.

[24] Joseph Goguen and Grigore Roşu. Institution Morphisms. *Formal Aspects of Computing*, 13:274–307, 2002.

[25] Joseph A. Goguen and Rod M. Burstall. Institutions: Abstract Model Theory for Specification and Programming. *Journal of the ACM*, 39(1):95–146, January 1992.

[26] Michael J. Gordon, Robert Milner, and Christopher P. Wadsworth. *Edingburgh LCF: A Mechanised Logic of Computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979.

[27] M.J.C. Gordon and T.F. Melham, editors. *Introduction to HOL: A theorem proving environment for higher order logic.* Cambridge University Press, 1993.

[28] The CRI RAISE Tools Group. The RAISE Toolset. In *VDM '91: Formal Software Development Methods*, pages 689–690. Springer-Verlag, October 1991.

[29] M.R. Hansen and Zhou Chaochen. Duration Calculus: Logical Foundations. *Formal Aspects of Computing*, 9:283–330, 1997.

[30] A. Haxthausen and J. Peleska. Formal Development and Verification of a Distributed Railway Control System. In *Proceedings of Formal Methods World Congress FM'99*, volume 1709 of *Lecture Notes in Computer Science*, pages 1546 – 1563. Springer-Verlag, 1999.

[31] A. Haxthausen and X. Yong. A RAISE Specification Framework and Justification Assistant for the Duration Calculus. In *Proceedings of ESSLLI-98 Workshop on Duration Calculus*, pages 51–58, 1998.

[32] Anne Elisabeth Haxthausen. *Structuring Mechanisms in Formal Specification Languages.* PhD thesis, Technical University of Denmark, July 1988. ID-TR: 1988-42.

[33] S. T. Heilmann. *Proof Support for Duration Calculus.* PhD thesis, Dept. of Information Technology, Technical University of Denmark, January 1999.

[34] Connie Heitmeyer and Nancy Lynch. The Generalized Railroad Crossing: A Case Study in Formal Verification of Real-Time System. In *Proceedings of the 15th IEEE Real-Time Systems Symposium, San Juan, Puerto Rico, December 1994*, pages 120–131. IEEE, IEEE Computer Society Press, December 1994.

[35] Leon Henkin. Completeness in the Theory of Types. *The Journal of Symbolic Logic*, 15(2):81–91, June 1950.

[36] Cliff B. Jones. *Systematic Software Development using VDM*. Prentice-Hall, Upper Saddle River, NJ 07458, USA, 1990.

[37] S. C. Kleene. On Notation for Ordinal Numbers. *Journal of Symbolic Logic*, 3(4):150–155, December 1938.

[38] Morten P. Lindegaard. Representing RSL Specifications in Isabelle/HOL. In David Basin and Burkhart Wolff, editors, *TPHOLs 2003 – Emerging Trends Proceedings*, pages 95–110, Rome, Italy, September 2003. Aracne Editrice S.R.L. Technical Report No. 187, Universität Freiburg.

[39] Morten P. Lindegaard and Peter Viuf. Modelling Railway Interlocking Systems. Master's thesis, Technical University of Denmark, April 2000.

[40] M.P. Lindegaard, P. Viuf, and A.E. Haxthausen. Modelling Railway Interlocking Systems. In E. Schnieder and U. Becker, editors, *Proceedings of the 9th IFAC Symposium on Control in Transportation Systems 2000, June 13–15, 2000 Braunschweig, Germany*, volume 1, pages 211–217. IFAC, 2000.

[41] Jaques Loeckx, Hans-Dieter Ehrich, and Markus Wolf. *Specification of Abstract Data Types*. WileyTeubner, 1996.

[42] J. Meseguer. General Logics. In *Logic Colloquium 87*, pages 275–329. North Holland, 1989.

[43] R. Milne. *Semantic Foundations of RSL*. CRI, March 1990.

[44] R. E. Milne. The Formal Basis for the RAISE Specification Language. In *Semantics of Specification Languages*, Workshops in Computing. Springer-Verlag, 1993.

[45] Till Mossakowski. Relating CASL with Other Specification Languages: the Institution Level. *Theoretical Computer Science*, 286:367–475, 2002.

[46] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL – A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

[47] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. Isabelle's Logics: HOL. Available at `http://www.cl.cam.ac.uk/Research/HVG/Isabelle/docs.html`, May 2003.

[48] S. Owre, J.M. Rushby, and N. Shankar. PVS: A Prototype Verification System. In D. Kapur, editor, *Automated Deduction – CADE-11, 11th International Conference on Automated Deduction, Saratoga Springs, NY, USA, June 1992, Proceedings*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752. Springer-Verlag, June 1992.

[49] Lawrence C. Paulson. The Foundation of a Generic Theorem Prover. *Journal of Automated Reasoning*, 5:363–397, 1989.

[50] Lawrence C. Paulson. Isabelle: The Next 700 Theorem Provers. In P. Odifreddi, editor, *Logic and Computer Science*, pages 361–386. Academic Press, 1990.

[51] Lawrence C. Paulson. Introduction to Isabelle. Available at `http://www.cl.cam.ac.uk/Research/HVG/Isabelle/docs.html`, May 2003.

[52] Lawrence C. Paulson. Isabelle's Logics. Available at `http://www.cl.cam.ac.uk/Research/HVG/Isabelle/docs.html`, May 2003. With contributions by Tobias Nipkow and Markus Wenzel.

[53] Lawrence C. Paulson. Isabelle's Logics: FOL and ZF. Available at `http://www.cl.cam.ac.uk/Research/HVG/Isabelle/docs.html`, May 2003. With contributions by Tobias Nipkow and Markus Wenzel.

[54] The RAISE Language Group. *The RAISE Specification Language*. The BCS Practitioners Series. Prentice Hall Int., 1992.

[55] The RAISE Method Group. *The RAISE Development Method*. The BCS Practitioners Series. Prentice Hall Int., 1995.

[56] Thomas Marthedal Rasmussen. *Interval Logic – Proof Theory and Theorem Proving*. PhD thesis, IMM, Technical University of Denmark, January 2002.

[57] A. Salibra and G. Scollo. A soft stairway to institutions. In M. Bidoit and C. Choppy, editors, *Proc. 8th ADT workshop*, volume 655 of *Lecture Notes in Computer Science*, pages 310–329. Springer Verlag, 1992.

[58] Ri Hyon Sul and He Jifeng. A Complete Verification System for Timed RSL. Technical Report 275, UNU/IIST, March 2003.

[59] A. Tarlecki. Moving between logical systems. In M. Haveraaen, O. Owe, and O.-J. Dahl, editors, *Recent Trends in Data Type Specifications. 11th Workshop on Specification of Abstract Data Types*, volume 1130 of *Lecture Notes in Computer Science*, pages 478–502. Springer Verlag, 1996.

[60] A. Tarlecki. Institutions: An Abstract Framework for Formal Specifications. In E. Astesiano, H.-J. Kreowski, and B. Krieg-Brückner, editors, *Algebraic Foundations of Systems Specification*, chapter 4, pages 105–130. Springer, 1999.

[61] Hung Dang Van, Chris George, Tomasz Janowski, and Richard Moore, editors. *Specification Case Studies in RAISE*. FACIT (Formal Approaches to Computing and Information Technology) series. Springer, 2002.

[62] Jaap van Oosten. Basic Category Theory. Technical Report BRICS LS-95-1, BRICS, January 1995.

[63] John F. Wakerly. *Digital Design – Principles and Practices.* Prentice-Hall International, Inc., 2nd edition, 1994.

[64] Markus Wenzel. *The Isabelle/Isar Reference Manual.* TU München, March 2002. Current version available at
`http://www.cl.cam.ac.uk/Research/HVG/Isabelle/docs.html`.

[65] Xia Yong. DC/RJ$^-$: A Justification Assistant for Duration Calculus. Technical Report 126, UNU/IIST, November 1997.