A Type-Theoretic Analysis of Modular Specifications

Savitri Maharaj

Doctor of Philosophy
University of Edinburgh
1995

Abstract

We study the problem of representing a modular specification language in a type-theory based theorem prover. Our goals are: to provide mechanical support for reasoning about specifications and about the specification language itself; to clarify the semantics of the specification language by formalising them fully; to augment the specification language with a programming language in a setting where they are both part of the same formal environment, allowing us to define a formal implementation relationship between the two.

Previous work on similar issues has given rise to a dichotomy between "shallow" and "deep" embedding styles when representing one language within another. We show that the expressiveness of type theory, and the high degree of reflection that it permits, allow us to develop embedding techniques which lie between the "shallow" and "deep" extremes. We consider various possible embedding strategies and then choose one of them to explore more fully.

As our object of study we choose a fragment of the Z specification language, which we encode in the type theory UTT, as implemented in the LEGO proof-checker. We use the encoding to study some of the operations on schemas provided by Z. One of our main concerns is whether it is possible to reason about Z specifications at the level of these operations. We prove some theorems about Z showing that, within certain constraints, this kind of reasoning is indeed possible. We then show how these metatheorems can be used to carry out formal reasoning about Z specifications. For this we make use of an example taken from the Z Reference Manual (ZRM).

Finally, we exploit the fact that type theory provides a programming language as well as a logic to define a notion of implementation for Z specifications. We illustrate this by encoding some example programs taken from the ZRM.

Declaration

I declare that this thesis was composed by myself, and that the work contained in it is my own except where otherwise stated. Some of this work has been published previously [Mah94].

Savitri Maharaj

Acknowledgements

First of all, I would like to thank my supervisor, Stuart Anderson, for his invaluable encouragement during this work. I would also like to thank Rod Burstall, who first introduced me to my thesis topic, and who has supported me in many ways throughout the last few years. Thanks also go to my examiners, Cliff Jones and Stephen Gilmore, whose comments on this thesis have substantially deepened my understanding of my work. My ideas have been influenced also by discussions and comments from the following people: Thorsten Altenkirch; Yves Bertot; Bob Constable; Ranan Fraer; Healf Goguen; Elsa Gunter; Claire Jones; Gilles Kahn; Zhaohui Luo; Lena Magnusson; Randy Pollack; Alex Simpson; the members of the LEGO club at Edinburgh.

I am also very grateful to the British Council and to the ORS for the material support that they have provided me, and to my family, friends, and fellow-sufferers, for support of the other kind.

Table of Contents

Chapter 1

Introduction

In the quest after formal methods, many expressive languages have been developed for use in writing program specifications. These languages often provide specification-building operations which enable specifications to be constructed in a modular fashion. For effective use of specification languages it is desirable to have automated support to facilitate reasoning about the properties of specified systems. It is even better if this support incorporates some means of exploiting the modularity in a specification in order to structure the proofs of these properties.

Computerised proof checkers provide support — type-checking, environment management, tactics, *etc* — for carrying out proofs within specific formal systems. One of the main areas of application for which these tools are intended is assisting with the proof obligations engendered by the use of formal methods. However, in order to do this, the logics supported by proof checkers must be enriched with the concepts used in formal methods. The essential requirement is a notion of "specification". Even more can be achieved if is is possible to add definitions of "program", "implementation", and "refinement". Proof checkers which are based on type theory are equipped with a particularly rich and versatile foundation upon which these concepts can be defined.

In this thesis we examine a specific combination of a specification language with an automated proof-checker. The specification language is the well-known and widely used Z notation. The tool with which it is combined is the LEGO proof-checker, which implements an expressive and powerful type theory, the Unifying Theory of dependent types (UTT). The concrete product of our work is an encoding within UTT of a substantial fragment of the Z notation, together with a large collection of theorems and lemmas about this encoding, all of which have been formally verified using LEGO. Several of these theorems concern the module-forming operations in Z, and provide a basis for using LEGO to prove theorems about Z specifications in a modular fashion. We also investigate ways of defining notions of "program", "implementation" and "refinement" that are appropriate to Z.

In the process of devising the encoding of Z in UTT we were led to consider the general topic of representing one formalism within another, and the variety of techniques available for doing so. The encoding process also entailed an in-depth examination of the semantics of the Z notation. The encoding itself may be viewed as an alternate presentation of the semantics of Z, using type theory as the underlying foundation.

1.1 Example: The Z Notation

In this section we introduce the specification language which is the focus of our work, and describe some of the goals which we aim to achieve by integrating this specification language with a general-purpose proof checker.

The Z notation was invented by Abrial [Abr84a,Abr84b,ASM79] around 1979. It has since undergone a series of evolutions and the second draft of a Base Standard for it is still under development. Several case studies have been carried out in Z (*e.g.* [Hay93]) and it has attracted many users in industry. Mike Spivey provided a theoretical foundation for Z in his PhD thesis [Spi88] and has since produced a reference manual[Spi92] (the ZRM). We shall use both of these works as our main references for Z.

Z allows us to construct specifications by putting together basic building blocks called *schemas*. A schema consists of two parts: a *signature*, in which variables are declared, and a *predicate* which places a constraint on the declared variables. Here is an example of a schema whose signature contains two variables x and y, both ranging over natural numbers. The predicate part of this schema states that x is greater than y.

$$\begin{array}{|l}
\multicolumn{1}{l}{S} \\
\hline
x, y : \mathbb{N} \\
\hline
x > y \\
\hline
\end{array}$$

Following the informal discussion in the ZRM, the meaning of a schema is given in terms of structures called *bindings*, which assign values to the variables in the signature of that schema. A schema is considered to denote all the bindings over the signature of that schema which make the schema predicate true. For example, the schema S above denotes all bindings in which the value assigned to x is greater than that assigned to y.

Our first goal in encoding Z in UTT is to provide support for reasoning *within* the logic of Z. By this we mean that we want to be able to show that some property is implied by a given schema. For example, we may wish to show that the schema S implies that $x + z > y + z$, for all values of z.

The Z notation provides several operations which allow schemas to be put together to build specifications. One example is the operation of schema conjunction. If T is the following schema,

$$\begin{array}{|l}
\multicolumn{1}{l}{T} \\
\hline
x, z : \mathbb{N} \\
\hline
x < z \\
\hline
\end{array}$$

then the conjunction, $S \wedge T$ is a schema whose signature is obtained by *joining* the signatures of S and T and conjoining their predicates:

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{SandT}} \\
\hline
x, y, z : \mathbb{N} \\
\hline
x > y \land x < z \\
\hline
\end{array}$$

Our second goal is to support reasoning about specifications at the level of modules. To achieve this we shall need theorems about the module-level operations such as schema conjunction. For example, we would like to prove that if two schemas both have some property P, then their conjunction also has the property P.

The Z notation incorporates conventions for specifying state-changing operations. The state after an operation is designated by variables decorated with the symbol $'$, while the state before is designated by undecorated variables. For example, the following schema describes an operation which increments a variable x:

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{Inc}} \\
\hline
x, x' : \mathbb{N} \\
\hline
x' = x + 1 \\
\hline
\end{array}$$

The increment operation can be implemented in a Pascal-like, imperative programming language, as follows:

```
procedure inc (var x: nat);
begin
   x := x + 1
end
```

The formalism of Z does not include any programming language. Our third goal is to make specifications and programs a part of the same formalism, so that it is possible to formally define an *implementation* relationship between the two. We can hope to achieve this by using type theory because this provides a notation in which both specifications and programs can be represented.

1.2 Overview

Chapter 1 continues with a description of the tools used for the work in this thesis (Section 1.3). These are the type theory UTT and its implementation in the LEGO proof-checker. The chapter concludes with a description of related work done by others (Section 1.4). This includes projects aimed at providing support for reasoning about modular specifications, particularly where this support involves the use of computerised tools for proof-checking. We also look at projects which proceed in the opposite direction, starting with a general-purpose proof-checker which is then adapted to the application of reasoning about specifications.

In Chapter 2 we examine a variety of representation techniques that are available for representing a language within the logic of a theorem-prover. We see that the choice of technique for a particular application depends on several things including: the expressive capabilities of the logic of the theorem-prover; the style of the definition of the language to be embedded; the intended use of the embedding.

The bulk of our embedding of Z in UTT is presented in Chapters 3, 5, and 6. The embedding is illustrated by a running example which is presented in parallel, in Chapters 4 and 7.

Chapter 3 begins with the definition of a reduced version of Z, called Z$'$, in which it is possible to distinguish between a "core" language and a "module" system. We then examine different techniques for representing Z$'$ in UTT, bearing in mind the issues discussed in Chapter 2. The basic semantic object we shall use, the schema is defined, and some of the details are worked out. These include defining well-formedness conditions on schemas, and formalising a notion of model which most closely captures the semantics of Z.

Our main specification example, introduced in Chapter 4, is adapted from an example in the ZRM. To encode the example we see that we must first develop a representation of finite set theory within UTT. The first schema of the example is then encoded, and LEGO to verify that it satisfies the well-formedness condition defined in Chapter 3.

Chapter 5 deals with the representation of the "logical" operations on schemas provided in the Z notation (conjunction, disjunction, *et c*). These operations are defined in UTT, and LEGO is then used to prove several theorems about them. The key theorems concern the relationship between the operations and the notions of model defined in Chapter 3. These results provide a means of reasoning about (encoded) Z specifications at a modular level. Other results (or observations about non-results) show some of the consequences of our decision to encode Z within a *constructive* type theory.

In Chapter 6 we discuss the Z conventions for specifying state-changing operations. We explain these conventions and then show how they may be encoded in UTT as operations upon schemas. We use LEGO to prove some theorems about the encoded operations, mainly showing that they preserve the well-formedness condition.

Chapter 7 continues the example introduced in Chapter 4. The encodings of the schema operations are used to encode a number of compound schemas, and LEGO is then used to prove several theorems about the encoded specification. Some of these are routine proofs, showing that the schemas introduced are well-formed. Two of the theorems (Theorems 55 and 57) illustrate how our encoding can be used to support reasoning about Z specifications. The first is a formalisation of a proof in the ZRM. The second is an example of a modular proof which makes use of the metatheorems proved in Chapters 5 and 6.

In Chapter 8 we speculate about possible extensions of our work, laying down the basis for definitions of "implementation" and refinement for (encoded) Z specifications. We begin by identifying a UTT type which provides a notion of "program" compatible with the way in which Z schemas are represented. We describe a simple programming language and show how programs may be translated into our chosen UTT type. Some examples are presented to illustrate the translation. We then define an implementation relationship between specifications and programs, and compare this with the way that implementations are dealt with in the ZRM. We also examine the way that the ZRM handles refinement of specifications, and speculate about a notion of refinement for Z that is closer to that used in other specification languages such as VDM.

Concluding the main body of the thesis, Chapter 9 contains some observations about Z, UTT and LEGO.

Appendices A-D give the definitions and descriptions of LEGO proofs referred to in the main text. Appendix E discusses the relationship between the semantics for Z defined by our encoding into UTT and the official semantics presented in [Spi88].

1.3 Introduction to the tools: UTT and LEGO

The tools which we shall use are the type theory UTT ([Luo90,Luo94,Gog94, Gog95]) and its implementation in the LEGO proof-checker ([LP92,Pol95,LEG95]). In this section we shall briefly describe UTT and LEGO; the cited references should be consulted for complete information.

1.3.1 UTT

The type theory UTT is an extension of the Calculus of Constructions ([CH88]) with dependent sum types and a predicative hierarchy of type universes and inductive types. Our understanding of UTT is based on an interpretation suggested by Luo, in which the type theory is thought of as consisting of two layers. The first of these is a single type universe, named *Prop*. Types which lie in the universe *Prop* are interpreted as propositions, using the "propositions as types" paradigm described in [How80]. It is possible to encode an intuitionistic, higher-order logic within the universe *Prop*. The encoding that is used in the LEGO system is described in Appendix D.

The second layer in UTT consists of an infinite hierarchy of type universes, Type_i. We shall think of this layer as representing all the objects in our universe of discourse: *e.g.* datatypes, programs, specifications, sets *et c*. The type universes are cumulative: all objects in Type_i are also contained in Type_{i+1}. In addition Type_0 contains the universe *Prop*. For simplicity of presentation, we shall use the name *Type* to refer to the entire hierarchy.

Both layers of UTT can be extended by means of inductive type definitions. We shall say more about this in Section 1.3.4.

1.3.2 LEGO

LEGO [Pol95] is a proof development system implemented by Randy Pollack. The LEGO Reference Manual [LP92] is the main documentation for users of this system. Figures 1–1, 1–2 and 1–3 show the notation which we use in this thesis for representing LEGO terms. Figure 1–1 gives the notation used for those LEGO terms which are concrete representations of the terms of UTT, plus other useful features (such as local definitions and implicit argument designations) some of which are described below. Figure 1–2 shows our notation for a number of frequently-used inductively-defined types. The definitions of these types are taken from the LEGO library and are described in Appendix D. Figure 1–3 shows the syntax that we used to represent the logic that is encoded in UTT. The details of this encoding are described in Appendix D.

The salient functions of the LEGO system include the following:

- Type-checking and partial type inference for UTT terms.
- Automatic handling of definitions, contexts, substitutions, *et c.*
- Built in tactics and environment control to support the incremental definition of UTT terms of a desired type. This feature is the basis of the proof-development function of the LEGO system, so we shall say more about it below.

Notation	Explanation
Prop	the universe *Prop*
Type	the hierarchy of universes $Type_i$
$\lambda x{:}\mathrm{T}.\,\mathrm{M}$	function with explicit argument
$\lambda x\,\vert\,\mathrm{T}.\,\mathrm{M}$	function with implicit argument
M N	function application to explicit argument
M|N	function application to implicit argument
$\Pi x{:}\mathrm{T_1}.\,\mathrm{T_2}$	dependent function space
$\mathrm{T_1} \rightarrow \mathrm{T_2}$	non-dependent function space
$[x = \mathrm{M}]$	local definition
$[x : \mathrm{T}]$	assumption (global λ abstraction)

Figure 1–1: Notation used for LEGO terms

- Tactics for introducing inductive type definitions.

- Typical ambiguity. The LEGO system allows the user to refer to the hierarchy of universes $Type_i$ as a single entity *Type*. The system computes appropriate values of i and detects cycles if they occur.

- Argument synthesis. When a function is applied to several arguments, it may be possible to infer the values of some of these arguments by looking at the values of the others. The LEGO system allows the user to exploit this fact by designating some arguments as implicit. This is done by using a vertical stroke | rather than a colon : in the definition of terms which can take an implicit argument. Figure 1–1 gives an example of what this looks like for function definitions.

1.3.3 Working with LEGO

When we start up the LEGO system we find ourselves in a mode called *LEGO mode*. In this mode we may do several things: add definitions to the LEGO

Notation	Explanation
$\Sigma\, x : \mathrm{T}_1 .\, \mathrm{T}_2$	dependent sum type
(M,N)	pairing
M.**1**	first projection
M.**2**	second projection
nat	natural number type
0	zero
suc	successor
nat_rec, *nat_iter*	elimination operators (See Section 1.3.4)
bool	boolean type
true	true
false	false
if	See Appendix D.
$\mathrm{M} + \mathrm{N}$	sum type
in_1	left injection
in_2	right injection
case	See Appendix D.
unit	one-element type
void	element of *unit*
list T	lists of elements of type T
cons	cons
$[x_1, \ldots, x_2]$	list construction
list_rec	See Appendix D.

Figure 1–2: Notation used for common inductively defined terms

Notation	Explanation
$P_1 \wedge P_2$	conjunction
$P_1 \vee P_2$	disjunction
$\neg P$	negation
$P_1 \Rightarrow P_2$	implication
$\forall x{:}T_1.\,T_2$	universal quantifier
$\exists x{:}T_1.\,T_2$	existential quantifier

Figure 1–3: Notation used for logic

context; add or discharge assumptions; load library files; enter *proof mode* by using the command `Goal`.

In proof mode we have a *goal*, which is a type, and our job is to construct a term of that type. To do this we may make use of the tactics provided by the LEGO system. Most often we will use a powerful tactic called `Refine`. We do not give a complete description of the behaviour of this tactic, but we will illustrate its use in the small example below. In proof mode we can also add definitions to the LEGO context but we cannot add new assumptions. When the proof is completed the LEGO system re-enters LEGO mode, where the user may then save the proof term as a definition in the LEGO context.

For example, let us look at one way of proving the following simple proposition in LEGO.

$$\forall T{:}\mathit{Type}.\ \forall P, Q{:}T \rightarrow \mathit{Prop}.\ \forall x{:}T.\ ((P\ x) \wedge (Q\ x)) \Rightarrow ((Q\ x) \wedge (P\ x))$$

We shall begin by adding some declarations to the LEGO context:

$$T : \mathit{Type}$$
$$P, Q : T \rightarrow \mathit{Prop}$$
$$x : T$$
$$H : (P\ x) \wedge (Q\ x)$$

Now we use the `Goal` command and tell the system that we would like to prove the term $(P\ x) \vee (Q\ x)$. The system enters proof mode and shows us our goal:

$$? : (Q\ x) \wedge (P\ x)$$

We begin by using the *and elimination* tactic, applied to the assumption H. The effect of this tactic is to introduce two new names into the LEGO context.

$$H_1 : P\ x$$
$$H_2 : Q\ x$$

Next we use the *and introduction* tactic. The effect of this is to replace our goal by the following two subgoals:

$$?_1 : Q\ x$$
$$?_2 : P\ x$$

The *current goal* is the first one displayed (in this case, $?_1$.) (If we wish to work on a different subgoal, we must explicitly change the current goal by using the `Next` command.) To solve the current goal we use the `Refine` tactic applied to H_2. This checks that H_2 is capable of proving the current goal. In general, `Refine` may generate further subgoals but this is not the case in our simple example.

Goal $?_2$ becomes the new current goal, and is proved by refining by H_1. This completes the proof, so the LEGO system returns to LEGO mode.

The proof process has the effect of constructing a $\boldsymbol{\lambda}$-term, the *proof object*. In our case, the proof object is the following:

$$pair\ (snd\ H)\ (fst\ H)$$

This can be stored in the LEGO context by using the `Save` command. We shall store our little proof under the name *And_commutes*.

Next, we *discharge* the assumptions we made before commencing the proof by giving the command `Discharge` T. This removes from the LEGO context the identifier T as well as all identifiers *declared* after the declaration of T. All definitions made after the declaration of T are $\boldsymbol{\lambda}$- abstracted over those declarations on which they depend. In our case, the only definition is that of the proof object, *And_commutes*. Discharging T changes the definition of *And_commutes* to the following:

$$\begin{aligned}&\lambda T\!:\!\mathit{Type}.\ \lambda P,Q\!:\!T\rightarrow \mathit{Prop}.\ \lambda x\!:\!T.\\ &\qquad \lambda H\!:\!(P\ x)\wedge(Q\ x).\\ &\qquad\quad \mathit{pair}\ (\mathit{snd}\ H)\ (\mathit{fst}\ H)\end{aligned}$$

The type of this term is the following:

$$\forall\, T\!:\!\mathit{Type}.\ \forall\, P,Q\!:\!T\rightarrow \mathit{Prop}.\ \forall\, x\!:\!T.\ ((P\ x)\wedge(Q\ x))\Rightarrow((Q\ x)\wedge(P\ x))$$

This is the goal which we originally set out to prove.

The LEGO User's Manual should be consulted for details about the tactics available for doing proofs in LEGO. In general, there are introduction and elimination tactics for all of the logical operators in Figure 1–3.

1.3.4 Inductive Types in LEGO

A theoretical foundation for adding inductive type definitions to UTT was developed by Goguen and Luo [Gog94,Gog95,Luo92,Luo94]. This was then implemented in LEGO by Claire Jones. As an example, the inductive type of natural numbers is defined by issuing the following command:

$$\begin{aligned}&\text{Inductive[}\ \mathit{nat} : \mathit{Type}\ \text{]}\\ &\text{Constructors[}\ \mathit{zero} : \mathit{nat},\\ &\qquad\qquad\quad \mathit{suc} : \mathit{nat}\rightarrow \mathit{nat}\ \text{]}\end{aligned}$$

This command has the effect of adding the names *nat*, *zero*, and *suc* to the context. It also automatically produces an *elimination schema* for the new inductive type, and *computation rules* which dictate the behaviour of the elimination schema. In the example given, the elimination schema that is produced is the following:

$$\begin{aligned}&\mathit{nat\_elim}\\ &: \mathbf{\Pi}\, T\!:\!\mathit{nat}\rightarrow \mathit{Type}.\ (T\ \mathit{zero})\rightarrow(\mathbf{\Pi}\, x\!:\!\mathit{nat}.\ (T\ x)\rightarrow(T\ (\mathit{suc}\ x)))\rightarrow(\mathbf{\Pi}\, x\!:\!\mathit{nat}.\ T\ x)\end{aligned}$$

The computational rules which are produced are as follows. These define the behaviour of *nat_elim*.

$$
\begin{array}{l}
[T : nat \rightarrow Type] \\
[H : T\ zero][H_1 : \Pi x{:}nat.\ (T\ x) \rightarrow (T\ (suc\ x))] \\
[x : nat] \\
\qquad nat\_elim\ T\ H\ H_1\ zero \Longrightarrow H \\
\qquad nat\_elim\ T\ H\ H_1\ (suc\ x) \Longrightarrow H_1\ (nat\_elim\ T\ H\ H_1\ x)
\end{array}
$$

So far, we have described what is automatically produced by the LEGO system when a new inductive type is introduced. Generally, a user will wish to make some additional definitions, as described below.

The elimination schema can be used to define an induction principle for the new inductive type. For *nat*, the induction principle has the following type.

$$
\begin{array}{l}
nat\_ind \\
: \forall P{:}nat \rightarrow Prop.\ (P\ zero) \Rightarrow (\forall x{:}nat.\ (P\ x) \Rightarrow (P\ (suc\ x))) \Rightarrow (\forall x{:}nat.\ P\ x)
\end{array}
$$

We can also define restricted versions of the elimination schema. These include the *recursor* (*nat_rec*) and the *iterator* (*nat_iter*):

$$
\begin{array}{l}
nat\_rec : \Pi T{:}Type.\ T \rightarrow (nat \rightarrow T \rightarrow T) \rightarrow nat \rightarrow T \\
nat\_iter : \Pi T{:}Type.\ T \rightarrow (T \rightarrow T) \rightarrow nat \rightarrow T
\end{array}
$$

Finally, the elimination schema may be used to define a decidable equality on the inductive type:

$$
\begin{array}{l}
nat\_eq \stackrel{def}{=} nat\_iter\ (nat\_iter\ true\ (\lambda\_{:}bool.\ false)) \\
\qquad\qquad\qquad\qquad (\lambda prev{:}nat \rightarrow bool.\ nat\_rec\ false\ (\lambda x{:}nat.\ \lambda\_{:}bool.\ prev\ x)) \\
: nat \rightarrow nat \rightarrow bool
\end{array}
$$

1.4 Related work

1.4.1 Tool support for modular specification languages

Other work aimed at providing proof support for Z includes: the ProofPower system [Jon92,Pro2] which is based upon a deep embedding of Z in a HOL-like theorem-prover with a user interface that closely resembles Z notation — this embedding is not deep enough, however, to allow facts like the commutativity of schema conjunction to be proved [BG94]; a shallow embedding of Z in the HOL theorem-prover [HOL95] carried out by Bowen and Gordon [BG94]; the work of Martin [Mar93] who has encoded W, a logic for Z, in the metalogical framework 2OBJ; the Z/EVES project [Saa91] which uses a theorem prover for ZF set theory; the Balzac project [Har91] at Imperial Software Technology.

The Vienna Development Method[Jon86] incorporates a specification language which is similar to Z in that it is *model-oriented* (a specification is viewed as a description of an abstract machine). However, the structuring mechanisms of the two notations are very dissimilar. VDM and Z are compared entertainingly in [HJN93]. The Mural system [JJ+91] provides a support system for VDM.

Algebraic specification languages are based on a different approach in which a specification is viewed as a collection of axioms describing the behaviour of a set of operations. The specification language CLEAR [BG81] extends this approach by providing a number of specification-building operations which allow the modular construction of specifications. Harper, Sannella and Tarlecki [HST89] have looked at using the Logical Framework to provide a support system for specification languages similar to CLEAR. A different means of achieving modularity is explored in Extended ML [San89,ST89,KST94], in which specifications are structured using constructs inspired by the module system of the programming language Standard ML.

The Prototype Verification System [CO+95], developed at SRI International Computer Science Laboratory, provides a specification language integrated with support tools and a theorem-prover. The Larch system [GGH93,Lar94] incorporates a specification language together with *interface languages* which allow translation to a variety of specific programming languages. The system includes a theorem prover, LP.

The RAISE formal method [Geo91,Rai95] provides a specification language RSL together with a development method. RSL is a "wide-spectrum" language which supports a variety of specification styles, including that of Z. Tool support includes a justification editor which provides assistance with formal proof.

1.4.2 Proof-checkers used for specification

Rod Burstall and James McKinna have worked on an approach to program specification and verification in the type theory ECC (a precursor to UTT) using constructs called "deliverables" [BM91,McK92]. Zhaohui Luo has looked at methods for expressing structured specifications in ECC using the Σ-type as the basic specification module [Luo91].

The Coq project [Coq95] centres upon the Coq proof assistant [DFH+93] which implements a constructive type theory very similar to UTT. One of the main applications to which this is being applied is the development of an environment for formal program development incorporating a specification language (Gallina) and a modular programming language (FML).

Chapter 2

Representation Techniques

In this chapter we discuss the general topic of representing languages within the logic of a theorem-prover. This is a subject which has been studied very formally in work on Logical Frameworks [HHP87,AHM87]. Our treatment of this topic will be more informal, because we wish to consider and compare a very broad class of representation techniques. The issues which arise are related to traditional issues in the logic of translation, but also include new concerns created by the translation process itself, and by the intended use of the final translation. (For example, we are interested in the applicability of automated tools to mechanise parts of the translation process. The availability of automatic type-checking, user-defined tactics, decision procedures, *et c.* all play a role here.)

We shall begin by stating a number of criteria by which representation techniques may be rated. We then give a small example of a language and discuss several ways of representing this language within UTT, evaluating the pros and cons of each method according to our defined criteria. In some cases we shall use other examples which specifically illustrate the power of a particular embedding technique, or the particular kind of language description for which that technique is best suited. We shall also see how the expressive type theory implemented in LEGO makes available to us embedding techniques that are more powerful than those available in the HOL proof-checker.

We focus our attention on representation techniques that involve only definitional extensions of the logic of the theorem-prover, or conservative extensions such as the introduction of a new inductive type. By avoiding axiomatisations we avoid the possibility of introducing logical contradiction.

Mike Gordon *et al* have considered the problem of embedding hardware description languages in the HOL theorem prover [Gor88] and have coined the terms "shallow" and "deep" [BG+92] to describe two opposing styles of representation. We shall consider how these terms may be interpreted with respect to type theory. We shall see that these two terms are not adequate to describe the gamut of representation styles possible in type theory.

We use the term "language definition" broadly and informally. A language definition may consist of a *syntax* definition and perhaps an *equational theory* on that syntax. There may be *types* and *well-typedness rules* as well. There may be *semantic objects* and *operational rules* or *denotation functions* which relate the syntax to the semantic objects. Or the language may be a logic, and there may be a notion of *model* and rules giving the *satisfaction relationship* between models and terms in the language. All of the highlighted terms are possible components of a language description. In embedding a language we must decide how (or whether) each of these components is to be represented in the embedding.

2.1 Rating embeddings

We state a number of informal criteria by which we can rate different embedding techniques.

Adequacy Can the embedding technique capture all the aspects of the language description? How accurate is the representation? For instance, suppose we represent the syntax Syn of a language by some object `Syn` in the logic of the theorem prover. Are the two classes Syn and `Syn` exactly equal? Are there spurious objects in the representing class `Syn`, or are there not enough objects to represent all the terms in Syn? Similar questions about adequacy may be asked about all the classes in any given language description.

Note that we do not intend to define the term adequacy as strictly as the above example might suggest. The meaning of the term must be decided *ad hoc* for any given embedding. However, certain combinations of language description style and embedding technique make it easier to define adequacy and to verify it.

Ease of use How easy is it for a user to translate terms of the object language into the language of the embedding, and then to use the embedding to reason about the relationships (e.g. equality, type-checking, evaluation, or satisfaction) defined in the object language.

Expressiveness To what extent does the embedding make it possible to reason about the meta-theory of the object language.

2.2 Examples of embedding techniques

We shall use as an example a very small typed programming language called L which is defined in Figure 2–1.

2.2.1 Embedding 1: A "deep embedding"

In the paper [BG+92], the deep embedding approach to representing a hardware description language (HDL) in the HOL theorem-prover is explained as follows:

> Represent the abstract syntax of HDL programs by terms, then define *within the logic* semantic functions that assign meanings to the programs.

This description can be applied to embedding the language L in the LEGO theorem-prover. In a deep embedding all parts of the language description are

Types

LType = `nat` | `bool` | (LType `->` LType)

Terms

LTerm = `true` | `false` | `0` | `suc` | LTerm `<` LTerm | (LTerm LTerm)

Typing rules

$$\frac{}{\texttt{true}:\texttt{bool}} \qquad \frac{}{\texttt{false}:\texttt{bool}} \qquad \frac{}{\texttt{<}:\texttt{(nat->(nat->bool))}}$$

$$\frac{}{\texttt{0}:\texttt{nat}} \qquad \frac{}{\texttt{suc}:\texttt{nat->nat}} \qquad \frac{\mathrm{tm1}:(\mathrm{ty1}\texttt{->}\mathrm{ty2}) \quad \mathrm{tm2}:\mathrm{ty1}}{\mathrm{tm1}\ \mathrm{tm2}:\mathrm{ty2}}$$

Semantic objects

Value = `true` | `false` | `zero` | `suc` Value

Evaluation rules

$$\frac{}{\texttt{true}\Rightarrow\texttt{true}} \qquad \frac{}{\texttt{false}\Rightarrow\texttt{false}} \qquad \frac{}{\texttt{0}\Rightarrow\texttt{0}} \qquad \frac{\mathrm{tm}\Rightarrow \mathrm{v}}{(\texttt{suc}\ \mathrm{tm})\Rightarrow\texttt{suc}\ \mathrm{v}}$$

$$\frac{}{\texttt{(0 < 0)}\Rightarrow\texttt{false}} \qquad \frac{}{\texttt{(0 < (suc}\ \mathrm{tm}\texttt{))}\Rightarrow\texttt{true}} \qquad \frac{\mathrm{tm1} < \mathrm{tm2}\Rightarrow \mathrm{v}}{(\texttt{suc}\ \mathrm{tm1})\ \texttt{<}\ (\texttt{suc}\ \mathrm{tm2})\Rightarrow \mathrm{v}}$$

Figure 2–1: Definition of the language L represented *within* the logic of the theorem prover. Concretely, this can be done by defining inductive types in LEGO to represent the terms and types of L. The evaluation rules and typing rules can be represented as inductive relations. Part of such an embedding is shown in Figure 2.2.1. We think of this embedding as a *full, internal description* of the language L in UTT.

Remarks

1. In embedding 1 all the entities that make up the language description are represented by some aspect of the logic of the theorem prover. For instance, the type-checking rules of L are represented within the logic (by the relation *has_type*) as opposed to being implemented outside the logic by some pre-processor which would filter out ill-typed terms before passing them to the theorem-prover.

2. The objects in the logic that represent L are all *internal* to the logic. By this we mean that we can talk about these objects within the logic: we can quantify over the types *LType*, *LTerm*, *Eval* t v (where t and v have types *LTerm* and *Value*, respectively), *etc*. Being able to quantify over these things allows us to express meta-theoretical statements about the language L. We shall see that property 2 does not necessarily follow from property 1.

3. The internal objects that represent L are all inductively defined. This gives us elimination rules which we can use for computing with these objects or proving results about them.

4. This embedding is very cumbersome to set up, as Figure 2-2 shows. It is also cumbersome to use, because even the simplest term in the embedded language becomes very long and ugly when translated into the embedding, since its abstract syntax must be written out in full. This problem could be ameliorated, to some extent, by a helpful user interface. However, when used in a theorem prover like LEGO which provides the ability to type-check and normalise terms automatically, this embedding

Representation of types, terms and values

Inductive [$LType$: $Type$]

Constructors [$nat\_ty, bool\_ty : LType,\ arrow\_ty : LType \rightarrow LType \rightarrow LType$]

Inductive [$LTerm$: $Type$]

Constructors [$true\_sy, false\_sy, zero\_sy, suc\_sy, lt\_sy : LTerm,$
$app\_sy : LTerm \rightarrow LTerm \rightarrow LTerm$]

Inductive [$Value$: $Type$]

Constructors [$true\_v, false\_v, zero\_v : LTerm,$
$suc\_v : Value \rightarrow Value$]

Representation of evaluation relation

Relation [$Eval : LTerm \rightarrow Value \rightarrow Prop$]

Constructors [$eval\_true : Eval\ true\_sy\ true\_v,$

$eval\_false : Eval\ false\_sy\ false\_v,$

$eval\_zero : Eval\ zero\_sy\ zero\_v,$

$eval\_app\_suc : \forall\, tm\!:\!LTerm.\ \forall\, v\!:\!Value.$

$(Eval\ tm\ v) \Rightarrow (Eval\ (app\_sy\ suc\_sy\ tm)\ (suc\_v\ v)),$

$eval\_app\_lt\mathbf{1}$:

$Eval\ (app\_sy\ (app\_sy\ lt\_sy\ zero\_sy)\ zero\_sy)\ false\_v,$

$eval\_app\_lt\mathbf{2} : \forall\, tm\!:\!LTerm.$

$Eval\ (app\_sy\ (app\_sy\ lt\_sy\ zero\_sy)\ (app\_sy\ suc\_sy\ tm))$

$true\_v,$

(∗*One rule omitted*∗)

The typing rules are represented in a similar way by a relation *has_type*.

Figure 2–2: Embedding 1

has a more serious disadvantage. The user must reason explicitly about the typing and evaluation of terms rather than having this done for free by the theorem-prover. Of course, this may not be seen as a disadvantage if the goal is to reason about the formal metatheory of the embedded language, rather than to reason about the behaviour of specific programs.

5. An advantage of embedding 1 is that it is very easy to compare it with the original definition of L in order to verify that L has been encoded accurately. The inductive types and inductive relations of the embedding mirror the original grammar and evaluation and typing rules of the operational semantics account of L. This advantage depends on the style of the language description and may be reduced for languages (such as Z) whose description is not given in an inductive style.

Embedding 1 provides a great deal of expressiveness thanks to points 1, 2, and 3. It is easy to be convinced of its adequacy. However, this embedding is relatively difficult to use.

Examples of this embedding style

An example is Altenkirch's encoding of Girard's System F in LEGO [Alt93], which he uses to formalise a proof of strong normalisation for System F.

Another example is an embedding in HOL of the dynamic semantics of the programming language Standard ML (SML) [VG94,MG94]. (Donald Syme [Sym93] has carried out a similar project using essentially slightly different techniques.) In this embedding, the syntactic classes, semantic objects, and evaluation relations of SML were all represented explicitly via inductive structures in HOL. An extension of SML with higher-order functors[MT94] was encoded in a similar fashion, and the encoding was then used to formally study the relationship between the two languages. This example illustrates the great expressive power of the deep embedding technique, and its usefulness when appropriately applied.

2.2.2 Embedding 2: A "shallow embedding"

The term "shallow embedding" is defined by Gordon *et al* as follows:

> Only define semantic operators in the logic and arrange that the user-interface parse input from HDL syntax directly to semantic structures, and also print semantic representations in HDL syntax.

It is not so clear how to apply this description to embeddings of L in type theory. One interpretation would be to represent only the semantic objects of L within the type theory, while all other elements of the language description are handled by an external user-interface. This interface would read LTerms, type-check them according to the typing rules of L, evaluate them to obtain Values, and then pass these values to the theorem-prover. The theorem-prover can only be used to prove results about Values. This can be useful if the values are the main object of interest. For instance we may be more interested in the functions computed by programs written in some particular programming language than in the syntax of the programs themselves.

We can think of this embedding as *modelling the universe of discourse* of the language L within UTT.

It is instructive to look at a couple of possibilities for representing Values in a shallow embedding of L, because they illustrate some of the issues around adequacy.

1. One possibility is to embed Value as an inductive type, just as in the deep embedding. In one sense this gives us the most "adequate" representation of Value, because the representing object (the type *Value*) is isomorphic to the class Value in the definition of the language L.

2. Another possibility is to use the LEGO types *nat* and *bool* to represent Values:

   $$Value \stackrel{def}{=} nat + bool$$

In one sense this embedding is not adequate since the representing type *nat* + *bool* does not contain representations of all possible Values. (Values such as *suc false* are not represented.) However, the values represented are exactly those obtained by evaluating well-typed expressions, so in this sense this is an adequate representation.

2.2.3 Embedding 3: A grammatical translation

We now consider an embedding which has property 1 of Section 2.2.1 but not property 2. This embedding can be thought of as a *grammatical translation*: well-formed terms of the language L are translated to terms of UTT.

For this translation we do not make any definitions within the logic of the theorem-prover. Rather we define, externally to the theorem prover, translation functions which translate individual LTerms, LTypes, and Values into the logic of the theorem-prover. In theorem-provers like HOL or Coq, a programming language is provided which acts as a meta-language to the logic of the theorem-prover. This may be conveniently used for defining these translations. Unfortunately, the LEGO proof-checker does not provide this facility.

Each type of L will be translated to a type in the logic. The type `nat` is translated as *nat*, `bool` as *bool et c.* Terms are translated similarly: `true` is translated as *true*, `0` as $\mathbf{0}$, `<` as $<$, function application in L by function application in the logic, and so on. Type-checking in L is translated by type-checking in the logic of the theorem-prover. Similarly, evaluation in L is translated by evaluation in the theorem-prover.

Remarks

1. All the parts of the definition of the language L are represented within the logic of the theorem-prover. However, the representing objects are not necessarily internally named within this logic. This reduces the express-

iveness of the embedding: for instance, we cannot express the statement that `zero` does not have the LType `bool`.

2. The expressive power of this embedding depends on the capability of the logic of the theorem-prover to reflect those aspects of itself that are being used for the embedding. For instance, the logic of LEGO contains type universes, which reflect the types in the logic. These types are our representation of the types in L. Since the logic allows us to quantify over type universes, the embeddings allow us to quantify over the types of L. It is true that there are a lot of spurious types present (The type universe *Type* contains far more than just *nat*, *bool* and the functional hierarchy over these!) but we shall see in the next section that the ability to quantify over types can be useful all the same.

3. From the point of view of adequacy, this style of embedding depends a lot on there being a good match between the embedded language and the logic of the theorem-prover. For instance, if we were to extend the language L with an operator for general recursion, this kind of embedding of L would no longer be possible in UTT.

4. This kind of embedding is easy to set up and to use. For languages that are non-trivial, this can be a huge advantage over the "deep embedding" technique. For instance, consider adding abstractions to L. The "deep" embedding technique would now require us to explicitly define the handling of substitution. With embedding style 3, the theorem prover automatically handles substitution. (A similar point is raised by Constable and Howe [CH89] with reference to the logic of the proof-checker NuPrl [Con86].).

2.2.4 An embedding example: value-passing CCS in LEGO

This embedding illustrates a point that was briefly mentioned in the previous section: having type universes in UTT enables us to devise representation techniques that are not possible in theories without type universes. We examine an embedding that was done in the HOL system (which lacks type universes) and show how a deficiency in that embedding can be corrected by moving to a system with type universes and dependent $\boldsymbol{\Pi}$ types.

In [Nes94], Monica Nesi describes an embedding of value-passing CCS[Mil89] in the HOL theorem-prover. This is a process algebra in which processes can input and output values through named ports. A fragment of the grammar of value-passing CCS expressions is shown below. Here the labels v, x, and e range over constants, variables, and expressions, all taken from some domain of values V. The label a stands for an input port, and the label $\overline{a}$ for an output port. The symbol $\boldsymbol{\tau}$ is called the silent action and its meaning is irrelevant to this discussion.

$$E ::= a(x).E \;\mid\; \overline{a}(e).E \mid \boldsymbol{\tau}.E$$

In the original definition of value-passing CCS, the values are restricted to a single domain V, just as described above and as implemented in Nesi's work. However, Milner mentions that in real applications values of many different types may be used. It is difficult to extend Nesi's embedding to a many-sorted version of value-passing CCS because of the inherent limitations of HOL. However, this extension can be done easily if the embedding is done in a type theory with universes and dependent types.

Nesi's embedding of CCS in HOL is essentially a grammatical translation (like embedding 3) in which the domain of values V is represented by a type parameter $\boldsymbol{\alpha}$. The phrase classes of CCS are translated by defining appropriate inductive types parameterised by $\boldsymbol{\alpha}$. For instance, labels and actions are represented as follows:

$$\begin{aligned} &\mathit{labelv} = \mathbf{in}\, n \mid \mathbf{out}\, n \\ &\mathit{actionv} = \mathbf{tauv} \;\mid \mathbf{labelv}\; \mathit{labelv}\; \boldsymbol{\alpha} \end{aligned}$$

(To simplify things slightly we have assumed that ports are to be named by natural numbers, n.) The type *labelv* consists of input or output ports, named by natural numbers. Actions are either the silent action or a label followed by a value expression of type α. The entire definition of CCS terms is parameterised by the type variable α representing the type of values.

There are two possibilities for moving from this definition to a multi-sorted value passing CCS. One method is to use the definition above, instantiating the type α as the disjoint sum of all the types that are required. This embedding is still essentially single-sorted: there can be no such thing as a port which passes only natural numbers while another passes only booleans. Both ports would have to pass values of the same type which would be whatever α is instantiated to. The second alternative is to define a distinct class of labels for each type that is required. This results in a huge and cumbersome embedding where many functions must be duplicated for each type. Both methods also have the deficiency that new types may not be introduced to the embedding dynamically: once α is instantiated to a particular sum type, or a particular class of typed labels has been chosen, no new types may be added.

Another possible way of representing the multi-sorted calculus in HOL would be to move to a full internal description (like embedding 1). This would require us to model the types and type-checking of value-passing CCS explicitly in HOL. This approach gives us an embedding that is difficult and awkward to use, because we are forced to reason explicitly about type-checking.

By contrast if we use UTT it becomes very easy to change the definition of label slightly so that labels carry around the type of values that they are allowed to pass. Here is a translation into UTT of the original, single-sorted definitions of labels and actions:

Inductive $[labelv : Type]$
Constructors $[in, out : nat \rightarrow labelv]$

Inductive $[actionv : Type \rightarrow Type]$
Constructors $[tauv : \Pi\, a{:}Type.\ actionv\ a]$
$[labvv : \Pi\, a{:}Type.\ labelv \rightarrow a \rightarrow (actionv\ a)]$

To extend this to a many-sorted system we simply tag labels with the type that they are allowed to pass:

$$labelv\mathbf{2} \stackrel{def}{=} labelv \times Type$$

Next, we define actions in the following way which specifies that the value that passes through a particular port must be of the type with which that port is tagged:

Inductive $[actionv\mathbf{2} : Type]$
Constructors $[tauv\mathbf{2} : actionv\mathbf{2}]$
$$[labvv : \Pi\, port : labelv\mathbf{2}.port.\mathbf{2} \to actionv]$$

Now suppose we wish to represent agents which have some ports that pass natural numbers and some that pass booleans. With the single-sorted system we have to make all ports pass the type $(nat + bool)$. Here is how we would represent the actions $a(true)$ and $b(\mathbf{0})$. First we define the ports a and b:

$$a \stackrel{def}{=} in\ zero$$
$$b \stackrel{def}{=} in\ one$$

Next, here are the definition of the actions $a(true)$ and $b(\mathbf{0})$:

$$labv\ (nat + bool)\ a\ (in\mathbf{2}\ true)$$
$$labv\ (nat + bool)\ b\ (in\mathbf{1}\ zero)$$

There is nothing to prevent us from inputting $in\mathbf{1}\ zero$ to the port a, or $in\mathbf{2}\ true$ to the port b.

With the multi-sorted version, here are the definitions of these two actions:

$$a\mathbf{2} \stackrel{def}{=} (in\ zero, bool)$$
$$b\mathbf{2} \stackrel{def}{=} (in\ one, nat)$$

$$labv\mathbf{2}\ a\ true$$
$$labv\mathbf{2}\ b\ zero$$

If we tried to input naturals to port a, or booleans to port b, the terms would fail to type-check.

Remarks

We first note that our embeddings of multi-sorted, value-passing CCS in LEGO are all incomplete, since they do not show how the rules of the calculus (or even its full syntax) are represented. We can extend our embedding to the full language by copying the techniques used by Nesi in her encoding.

All of the embeddings discussed represent the syntax of the object language "deeply", as in embedding 1, and therefore provide the ability to quantify over syntactic structures and to express metatheorems about the embedded language. In all of the embeddings the class of types of the object language is represented by the class of types of the metalanguage, as in embedding 3. The embeddings done in LEGO are more expressive because UTT allows us to quantify over all of its types. None of the embeddings allow us to express statements about the typing relationship itself, since this is represented by the typing relationship of the metalanguage, and neither of the metalanguages used provides an internal reflection of its own typing judgements.

Our multi-sorted embedding identifies the types and type-checking of the object language with those of the meta-language. This helps to make the embedding easier to use: type-checking is provided for free, rather than being a proof obligation.

2.2.5 An embedding example: Union types

Our final example is intended to illustrate the kind of embedding techniques we shall develop for the Z notation. Consider the small language T described in Figure 2.2.5. In this language we define a union type constructor $\bigcup$ whose behaviour is specified by three equations. We would like to embed this language in LEGO in order to study its properties.

If we attempt a full internal description like embedding 1, we run into a problem representing the equations on the types of T. The types of T are not freely defined, so they cannot be adequately represented by an inductively

Types

TType = `bool` | `nat` | Ttype $\bigcup$ Ttype

Equations on Types

$\mathrm{T} \bigcup \mathrm{T} = \mathrm{T}$

$\mathrm{S} \bigcup \mathrm{T} = \mathrm{T} \bigcup \mathrm{S}$

$\mathrm{S} \bigcup (\mathrm{T} \bigcup \mathrm{U}) = (\mathrm{S} \bigcup \mathrm{T}) \bigcup \mathrm{U}$

Terms

TTerm = `true` | `false` | `0` | `suc`

Typing rules

$$\frac{}{\texttt{true:bool}} \qquad \frac{}{\texttt{false:bool}} \qquad \frac{}{\texttt{0:nat}} \qquad \frac{}{\texttt{suc:nat->nat}} \qquad \frac{\mathrm{x:S}}{\mathrm{x:S} \bigcup \mathrm{T}}$$

Figure 2–3: A type theory with Union types

defined type. (We could devise an embedding in which the equality of the types in the object language is represented explicitly, rather than being identified with equality in UTT, but this would be very awkward to set up and to use. Alternatively, we could use a type-theory with quotient types, such as NuPrl [Con86].) If we attempt a grammatical translation in the style of embedding 3 we find that there are no objects in the logic that can provide a satisfactory representation of the union type. The nearest thing that we can find is a disjoint sum type, and this does not satisfy the equations of the Union type.

We observe, however, that a restricted version of the disjoint sum type would provide the representation that we are looking for. If we could restrict ourselves to disjoint sums whose summands contain no duplicates and are ordered in some canonical way, then we would obtain a class of types that is isomorphic to the types that we are trying to embed. We can think of this embedding as a *non-literal translation*: the terms of T cannot all be represented directly as terms of UTT, but must be translated indirectly in some cases.

We achieve such a translation by doing the following: First we define the *syntax* of types of T as an inductive type, just as we would for a deep embedding:

$$\text{Inductive } [\mathit{TType} : \mathit{Type}]$$
$$\text{Constructors } [\mathit{bool\_ty}, \mathit{nat\_ty} : \mathit{TType}]$$
$$[\mathit{U\_ty} : \mathit{TType} \to \mathit{TType} \to \mathit{TType}]$$

Next we define a function which maps these syntactic TTypes to the UTT types which represent them. We shall call this function *Typ*. Its type is $\mathit{TType} \to \mathit{Type}$ and its definition is described below:

First we define the base types of T:

$$\text{Inductive } [\mathit{BType} : \mathit{Type}]$$
$$\text{Constructors } [\mathit{bool\_bty}, \mathit{nat\_bty} : \mathit{BType}]$$

We define a computational equality $\mathit{BType\_eq} : \mathit{BType} \to \mathit{BType} \to \mathit{bool}$ and a total order $\mathit{BType\_lt} : \mathit{BType} \to \mathit{BType} \to \mathit{bool}$.

For convenience we shall define a type of non-empty lists:

$$\text{Inductive } [\mathit{nelist} : \mathit{Type} \to \mathit{Type}]$$
$$\text{Constructors } [\mathit{single} : \Pi\, t{:}\mathit{Type}.\, t \to (\mathit{nelist}\ t)]$$
$$[\mathit{more} : \Pi\, t{:}\mathit{Type}.\, t \to (\mathit{nelist}\ t) \to (\mathit{nelist}\ t)]$$

Now we can define a function *get_btypes* which maps a *TType* to a non-empty list made up of all the *BTypes* in that *TType*. We also define two functions $\mathit{remove\_dup} : (\mathit{nelist}\ \mathit{BType}) \to (\mathit{nelist}\ \mathit{BType})$ and $\mathit{sort} : (\mathit{nelist}\ \mathit{BType}) \to (\mathit{nelist}\ \mathit{BType})$. As the names suggest, the first of these removes duplicates from a non-empty list of *BTypes*, using *BType_eq* for comparison, while the second sorts a non-empty list of *BTypes* into, say, ascending order according to the ordering *BType_lt*.

Next we define a function which maps *BTypes* to types in the type theory. We call this function *Typ_BType*. Its type is $\mathit{BType} \to \mathit{Type}$ and its behaviour is very simple: it maps *nat_bty* to *nat* and *bool_bty* to *bool*.

We extend this function to a function *Typ_BType_list* which forms the disjoint sum of all the types obtained by applying *Typ_BType* to all the members of a non-empty list of *BType*s.

Finally we can define the function *Typ*:

$$Typ \stackrel{def}{=} \boldsymbol{\lambda} t : TType \colon Typ\_BType\_list\ (sort\ (remove\_dup\ (get\_Btypes\ t))).$$

The range of the function *Typ* is our desired representation of the types of T. We can verify that the equations on the types are satisfied: $Typ\ (U\_ty\ T\ T) = Typ\ T$, and so on.

Next we must decide how to represent the terms of T. We shall simply represent `true` by *true*, `false` by *false*, `0` by $\mathbf{0}$ and `suc` by *suc*. We can check that these terms satisfy the first four typing rules: the type of *true* is *Typ bool_ty* and so on. However, the fifth typing rule presents a problem. Unions are represented as disjoint sums — and an object of some type S is not going to have the type $S + T$ for any other T.

To get around this we must define a function which computes the combinations of $in\mathbf{1}$s and $in\mathbf{2}$s required to coerce an object of type $Typ\ S$ into the union of S with any other $TType\ T$. We omit the definition of this function which we shall name *push*. Its type is

$$\boldsymbol{\Pi}\, S, T : TTtype.\ (Typ\ S) \rightarrow (Typ\ (U\_ty\ S\ T))$$

To embed the terms of T correctly, we shall have to use a pre-processor of some kind which will compute correct *TType* arguments to give to the function *push* in order to coerce a term into the desired type.

Properties of this embedding

The embedding provides an expressive language in which we can quantify over types and terms in the object language. We can express statements about the equality of specific object language types, but not about the rules by which equality or typing relationships are judged.

We shall not give a formal statement of adequacy for this embedding, but we believe that it is possible to do so and to verify that this embedding is adequate in a very strict sense.

This embedding is relatively easy to use compared to deep embeddings. However, there is a need for an external interface to compute the appropriate arguments to *push* when translating terms into the embedding, and to remove the occurrences of *push* when translating terms out of the embedding.

2.3 Conclusion

In chosing an embedding technique, there are tensions between the various desirable properties. Adequacy and expressivity are often traded off against ease of use. The use of a theorem-prover which implements a powerful logic, and which provides extensive automated support for that logic, can enable us to develop embedding techniques which offer good compromises among these properties.

Chapter 3

Encoding Z : Schemas

In this chapter we discuss our choice of representation technique for the Z notation. In accordance with the ideas discussed in the previous chapter, we are looking for an embedding style which will provide a reasonable compromise among adequacy, expressiveness, and ease of use. We shall consider three different encodings and discuss their advantages and drawbacks before we settle on one of them to use for the rest of this work.

For several reasons, we shall not consider doing a full internal description such as embedding 1 of Chapter 2. Such embeddings are difficult to set up and to use, and there exist simpler embedding techniques which give satisfactory results. In addition to these reasons, we do not consider the style of the definition of Z, as presented in [Spi88] to be amenable to the deep embedding technique. The definition of Z is itself presented as a Z specification: to represent this faithfully in UTT via a deep embedding we need to have already defined a translation from Z to UTT!

We shall use a reduced version of the Z notation which we call Z' . One of the main motivations behind the restrictions we impose is that we want to be able to distinguish between a core language and a module system. This point is discussed further in Section 3.1. In Section 3.2 we formally define the syntax and semantics of Z' . The remainder of the chapter deals with the technical aspects of our representation in UTT of the basic specification module in Z' , the schema.

3.1 Separate core and module languages

In the full Z language it is impossible to identify separate core and module languages. This is because any term in Z, and in particular, any schema, can be treated as a type. We shall not permit this in Z' . Instead we introduce a separate phrase class of types. This allows us to divide the syntax into separate core and module languages. Such a separation is desirable for a number of reasons. From the point of view of embedding Z in type theory, it allows us to embed the core language independently of the representation of the module language. As we shall see, this gives us greater flexibility in our choice of representation technique. This advantage was our main motivation for making this change to the Z notation. However, we believe that a separation of this kind is desirable in general in the definition of any large language, because it allows both the user and the metatheorist to understand the language in a piecewise fashion. Another advantage is that it allows us to consider the possibility of using the same module language with a different core language, or *vice versa*.

3.2 The syntax of Z'

Figures 3–1 and 3–2 show the abstract syntax of Z' . This syntax is a modified version of the syntax used in [Spi88]. Our syntax can be embedded into the latter: the presentation of the rules for doing this is deferred until Appendix E which discusses the relationship between our semantics for Z' and Spivey's semantics for Z.

A specification (SPEC) consists of a PRELUDE, which lists the given types over which that specification, followed by a main specification (MAINSPEC) which contains axiomatic descriptions and schema definitions. The separation of the PRELUDE is a departure from the syntax in [Spi88], where new given types may be introduced at any point within a specification.

A major difference between Z′ and the full Z notation is the reduced grammar for declarations. In the full Z notation, any TERM can appear at the right of the colon in a basic declaration. In Z′ , we introduce instead a special syntax class called TYPE, representing the allowed types. These may be given types, the type of finite sets over some other type, or the product of two types. (We restrict ourselves to *finite* sets in order to avoid difficulties in dealing with set-theoretic functions in type theory [Mah90].) We also add two primitive types $\mathbb{N}$ and $\mathbb{B}$

A related restriction involves the grammar of predicates (PRED). Whereas in Z the existential and universal quantifiers are allowed to quantify over SCHEMAs, in Z′ they quantify over an IDENT together with a TYPE.

The full Z notation also allows the use of schema designators (SDES) as declarations. This is not allowed in Z′ , but is partially recaptured by the addition of an `include` operation to the grammar of schema expressions. This allows schema designators to be used a declarations within schema expressions, but not within axiomatic descriptions.

Another big constraint is that our reduced syntax does not support schemas that are parameterised over generic formal parameters. However, it will be possible for whole specifications to be parameterised over generic given types listed in the PRELUDE.

The grammar of terms (TERM) is reduced to reflect the fact that terms and types have been separated. We also do not allow sets formed by comprehension, $\boldsymbol{\theta}$-terms, $\boldsymbol{\lambda}$-terms, or $\boldsymbol{\mu}$-terms.

Minor restrictions include the following: we do not treat the operation of projection on schemas; only primes can be used to decorate schema designators. There is no theoretical difficulty in removing these restrictions.

SPEC	::=	PRELUDE `in` MAINSPEC
PRELUDE	::=	`given` IDENT, . . . ,IDENT
MAINSPEC	::=	`let` SCHEMA `end`
	|	`let` WORD = SEXP
	|	MAINSPEC `in` MAINSPEC
SEXP	::=	`schema` SCHEMA `end`
	|	SDES
	|	$\neg$ SEXP
	|	SEXP $\wedge$ SEXP
	|	SEXP $\vee$ SEXP
	|	SEXP $\Rightarrow$ SEXP
	|	SEXP $\backslash$ (IDENT,. . .,IDENT)
	|	$\exists$ SCHEMA $\bullet$ SEXP
	|	$\forall$ SCHEMA $\bullet$ SEXP
	|	`include` SDES DECL PRED
SCHEMA	::=	DECL | PRED
DECL	::=	IDENT : TYPE
	|	DECL ; DECL

Figure 3–1: Syntax of Z' (part 1)

$$
\begin{array}{lcl}
\text{PRED} & ::= & \text{TERM} = \text{TERM} \\
 & | & \text{TERM} \in \text{TERM} \\
 & | & \texttt{true} \\
 & | & \texttt{false} \\
 & | & \neg\ \text{PRED} \\
 & | & \text{PRED} \wedge \text{PRED} \\
 & | & \text{PRED} \vee \text{PRED} \\
 & | & \text{PRED} \Rightarrow \text{PRED} \\
 & | & \exists\ \text{IDENT} : \text{TYPE} \bullet \text{PRED} \\
 & | & \forall\ \text{IDENT} : \text{TYPE} \bullet \text{PRED} \\
 & & \\
\text{TERM} & ::= & \text{IDENT} \\
 & | & \emptyset\ [\text{TYPE}] \\
 & | & \{\text{TERM},\ldots,\text{TERM}\} \\
 & | & (\text{TERM},\text{TERM}) \\
 & | & \text{TERM}\ (\text{TERM}) \\
 & & \\
\text{TYPE} & ::= & \text{IDENT} \\
 & | & \mathbb{N} \\
 & | & \mathbb{B} \\
 & | & \mathbb{F}\,\text{TYPE} \\
 & | & \text{TYPE} \times \text{TYPE} \\
 & & \\
\text{SDES} & ::= & \text{WORD PRIMES} \\
\text{IDENT} & ::= & \text{WORD DECOR} \\
 & & \\
\text{WORD} & - & \text{undecorated identifiers (alphanumeric strings)} \\
\text{DECOR} & - & \text{decorations (strings over } \{{}',!,?\}) \\
\text{PRIMES} & - & \text{primes (strings over } \{{}'\})
\end{array}
$$

Figure 3–2: Syntax of Z (part 2)

3.3 The semantics of Z′

Here we present a short summary of the official semantics of Z proposed in [Spi88]. This discussion is necessarily incomplete, and is intended simply to give a flavour of the semantics of Z, and to introduce the names, though not the formal definitions, of the semantic functions that will be referred to in Appendix E.

For our reduced language Z′ , the corresponding semantics will be obtained by first applying the embedding into Z defined in figures E–1 and E–2 of Appendix E.

The denotation of a Z specification in Spivey's semantics is an *environment*. This consists of two parts. The first is a *variety*, called the *global variety*, which represents all the given types and axiomatic descriptions of the specification. The second, the *schema dictionary*, is a mapping from names to varieties, and represents all the schema definitions in the specification.

A variety consists of a *signature*, together with a set of *models* over that signature. A signature contains names of given types and variables, and associates types with all the variables.

Several operations are then defined on these semantic objects. The *join* operation, for example, combines two signatures to form a new signature containing the union of the variables in the two argument signatures. This function is defined only for signatures in which the common variables have the same types. This operation is used to obtain the signature of the schema denoted by any of the various binary schema expressions.

Another important operation, *enrich* takes as arguments a variety V and an environment E whose global component must be a sub-variety of V (i.e. , the signature of V must contain all variables in the signature of $E.global$, and all models in V must also be models in $E.global$.) This operation then returns the environment formed by replacing the global component of E with V. Intuitively, the environment E has been enriched by adding the variables and the constraints of the variety V.

An example of the use of *enrich* is the semantics of a schema body (SCHEMA). This is evaluated within an environment and denotes a variety. To obtain this variety, the environment is first enriched with a variety obtained by evaluating the declarations of the SCHEMA. Another semantic function called *pred* is then used to evaluate the predicate of the SCHEMA within this new, enriched environment. The denotation of the predicate is a set of models. The denotation of the SCHEMA is then the variety whose signature is that of the global variety of the enriched environment, and whose models component is the denotation of the predicate.

Several operations are defined on varieties. For example, the operations *combine* is used in giving the meaning of schema expressions formed by conjunction. The *combine* operation takes a pair of varieties as arguments and returns a new variety. The signature of the result is formed by applying *join* to the signatures of the argument. The models of the result are all models over the new signature which, when suitably restricted, yield models of both of the argument varieties.

3.4 Choosing a Representation Technique

In this section we consider various possibilities for encoding the syntax and semantics of Z' in type theory. We must find suitable representations for all the components of the definition of Z' — the syntactic classes, the semantic objects, and the relationships between them — bearing in mind that all these parts must be capable of being put together to provide a representation of the whole. This will impose a requirement of expressiveness on the representation chosen for some components. We would also like to keep all the representations as simple as possible, and to be able to identify measures of the adequacy of each representation.

Since our goal is simplicity, we shall begin by considering the simplest possible embedding style: the shallow embedding. This turns out to be unsatisfactory, so we shall move to a slightly deeper embedding in the style of Section 2.2.5.

We shall look at how the following small Z′ specification can be represented. This example is written in the concrete syntax of Z, and makes use of some definitions from the Z library.

S

$x, y : \mathbb{N}$

$x < y$

T

$x, z : T$

$x + z = \mathbf{0}$

$U \mathrel{\widehat{=}} S \wedge T$

According to the definition of schema conjunction, the schema U denotes the same variety as the following schema:

U

$x, y, z : \mathbb{N}$

$(x < y) \wedge (x + z = \mathbf{0})$

3.4.1 Embedding 1: a shallow embedding

The obvious method of embedding Z' shallowly in UTT, within a theorem-prover such as LEGO, is as follows: The global part of a Z' specification is represented by assumptions added to the LEGO context. For each given set a type is assumed. For each variable within an axiomatic description, a LEGO variable of the appropriate type is assumed. For each axiom within an axiomatic description, a corresponding axiom is assumed in the LEGO context. Next, we must decide how to represent the schema definitions. This can be represented by a collection of definitions in LEGO: for each name defined in the Z′ specification, a corresponding identifier is defined in LEGO. The value associated with that identifier will be a UTT term which somehow represents the meaning of the associated schema expression. This leaves us with the question of deciding what UTT terms should be used to represent these meanings.

The obvious candidate for this is the Σ-types of UTT. This approach is explored in [Mah90]. We can think of these Σ-types as representing the varieties of Spivey's semantics. The signature of the variety is represented by a UTT type, formed by taking the product of all the types in the signature. The models component of the variety is then represented by a UTT predicate over this type. The two are combined to form a Σ type. As an example, here is the representation of part of the example specification:

$$
\begin{aligned}
&T : \mathit{Type}\\
&S\_sig \stackrel{\mathit{def}}{=} T \times T\\
&S\_pred \stackrel{\mathit{def}}{=} \boldsymbol{\lambda} b : S\_sig.\, b.\mathbf{1} = b.\mathbf{2}\\
&S \stackrel{\mathit{def}}{=} \boldsymbol{\Sigma}\, b : S\_sig.\, S\_pred\ b
\end{aligned}
$$

This encoding of varieties is reminiscent of other representations of modular specifications in LEGO, such as the work of Luo [Luo91] and the "deliverables" approach of Burstall and McKinna [BM91,McK92].

The shortcoming of this representation is that it provides a poor degree of expressiveness. There is no way for us to capture, within UTT, the operations on varieties which are used to give meaning to compound schema expressions. The reason is that our embedding does not provide us with any object in the type theory that represents the class of *all* varieties. The nearest thing to such an object is the universe *Type*, since all varieties are represented as types, but this universe also contains many types that are not representations of varieties.

In order to define an operation on varieties we would need to extend that operation somehow so that it applied to all types. There is no obvious way of doing this meaningfully. What is more, even if such an extension could be found, we would still be unable to define the *combine* operation because we have no representation of the names of the identifiers in a signature, only of their types.

Our challenge is to find a UTT type which is a more accurate representation of varieties. This leads us to consider a slightly deeper embedding.

3.4.2 Embedding 2: Syntactic names and types in signatures

The problem with the previous embedding arose because of the way that signatures were represented. Firstly, we kept no information about the names in a signature. Secondly, the UTT type used to represent signatures (*viz. Type*) was too general. Our second encoding addresses both of these problems.

We introduce a type to represent Z identifiers. For convenience, we shall use the type of natural numbers:

Definition 1 Ident,Ident_eq.

$$Ident \stackrel{def}{=} nat : Type$$
$$Ident\_eq \stackrel{def}{=} nat\_eq : Ident \rightarrow Ident \rightarrow bool$$

Similarly, we introduce a type to represent the *names* of the types in Z' . We shall assume that we have already defined a types *GivenType* representing the given types used in some particular specification. We define an inductive type called *Ztype*.

Definition 2 Ztype.

$$\begin{array}{l} \mathit{Inductive[\ Ztype : Type\ ]} \\ \mathit{Constructors[\ nat\_ty, bool\_ty : Ztype} \\ \qquad\qquad \mathit{given\_ty : GivenType \rightarrow Ztype} \\ \qquad\qquad \mathit{finset\_ty : Ztype \rightarrow Ztype} \\ \qquad\qquad \mathit{prod\_ty : Ztype \rightarrow Ztype \rightarrow Ztype\ ]} \end{array}$$

This embedding represents the class TYPE exactly. The elimination rule, *Ztype_elim*, produced by the inductive definition gives us a lot of power to define operations and prove theorems about TYPE. If we are able to define a decidable equality on *GivenType*, or if we assume that such an equality exists, then the elimination rule allows us to extend this to a decidable equality on *Ztype*:

Definition 3 Ztype_eq.

$$\mathit{Ztype\_eq} \stackrel{def}{=} \mathit{[omitted]} : \mathit{Ztype} \rightarrow \mathit{Ztype} \rightarrow \mathit{bool}$$

We could, if we chose, develop this into a deep embedding of Z′ but we prefer not to take this route. Deep embeddings, at least in the LEGO system, are difficult and painful to use. As we have seen in the previous chapter, type theory provides us with another possibility:

We adopt the approach used in for representing union types in Section 2.2.5. We define an operation *Typ* which maps *Ztype*s to *Type*s. *Typ* can be thought of as mapping each TYPE to the semantic object that it denotes. (We shall assume that we are able to define a similar operation *Typ_gtype* on *GivenType*.)

Definition 4 Typ.

$$\begin{array}{l} \mathit{Typ} \stackrel{def}{=} \mathit{Ztype\_rec}\ (\boldsymbol{\lambda} g : \mathit{GivenType}.\ \mathit{Typ\_gtype}\ g) \\ \qquad\qquad (\boldsymbol{\lambda} z : \mathit{Ztype}.\ \boldsymbol{\lambda} \mathit{Typ\_z} : \mathit{Type}.\ \mathit{fin_set}\ \mathit{Typ\_z}) \\ \qquad\qquad (\boldsymbol{\lambda} z, z' : \mathit{Ztype}.\ \boldsymbol{\lambda} \mathit{Typ\_z}, \mathit{Typ\_z'} : \mathit{Type}.\ \mathit{Typ\_z} \times \mathit{Typ\_z'}) \\ : \mathit{Ztype} \rightarrow \mathit{Type} \end{array}$$

TYPEs are represented by the type *Ztype* and their denotations by the types in the range of *Typ*. (The type $Typ\ (fin\_set\ z)$ is defined as $list\ z$: the details of the representation of finite sets are described in Chapter 4.)

We define signatures as lists of *signature items*, which are pairs consisting of an *Ident* and a *Ztype*.

Definition 5 Signature.

$$\begin{aligned}
&sig_item \stackrel{def}{=} (Ident \times Ztype) : Type \\
&Signature \stackrel{def}{=} list\ sig_item : Type \\
&nil_sig \stackrel{def}{=} nil \mid sig_item : Signature
\end{aligned}$$

We can define decidable equalities on both of these types:

$$\begin{aligned}
&sig_item_eq : sig_item \rightarrow sig_item \rightarrow bool \\
&Sig_eq : Signature \rightarrow Signature \rightarrow bool
\end{aligned}$$

Next we extend *Typ* to a function *Typify* of type $Signature \rightarrow Type$ which forms a product of the types obtained by applying *Typ* to all of the *Ztype*s in a given signature, together with the unit type in the case of the empty signature. This function can be defined easily using induction on lists. *Typify sig* can be seen as representing the set of models over a signatures *sig*. Imitating the terminology used in the ZRM, we shall call such objects *bindings*.

We can use this new definition of signatures to define a much better representation of the class of Schemas than that provided by the previous embedding. A schema is defined as a *Signature sig* paired with a predicate over bindings of type *Typify sig*:

$$\begin{aligned}
&Schema \stackrel{def}{=} \Sigma\, sig : Signature.\ (Typify\ sig) \rightarrow Prop \\
&: Type
\end{aligned}$$

To help us work with these syntactic signatures we shall use a function called *lookup* which has the following somewhat complicated definition:

$$
\begin{array}{l}
lookup\_aux \stackrel{def}{=} \boldsymbol{\lambda} i : Ident. \\
[result\_type = \boldsymbol{\lambda} s : Signature.\ \Sigma\, t : Type.\ (Typify\ s) \rightarrow t] \\
\quad list\_elim\ result\_type \\
\qquad\quad (Error, \boldsymbol{\lambda} x : Error.\ x) \\
\qquad\quad (\boldsymbol{\lambda} hd : Ident \times Ztype.\ \boldsymbol{\lambda} tl : Signature.\ \boldsymbol{\lambda} prev : \Sigma\, t : (Typify\ tl) \rightarrow t.\ . \\
\qquad\qquad if\ (Ident\_eq\ i\ hd.\mathbf{1}) \\
\qquad\qquad\quad (Typ\ hd.\mathbf{2}, \boldsymbol{\lambda} x : Typify\ (cons\ hd\ tl).\ x.\mathbf{1}) \\
\qquad\qquad\quad (prev.\mathbf{1}, \boldsymbol{\lambda} x : Typify\ (cons\ hd\ tl).\ prev.\mathbf{2}\ x.\mathbf{2})) \\
: Ident \rightarrow \boldsymbol{\Pi}\, sig : Signature.\ (Typify\ sig) \rightarrow \Sigma\, t : Type.\ t
\end{array}
$$

$$
lookup \stackrel{def}{=} \boldsymbol{\lambda} i : Ident.\ \boldsymbol{\lambda} sig : Signature.\ (lookup\_aux\ i\ sig).\mathbf{2}
$$

When given an identifier i and a signature *sig*, *lookup_aux* attempts to locate i in *sig*. If i is found then *lookup_aux* returns a dependent pair consisting of the type t with which i is found to be associated and a function of type $(Typify\ sig) \rightarrow t$. When applied to a tuple of type *Typify sig*, this function projects the component which corresponds to the position of i in *sig*. If i does not occur in *sig* then *lookup_aux* returns the pair $(Error, \boldsymbol{\lambda} x : Error.\ x)$. The function *lookup* simply extracts the second component of the result returned by *lookup_aux*.

To understand the working of *lookup* let us look at the following example. We first define some identifiers and syntactic signatures.

$$
\begin{array}{l}
x \stackrel{def}{=} \mathbf{0} \\
y \stackrel{def}{=} \mathbf{1} \\
Sig_1 \stackrel{def}{=} [(x, nat\_ty), (y, nat\_ty)] \\
Sig_2 \stackrel{def}{=} [(y, nat\_ty), (x, bool\_ty)] \\
Sig_3 \stackrel{def}{=} [(y, nat\_ty)]
\end{array}
$$

Next we define some bindings typed by the signatures above:

$$Bin_1 \stackrel{def}{=} (\mathbf{2}, \mathbf{3}, error) : Typify\ Sig_1$$
$$Bin_2 \stackrel{def}{=} (\mathbf{2}, true, error) : Typify\ Sig_2$$
$$Bin_3 \stackrel{def}{=} (\mathbf{2}, error) : Typify\ Sig_3$$

The following table shows some of the results of applying *lookup* to these signatures and bindings:

Term	Type	Value
$(lookup\ x\ Sig_1)\ Bin_1$	*nat*	$\mathbf{2}$
$(lookup\ x\ Sig_1)\ Bin_2$	fails to typecheck	
$(lookup\ x\ Sig_2)\ Bin_2$	*bool*	*true*
$(lookup\ x\ Sig_3)\ Bin_3$	*Error*	*error*

The main use of *lookup* is in writing schema predicates. This is illustrated by the new definition of the schema *S*.

$$x \stackrel{def}{=} \mathbf{0}$$
$$y \stackrel{def}{=} \mathbf{1}$$
$$S\_sig \stackrel{def}{=} [(x, nat\_ty), (y, nat\_ty)]$$
$$S\_pred \stackrel{def}{=} \boldsymbol{\lambda} bin : Typify\ S\_sig.\ ((lookup\ x\ S\_sig)\ bin) < ((lookup\ y\ S\_sig)\ bin)$$
$$S \stackrel{def}{=} (S\_sig, S\_pred)$$

Typechecking will detect when *lookup* fails since the value returned in the case of failure will have the type *Error*, rather than the type required by the rest of the predicate in which the *lookup* occurs. For example, in the above definition of the schema *S*, if we remove the identifier *x* from *S_sig* (or if we paired it with some other *Ztype*), then the definition of *S_pred* will fail to typecheck.

Encoding operations on schemas

This representation of schemas provides us with enough expressiveness to define the operations on schemas. Let us look at the operation of conjoining two schemas S and T. Suppose S and T are represented by *Schema*s *S* and

T, respectively. First we compute the new signature *newsig* by joining the signatures of S and T. Having computed the new signature, we must somehow conjoin the predicates of S and T, whose domains are *Typify* $S.\mathbf{1}$ and *Typify* $T.\mathbf{1}$, respectively, to form a new predicate over the domain *Typify newsig*. To do this we must be able to extract from objects of type *Typify newsig* the components that make up objects of type *Typify* S and *Typify* T.

Whenever two signatures are joined, as they are for many of the schema operations, it will be necessary to perform extractions such as that described above. So, whenever we join two signatures we shall also compute coercion functions which extract bindings of the type of the two argument signatures from bindings of the type of the result signature.

$$
\begin{array}{l}
\mathit{join} \stackrel{\mathit{def}}{=} \boldsymbol{\lambda} s : \mathit{Signature}.\\
\quad \mathit{list\_rec}\ (s, \boldsymbol{\lambda} b : \mathit{Typify}\ s.\ b)\\
\qquad (\boldsymbol{\lambda} h : \mathit{sig\_item}.\ \boldsymbol{\lambda} s' : \mathit{Signature}.\ \boldsymbol{\lambda} \mathit{prev} : \Sigma\, v : \mathit{Signature}.\ (\mathit{Typify}\ v) \longrightarrow (\mathit{Typify}\ s).\\
\qquad\quad \mathit{if}\ (\mathit{member}\ \mathit{sig\_item\_eq}\ h\ s)\\
\qquad\qquad \mathit{prev}\\
\qquad\qquad (\mathit{cons}\ h\ \mathit{prev}.\mathbf{1}, \boldsymbol{\lambda} x : \mathit{Typify}\ (\mathit{cons}\ h\ \mathit{prev}.\mathbf{1}).\ \mathit{prev}.\mathbf{2}\ x.\mathbf{2}))\\
: \boldsymbol{\Pi}\, s, \_ : \mathit{Signature}.\ \Sigma\, \mathit{newsig} : \mathit{Signature}.\ (\mathit{Typify}\ \mathit{newsig}) \longrightarrow (\mathit{Typify}\ s)
\end{array}
$$

The *join* operation, as we define it above, takes two signatures s_1 and s_2 and yields a new signature *newsig* together with a coercion back from *newsig* to s_1. The coercion takes a tuple of type *Typify newsig* and produces a tuple of type *Typify* s_2 by projecting only those components that correspond to identifiers present in s_1. No coercion back to the second signature s_2 is computed since this may not exist if s_1 and s_2 happen to be incompatible for joining. (For instance, if s_1 is $[(x, \mathit{bool\_ty})]$, and s_2 is $[(x, \mathit{nat\_ty})]$, they are incompatible for joining. The *join* operation, as defined, will use the *Ztype* obtained from s_1 rather than s_2 in its result, so it returns the signature $[(x, \mathit{bool\_ty})]$.)

Another operation *coerce* attempts to find coercions between arbitrary signatures. The type of *coerce* reflects the fact that it is partial: if no coercion exists it returns *in*$\mathbf{2}$ *error*. The definition of *coerce* is fairly long, though not conceptually difficult, so we shall not include it here.

$$coerce : \mathbf{\Pi}\, S, S' : Signature.\ ((Typify\ S) \rightarrow (Typify\ S')) + Error$$

Next we define a trivial, unsatisfiable schema, *Absurd*:

$$Absurd \stackrel{def}{=} (nil, \boldsymbol{\lambda}\_ : Error.\ absurd)$$

Now we can represent schema conjunction. To conjoin S and S' we first *join* their signatures to form a new signature *newsig*. Then we attempt to coerce *newsig* back to the signature of S'. This will fail if S and S' happen to be type-incompatible, in which case we return *Absurd* as our result. Otherwise we return a schema made up of *newsig* and the predicates of S and S' conjoined and composed with coercions as appropriate. The definition is as follows:

$$
\begin{array}{l}
And \stackrel{def}{=} \boldsymbol{\lambda} S, S' : Schema. \\
\quad [tmp \stackrel{def}{=} join\ S\ S'] \\
\quad [newsig \stackrel{def}{=} tmp.\mathbf{1}] \\
\quad [coercion\mathbf{1} \stackrel{def}{=} tmp.\mathbf{2}] \\
\quad [coercion\mathbf{2} \stackrel{def}{=} coerce\ newsig\ S'.sig] \\
case \\
\quad (\boldsymbol{\lambda} f : (Typify\ newsig) \rightarrow (Typify\ S'.sig). \\
\qquad (newsig, \boldsymbol{\lambda} s : Typify\ newsig.\ (S.pred\ (coercion\mathbf{1}\ s)) \wedge (S'.pred\ (f\ s)))) \\
\quad (\boldsymbol{\lambda} x : Error.\ Absurd) \\
\quad coercion\mathbf{2} \\
: Schema \rightarrow Schema \rightarrow Schema
\end{array}
$$

With this method it will be possible to define all the operations on schemas. However, we have to deal with coercions. Whenever a schema operation makes a change to a signature, or somehow combines two or more signatures, it may be necessary to compute coercions between the new signature and the old one(s). In practice, we found that reasoning about these coercions turns out to be awkward. We therefore chose to explore a third method of representing schemas, in which we have managed to avoid the need to talk about coercions between signatures.

3.4.3 Embedding 3: Syntactic names in bindings

In the previous embedding, the Z typing relationship between bindings and Signatures was identified with the typing relationship between terms and types in UTT. Each distinct Signature gave rise to a distinct type representing the bindings over that signature. Whenever an operation caused a signature to be changed in any way, it was necessary to compute coercions relating the new type of bindings associated with the new signature to the type of bindings associated with the old signature.

We shall now look at an embedding technique in which we do not make the identification described above, but instead explicitly define the typing relationship between bindings and signatures. This allows us to have a single, uniform type to represent all bindings, rather than a different type for each Signature. Schema Predicates, as a result, will also be represented by a single type, instead of a different type for each Signature. As we shall see, this choice makes it much easier to define operations on schemas. Because of the ease of use, this will be the embedding technique that we shall adopt for the rest of this work.

We shall define a general type to represent bindings. Bindings are lists of *bin_item*s, which are themselves triples in which a syntactic identifier and a *Ztype* z are paired with a value of type *Typ* z:

Definition 6 Binding.

$$\begin{aligned}
&\mathit{bin_item} \stackrel{def}{=} \Sigma\, p\!:\!\mathit{Ident}\times \mathit{Ztype}.\, \mathit{Typ}\ p.\mathbf{2}\ :\ \mathit{Type}\\
&\mathit{Binding} \stackrel{def}{=} \mathit{list}\ \mathit{bin_item}\ :\ \mathit{Type}\\
&\mathit{nil_bin} \stackrel{def}{=} \mathit{nil}\mid \mathit{bin_item}\ :\ \mathit{Binding}
\end{aligned}$$

We shall write some functions to handle these syntactic bindings. First we define a function which simply extracts a *Signature* from a *Binding*:

Definition 7 extract_sig.

$$\begin{aligned}
&\mathit{extract_sig} \stackrel{def}{=} \mathit{map}\ \lambda x\!:\!\mathit{bin_item}.\, x.\mathbf{1}\\
&:\ \mathit{Binding}\rightarrow \mathit{Signature}
\end{aligned}$$

The next function, *matches*, checks whether a given *Binding* is typed by a given *Signature*:

Definition 8 matches.

$$\begin{aligned}
&matches \stackrel{def}{=} \boldsymbol{\lambda} S : Signature.\ \boldsymbol{\lambda} bin : Binding.\ is_true\ (Sig_eq\ S\ (extract_sig\ bin))\\
&: Signature \longrightarrow Binding \longrightarrow bool
\end{aligned}$$

We redefine *lookup* to work with our new representation of bindings.

Definition 9 lookup.

$$\begin{aligned}
&small_item \stackrel{def}{=} \Sigma\, z : Ztype.\ (Typ\ z)\ :\ Type\\
&lookup \stackrel{def}{=} \boldsymbol{\lambda} x : sig_item.\\
&\qquad list_iter\ (in\mathbf{1}\ void)\\
&\qquad\qquad (\boldsymbol{\lambda} y : bin_item.\ \boldsymbol{\lambda} prev : (Error + small_item).\\
&\qquad\qquad\quad if\ (sig_item_eq\ x\ y.\mathbf{1})\ (in\mathbf{2}\ (y.\mathbf{1}.\mathbf{2}, y.\mathbf{2}))\ prev)\\
&: sig_item \longrightarrow Binding \longrightarrow (Error + small_item)
\end{aligned}$$

The next function *restrict*, attempts to cut down a *Binding* so that it has only those components specified in a given *Signature*; *restrict* fails if the *Signature* requires components that are not in the *Binding*.

Definition 10 restrict.

$$\begin{aligned}
&restrict \stackrel{def}{=} \boldsymbol{\lambda} str : Binding.\\
&\quad list_iter\ (in\mathbf{2}\ nil_bin)\\
&\qquad (\boldsymbol{\lambda} x : sig_item.\ \boldsymbol{\lambda} prev : (Error + Binding).\\
&\qquad case\ \boldsymbol{\lambda}_ : Error.\ in\mathbf{1}\ void\\
&\qquad\quad \boldsymbol{\lambda} s : Binding.\ case\ \boldsymbol{\lambda}_ : Error.\ in\mathbf{1}\ void\\
&\qquad\qquad\qquad \boldsymbol{\lambda} si : small_item.\ in\mathbf{2}\ (cons\ ((n.\mathbf{1}, si.\mathbf{1}), si.\mathbf{2})\ s)\\
&\qquad\qquad\qquad (lookup\ x\ bin)\\
&\qquad\quad prev)\\
&: Binding \longrightarrow Signature \longrightarrow (Error + Binding)
\end{aligned}$$

An example of a *Binding* is the following:

$$x \stackrel{def}{=} \mathbf{0} : Ident$$
$$b \stackrel{def}{=} \mathbf{1} : Ident$$
$$B \stackrel{def}{=} [(x, nat\_ty, 0), (b, bool\_ty, true)] : Binding$$

If we define *sig* to be the signature $[(b, bool\_ty)]$ then $(matches\ sig\ B)$ evaluates to *false* and $(restrict\ B\ sig)$ evaluates to the *Binding* $[(b, bool\_ty, true)]$.

Schema predicates will be encoded by the following type:

Definition 11 Predicate.

$$Predicate \stackrel{def}{=} Binding \longrightarrow Prop$$

Our new definition of the type *Schema* is:

Definition 12 Schema.

$$Schema \stackrel{def}{=} Signature \times Predicate : Type$$

The *join* operation for combining Signatures can be defined much more simply than in embedding 2 because there is no need to keep track of coercions between types:

Definition 13 join.

$$\begin{array}{l}
join \stackrel{def}{=} \lambda s, t : Signature. \\
\qquad list_iter \\
\qquad\quad t \\
\qquad\quad (\lambda x : sig_item.\ \lambda prev : Signature. \\
\qquad\qquad\quad if\ \ (member\ sig_item_eq\ x\ prev)\ prev\ \ (cons\ x\ prev)) \\
\qquad\quad s \\
: Signature \longrightarrow Signature \longrightarrow Signature
\end{array}$$

Schema conjunction can then be represented as follows:

$$\begin{array}{l}
And \stackrel{def}{=} \lambda S, S' : Schema. \\
\qquad (join\ S.sig\ S'.sig,\ \ \lambda s : Binding.\ (S.pred\ s)\ \wedge\ (S'.pred\ s)) \\
: Schema \longrightarrow Schema \longrightarrow Schema
\end{array}$$

3.5 Working out the details

In this section we discuss some of the technical issues that must be dealt with in working out the details of our embedding technique.

3.5.1 Relating schemas and bindings

Embedding 2 provided only one possible relationship between bindings and schemas. If S is of type *Schema*, as defined in embedding 2, then an object of type S can be interpreted as a binding over the signature of S together with a proof that the binding satisfies the predicate of S. With embedding 3, it is possible to define several different relationships between schemas and bindings. It is necessary to choose which of these best captures the Z notion of model. We consider some of the possibilities and the relationships between them.

- **Exact models**

  We say that a *Binding* is an exact model of a *Schema* if it matches the *Signature* of that *Schema* and if we can prove that it satisfies the *Predicate* of that *Schema*. This is essentially the same as the notion of model in embedding 2, rephrased in the language of embedding 3. This definition is closest to the satisfaction relationship between schemas and bindings defined in Section 3.3.

  Definition 14 exactly_models.

  $$\begin{aligned}&\mathit{exactly\_models} \stackrel{def}{=} \boldsymbol{\lambda} S : \mathit{Schema}\boldsymbol{.}\ \boldsymbol{\lambda} b : \mathit{Binding}\boldsymbol{.}\\ &\qquad (\mathit{is\_true}\ (\mathit{matches}\ S\boldsymbol{.}\mathbf{1}\ b)) \wedge (S\boldsymbol{.}\mathbf{2}\ b)\\ &: \mathit{Schema} \longrightarrow \mathit{Binding} \longrightarrow \mathit{Prop}\end{aligned}$$

- **Restricting models**

  Exact models seem to capture the Z notion of model, so why should we consider any other definitions? The reason is similar to the reason why we needed coercion functions in method 2 of encoding schemas. We are often interested in the relationship between a single binding and various different schemas with differering signatures. In embedding 2 we took care of this need by finding coercions between the various signatures. For embedding 3, we shall invent the notion of "restricting models" of schemas. A binding is a restricting model of a schema S if it is capable of being restricted to the signature of S, and the restricted binding satisfies the predicate of S.

Another way of looking at this definition is to think of a schema as a minimal description: bindings must contain at least those identifiers present in the schema signature. When restricted to this signature, the binding must satisfy the schema predicate.

Definition 15 restricts_to_model.

$$\begin{aligned}
&\mathit{restricts_to_model} \stackrel{def}{=} \boldsymbol{\lambda} S : \mathit{Schema}.\ \boldsymbol{\lambda} b : \mathit{Binding}. \\
&\qquad \exists\, b' : \mathit{Binding}.\ (\mathit{in}\mathbf{2}\ b' = \mathit{restrict}\ b\ S.\mathbf{1}) \wedge (S.\mathbf{2}\ b') \\
&: \mathit{Schema} \rightarrow \mathit{Binding} \rightarrow \mathit{Prop}
\end{aligned}$$

- **Signature-independent models**

  Finally, we consider bindings which satisfy the predicate of a schema, independently of their relationship with the signature of that schema. We shall call such bindings "models" of a schema.

Definition 16 models.

$$\begin{aligned}
&\mathit{models} \stackrel{def}{=} \boldsymbol{\lambda} S : \mathit{Schema}.\ S.\mathbf{2} \\
&: \mathit{Schema} \rightarrow \mathit{Binding} \rightarrow \mathit{Prop}
\end{aligned}$$

Now we shall look at the relationships among these three notions of model for schemas.

Exact models and models

All exact models of a *Schema S* are also models of *S*:

Theorem 1 exact_models_are_models.

$$\forall\, S\!:\!Schema.\ \forall\, b\!:\!Binding.\ (exactly\_models\ S\ b) \Rightarrow models\ S\ b$$

Proof. Trivial.

It is easily shown that the converse of the above statement is not true.

Exact models and restricting models

Bindings which are exact models of a schema are always capable of being restricted to the signature of that schema. (This is implied by lemma 16 of Appendix A). However, it is not the case that the binding produced by the restriction must necessarily satisfy the predicate of the schema. The restricted binding is not necessarily equal to the original binding: if a signature contains more than one occurrence of the same identifier, a binding that matches this signature will not necessarily contain the same values as its restricted form, since the restriction function grabs the first occurence of each identifier. If we restrict ourselves to schemas whose signatures do not contain repeated identifiers, then we can show that the restricted binding does indeed satisfy the schema predicate.

First we shall define a predicate *unique_idents* on *Signature*s. As the name implies, this is true of a signature if no identifier occurs more than once in that signature. The definition makes use of a decidable equality $sig\_item\_ident\_eq$: $sig\_item \rightarrow sig\_item \rightarrow bool$ which simply judges two *sig_item*s to be equal if their *Ident* components are equal under *Ident_eq*.

Definition 17 unique_idents.

$$
\begin{array}{l}
\mathit{unique\_idents} \stackrel{def}{=} \\
\quad \mathit{list\_rec}\ \mathit{trueProp} \\
\qquad\qquad (\lambda x : \mathit{sig\_item}.\ \lambda l : \mathit{Signature}.\ \lambda \mathit{prev} : \mathit{Prop}. \\
\qquad\qquad\qquad\qquad (\mathit{is\_false}\ (\mathit{member}\ \mathit{sig\_item\_ident\_eq}\ x\ l)) \wedge \mathit{prev}) \\
: \mathit{Signature} \longrightarrow \mathit{Prop}
\end{array}
$$

Theorem 2 exact_models_are_restricting_models.

$$
\begin{array}{l}
\forall S : \mathit{Schema}.\ \forall b : \mathit{Binding}.\ (\mathit{unique\_idents}\ S.\mathbf{1}) \Rightarrow \\
\qquad (\mathit{exactly\_models}\ S\ b) \Rightarrow \mathit{restricts\_to\_model}\ S\ b
\end{array}
$$

Proof. This follows easily from the Lemma 15 which says that a binding is equal to the result of restricting it to its own signature, provided it has no duplicate identifiers. □

It is easy to show that the converse of the above result is false: not all restricting models are exact models. However, all restricting models yield exact models when restricted:

Theorem 3 restricting_models_give_exact_models.

$$
\begin{array}{l}
\forall S : \mathit{Schema}.\ \forall b : \mathit{Binding}.\ (\mathit{restricts\_to\_model}\ S\ b) \Rightarrow \\
\qquad \exists b' : \mathit{Binding}.\ (\mathit{in}\mathbf{2}\ b' = \mathit{restrict}\ b\ S.\mathbf{1}) \wedge (\mathit{exactly\_models}\ S\ b')
\end{array}
$$

Proof. This is an easy consequence of lemma 17 which states that any binding obtained by restriction to a signature will match that signature. □

Models and restricting models

The most interesting relationship to consider is that between models and restricting models. First of all, is it the case that a model of a schema S is necessarily a restricting model of S? The following object of type *schema* gives us a negative answer to this question. (The operation *lookup* used in this definition is similar in function to the *lookup* that was defined for embedding 2. Its detailed definition will be given later in this chapter.)

$$S \stackrel{def}{=} (nil\_sig, \lambda b{:}Binding.\ lookup\ x\ b = (nat\_ty, \mathbf{0}))$$
$$: Schema$$

This *Schema* arises as the translation of the following ill-formed Z schema:

S

$x = \mathbf{0}$

There are many *Binding*s which are models of this *Schema*, but there are none which are restricting models. In general, any *Schema* whose predicate makes reference to identifiers not found in its signature will fail to have restricting models.

What about the converse situation? If a *Binding* is a restricting model of a *Schema*, is it itself a model? Again the answer is no. Consider the following term of type *Schema*:

$$T \stackrel{def}{=} (nil\_sig, \lambda b{:}Binding.\ length\ b = \mathbf{0})$$
$$: Schema$$

All bindings are restricting models of this *Schema*, but only the empty binding is a model. However, unlike the schema *S* above, this LEGO term does not arise from the translation of any Z schema, even one that is ill-formed.

Finally, let us look at one more *Schema*:

$$U \stackrel{def}{=} ([(x, nat\_ty), (y, nat\_ty)], \lambda b{:}Binding.\ lookup\ x\ b = (nat\_ty, \mathbf{0}))$$
$$: Schema$$

This *Schema* is the translation of the following Z schema:

U

$x, y : \mathbb{N}$

$x = \mathbf{0}$

Any *Binding* which is a restricting model of U will also be a model of U. However, not all models of U are restricting models of U: the binding $[(x, nat\_ty, \mathbf{0})]$ is a model of U but cannot be restricted successfully to the *Signature* of U, and therefore cannot be a restricting model. However, all models of U that can be successfully restricted to the *Signature* of U are indeed restricting models. This leads us to our definition of well-formedness for schemas.

3.5.2 Well-formedness conditions

Many schemas exist which do satisfy the properties that restricting models are models, and sufficiently large models are restricting models. Later on we shall see that these properties are preconditions for some metatheorems about the schema operations. We formalise these properties in LEGO by means of the following definitions:

Definition 18 Down_closed.

$$\begin{aligned} &Down\_closed \stackrel{def}{=} \boldsymbol{\lambda} S\!:\!Schema.\ \forall\, b\!:\!Binding. \\ &\qquad (models\ S\ b) \Rightarrow (succeeds\ (restrict\ b\ S.sig)) \Rightarrow (restricts\_to\_model\ S\ b) \end{aligned}$$

Definition 19 Up_closed.

$$\begin{aligned} &Up\_closed \stackrel{def}{=} \boldsymbol{\lambda} S\!:\!Schema.\ \forall\, b\!:\!Binding. \\ &\qquad restricts\_to\_model\ S\ b \Rightarrow models\ S\ b \\ &: Schema \rightarrow Prop \end{aligned}$$

These two properties are independent of each other. The schema S is not down-closed, as we have seen, but it is up-closed. The schema T is down-closed but is not up-closed. We have verified these facts using LEGO.

Finally we shall define a well-formedness condition on schemas:

Definition* 20 *Well_formed.

$$Well\_formed \stackrel{def}{=} \boldsymbol{\lambda} S : Schema.$$
$$(Up\_closed\ S) \wedge (Down\_closed\ S) \wedge (unique\_idents\ S.\mathbf{1})$$
$$: Schema \rightarrow Prop$$

3.5.3 Properties of schemas

We define what it means for a schema to have a property:

Definition* 21 *Has_prop.

$$Has\_prop \stackrel{def}{=} \boldsymbol{\lambda} S : Schema.\ \boldsymbol{\lambda} P : Predicate.$$
$$\forall\, b : Binding.\ (restricts\_to\_model\ b\ S) \Rightarrow (P\ b)$$
$$: Schema \rightarrow Predicate \rightarrow PropSchema \rightarrow Prop$$

3.5.4 Logical equivalence of schemas

The three different relationships between *Schema*s and *Binding*s that we have defined give rise to several possible ways in which we can define a logical equivalence on schemas. Some of these are the following:

***Definition* 22.** *Schema equivalences*

$$Equiv_1 \stackrel{def}{=} \boldsymbol{\lambda} S, T : Schema.\ \forall\, b : Binding.$$
$$(exactly\_models\ S\ b) \Longleftrightarrow (exactly\_models\ T\ b)$$
$$Equiv_2 \stackrel{def}{=} \boldsymbol{\lambda} S, T : Schema.\ \forall\, b : Binding.$$
$$(restricts\_to\_model\ S\ b) \Longleftrightarrow (restricts\_to\_model\ T\ b)$$
$$Equiv_3 \stackrel{def}{=} \boldsymbol{\lambda} S, T : Schema.\ \forall\, b : Binding.$$
$$(models\ S\ b) \Longleftrightarrow (models\ T\ b)$$

We can prove that these relationships are all different from each other.

If two schemas have signatures which are unequal permutations of each other, then they can be equivalent under $Equiv_2$, without being equivalent under $Equiv_1$.

Theorem 4 $Equiv_2$_not_$Equiv_1$.

$$\exists\, S, T : Schema.\ (Equiv_2\ S\ T) \wedge \neg(Equiv_1\ S\ T)$$

Proof. The *Schemas* $([(y, nat\_ty), (x, \_nat\_ty)], trueProp)$ and $([(x, nat\_ty), (y, nat\_ty)], trueProp)$ are equivalent under $Equiv_2$ but not under $Equiv_1$. □

However, all schemas which are equivalent under $Equiv_1$ are also equivalent under $Equiv_2$:

Theorem 5 $Equiv_1$_implies_$Equiv_2$.

$$\forall\, S, T : Schema.\ (Equiv_1\ S\ T) \Rightarrow (Equiv_2\ S\ T)$$

Proof. See Proof 1 in Appendix A.

Two schemas with unequal signatures can be equivalent under $Equiv_3$ if they have logically equivalent predicates. Such schemas are not necessarily equivalent under $Equiv_1$ or $Equiv_2$.

Theorem 6 $Equiv_3$_not_$Equiv_1$.

$$\exists\, S, T : Schema.\ (Equiv_3\ S\ T) \wedge \neg(Equiv_1\ S\ T)$$

Proof. The *Schemas* $([\ ], trueProp)$ and $([(x, nat\_ty)], trueProp)$ are equivalent under $Equiv_3$ but not under $Equiv_1$. □

Theorem 7 $Equiv_3$_not_$Equiv_2$.

$$\exists\, S, T : Schema.\ (Equiv_3\ S\ T) \wedge \neg(Equiv_2\ S\ T)$$

Proof. The example used for Theorem 6 works here as well. □

Of the three relationships, $Equiv_2$ seems closest to the notion of logical equivalence that is used in the ZRM. $Equiv_1$ is too strong, since it requires equivalent schemas to have their signature items listed in the same order. $Equiv_3$ seems too weak, since it does not take account of the schema signatures. We therefore define:

***Definition* 23.** *Schema equivalence*

$$Equiv \stackrel{def}{=} Equiv_2$$

3.5.5 Equality

We shall assume that all given types have computational equalities defined on them. In other words, if we have $G : GivenType$, we shall assume an equality

$$G\_eq : (Typ\ (given\_ty\ G)) \rightarrow (Typ\ (given\_ty\ G)) \rightarrow bool$$

This enables us to define, by induction on *GivenType* and on *Ztype*, a general computational equality.

***Definition* 24.** *Equal*

$$Equal : \mathbf{\Pi}\, t{:}Ztype\mathbf{.}\ (Typ\ t) \rightarrow (Typ\ t) \rightarrow bool$$

3.5.6 Handling failed lookups

Just as in embedding 2, it is possible for the *lookup* function to fail when asked to look up an identifier in a signature. Unfortunately, dealing with the consequences of failure is now a lot more difficult. The difficulty arises from the fact that our notion of *Binding* is more general than that in embedding 2. In embedding 2 *Binding*s were typed by schema predicates. This means that in order to know whether a particular identifier can be looked up in a particular *Binding*, it is sufficient to know the *Signature* that gives the type of that *Binding*. This is the reason why the definition we have given for the schema S under embedding 2 can be successfully type-checked by LEGO. LEGO can check that each identifier that is looked up actually occurs in the *Signature S_sig*, and this information is enough to show that that identifier can be successfully looked up in any binding *bin* whose type is *Typify S_sig*.

We cannot use the same trick to handle failed *lookup*s in embedding 3. In this embedding schema predicates need to look up identifiers in arbitrary *Binding*s, unrelated to the associated schema signatures. We must therefore find some other way of dealing with failed *lookup*s.

Let us suppose we wish to encode the schema S under embedding 3. This schema has as its predicate:

$$P \mathrel{\widehat{=}} x < y$$

We shall represent this predicate by a UTT predicate of the form:

$$\boldsymbol{\lambda} b : \mathit{Binding}\boldsymbol{.}\ (\mathit{lookup}\ (x, \mathit{nat\_ty})\ b)\ \ \mathbf{??}\ \ (\mathit{lookup}\ (y, \mathit{nat\_ty})\ b)$$

The question is what shall we use in place of the **??** to represent the $<$ relation. The operation we need must have as its type something like $(\mathit{Error} + \mathit{small\_item}) \longrightarrow (\mathit{Error} + \mathit{small\_item}) \longrightarrow \mathit{Prop}$. In the case where both of the *lookup*s succeed, this operation must behave like the $<$ relation. In other cases we must decide what this operation should do. There are two kinds of failure that may happen. First, the *lookup* may fail entirely because an identifier-Ztype pair is looked up in a binding in which it does not occur. Second, it is possible to look up the wrong Ztype. Consider the term:

$$\boldsymbol{\lambda} b : \mathit{Binding}\boldsymbol{.}\ (\mathit{lookup}\ (x, \mathit{bool\_ty})\ b)\ \ \mathbf{??}\ \ (\mathit{lookup}\ (y, \mathit{nat\_ty})\ b)$$

We shall need to decide what the operation **??** should do when placed in such a context. Let us consider the options for dealing with these two kinds of failure.

It seems desirable that we should never be able to prove that a predicate is true of a binding b if the predicate looks up an identifier-Ztype combination that is not present in b. There are several ways in which we can try to achieve this.

One technique we can adopt is to explicitly add definedness conditions to schema predicates when we translate them into Lego. For example, the predicate P would become:

$$\begin{array}{l} \lambda b : Binding. \\ \qquad (succeeds\ (lookup\ (x, nat)\ b)) \wedge (succeeds\ (lookup\ (y, nat)\ b)) \wedge \\ \qquad (lookup\ (x, nat)\ b\ \mathbf{??}\ lookup\ (y, nat)\ b) \end{array}$$

In this case it will not matter what the operation **??** does in the case of failed *lookup*s, since the definedness conditions will guarantee that the entire predicate is false. However, this method is inelegant since it seems to be duplicating the purpose of the schema signature.

Perhaps the most obvious solution is to treat the failure of *lookup* as an error condition, and to propagate this error throughout the predicate. This requires us to change the type of schema predicates from $Binding \longrightarrow Prop$ to $Binding \longrightarrow (Prop + Error)$. When a predicate is applied to a binding any failed *lookup*s will be propagated through the term and will result in a value of $(in1\ error)$.

Why should we want to consider other possibilities? The reason is that dealing with sums is awkward, and makes our definitions and theorems more clumsy. For instance we would need to redefine all our logical combinators (such as $\wedge$, $\vee$, $\Rightarrow$, *et c.*) so that they propagate errors. Theorems about these combinators would need to be hedged by conditions stating that their arguments must not be equal to $in1\ error$.

We shall get around the need for sums by taking advantage of the fact that there is an independent means of checking that a binding contains the identifiers required by a schema predicate: we can check that it matches, or can be restricted to, the schema signature. If we only care about the cases when the predicate is applied to bindings that satisfy this condition, then it does not really matter how we deal with the cases where *lookup* fails. So to simplify matters, we can simply allow these cases to "collapse" to some arbitrarily chosen value of type *Prop*. We use the value *absurd*.

This method only works if the schema signature does in fact include all of the identifiers looked up by the schema predicate. We cannot express this syntactic condition within the type theory because our encoding of schema predicates is not sufficiently "deep". However, we have tried to capture the essence of this condition by the well-formedness conditions (Definition 20) mentioned in the previous section.

3.5.7 Wrapping functions

We shall now discuss the technicalities of handling failed *lookup*s by "collapsing to *absurd*". What does this mean in practice? How is the operation **??** to be defined?

When *lookup* fails it returns the value $in1\ error$. So, clearly, one thing that must be done is to perform a case analysis on the result of the *lookup* and return the value *absurd* in the case where the result is $in1\ error$. We shall define a "wrapped" version of the equality predicate "=" which performs this case analysis. In general, we must define similarly wrapped versions of all the UTT terms which represent the Expressions and relations (Rel) of Z. More precisely, whenever we have a predicate whose type is, say, $(Typ\ z) \rightarrow Prop$, where z is some *Ztype*, we shall define a wrapped version of type $(Error + small\_item) \rightarrow Prop$. For a function of type $(Typ\ z_1) \rightarrow (Typ\ z_2)$ we shall define a wrapped version of type $(Error + small\_item) \rightarrow (Error + small\_item)$. This can be generalised for relations and functions of higher arity.

So far we have only discussed the possibility that *lookup* fails outright. In order to define the wrapped version of functions we must also deal with the possibility that the wrong type was looked up. To do this we must employ another elaborate case-analysis which analyses the *Ztype* in a *small_item* and returns *absurd* (or $in1\ error$, if appropriate) if this *Ztype* does not conform to the type of the function or predicate that is being wrapped. To make this more concrete, we shall display the wrapped version of the boolean negation function $inv : (Typ\ bool\_ty) \rightarrow (Typ\ bool\_ty)$. The wrapped version, *INV* has type $(Error + small\_item) \rightarrow (Error + small\_item)$.

$$
\begin{aligned}
INV \stackrel{def}{=} {} & case\ \boldsymbol{\lambda}\_ : Error.\ in\mathbf{1}\ error \\
& \quad (sigma\_rec \\
& \quad\quad (Ztype\_elim\ (\boldsymbol{\lambda} x : Ztype.\ (Typ\ x) \rightarrow (Error + small\_item)) \\
& \quad\quad\quad (\boldsymbol{\lambda}\_ : Typ\ nat\_ty.\ in\mathbf{1}\ error) \\
& \quad\quad\quad (\boldsymbol{\lambda} b : Typ\ bool\_ty.\ in\mathbf{2}\ (inv\ b)) \\
& \quad\quad\quad (\boldsymbol{\lambda} g : GivenType.\ \boldsymbol{\lambda}\_ : Typ\ (given\_ty\ g).\ in\mathbf{1}\ error) \\
& \quad\quad\quad (\boldsymbol{\lambda} x : Ztype.\ \boldsymbol{\lambda}\_ : (Typ\ x) \rightarrow Prop.\ \boldsymbol{\lambda}\_ : Typ\ (finset\_ty\ x).\ in\mathbf{1}\ error) \\
& \quad\quad\quad (\boldsymbol{\lambda} x, y : Ztype.\ \boldsymbol{\lambda}\_ : (Typ\ x) \rightarrow Prop.\ \boldsymbol{\lambda}\_ : (Typ\ y) \rightarrow Prop. \\
& \quad\quad\quad\quad \boldsymbol{\lambda}\_ : Typ\ (prod\_ty\ x\ y).\ in\mathbf{1}\ error)))
\end{aligned}
$$

Wrapped functions are very elaborate to define, even for a simple, unary function like *inv*. The definition of the binary operation **??** is too long to display here. Fortunately, it is possible to define UTT functions which will compute these nestings for us. These functions exploit the fact that *Ztype* is an inductive type: the nesting is computed by induction on the *Ztype*s of the arguments to the nested function, and the *Ztype* of the value to which that function is being applied. The definitions of these functions are given in Appendix C, together with some lemmas about their behaviour.

We shall call these functions *wrappers*. We distinguish the wrapped versions of constants by using the name of the constant written in ALL CAPITALS. The wrapper for unary functions is called *wrap* and has the type:

$$
\begin{aligned}
& \boldsymbol{\Pi}\, z_1, z_2 : Ztype.\ (Typ\ z_1) \rightarrow (Typ\ z_2) \rightarrow \\
& \quad\quad (Error + small\_item) \rightarrow (Error + small\_item)
\end{aligned}
$$

The wrapped version of the function *inv* can be computed using *wrap*, as follows:

$$wrap\ bool\_ty\ bool\_ty\ inv$$

The value obtained is equal to *INV*.

3.5.8 Representing the schema S

Here is the representation of the schema S under embedding 3. The wrapped version of the $<$ relation is represented by the term $LT : (Error + small\_item) \rightarrow (Error + small\_item) \rightarrow Prop$.

$$x \stackrel{def}{=} \mathbf{0}$$
$$y \stackrel{def}{=} \mathbf{1}$$
$$S\_sig \stackrel{def}{=} [(x, nat\_ty), (y, nat\_ty)]$$
$$S\_pred \stackrel{def}{=} \boldsymbol{\lambda} bin{:}Binding.\ ((lookup\ x\ S\_sig)\ bin)\ LT\ ((lookup\ y\ S\_sig)\ bin)$$
$$S \stackrel{def}{=} (S\_sig, S\_pred)$$

3.6 Conclusion

We have considered a number of possibilities for representing (a reduced version of) the Z schema in UTT, and decided upon one technique which provides a good compromise between expressiveness and ease of use.

Chapter 4

A Specification Example

In this chapter we introduce a Z specification which we are going to use as a running example through this thesis. The specification describes a system for recording birthdays and is an adaptation of the BirthdayBook example from the ZRM. We shall present the first schema of the specification, in which the basic system is described, and show how this schema is encoded in LEGO. The rest of the specification will be given in later chapters.

The specification is parameterised over two given sets: a set of names and a set of dates:

$$[NAME, DATE]$$

The basic system is specified by the schema *BirthdayBook*. This describes a state space in which two identifiers are defined: *known*, which is the set of names known to the system, and *birthday*, which is a partial function giving the birthdays associated with the names in its domain. The set *known* is specified to be equal to the domain of the *birthday* function.

$$
\begin{array}{|l}
\textit{BirthdayBook} \\
\hline
known : \mathbb{F}\ NAME \\
birthday : NAME \rightarrow\!\!\!\!\!\!\!+\;\; DATE \\
\hline
known = \mathbf{dom}\ birthday
\end{array}
$$

4.1 An encoding of finite set theory

The Z notation is based upon set theory, and the Birthday Book specification makes use of various set-theoretical constructs such as sets, partial functions, function domains, and equality of sets. We must therefore develop representations of all of these things within UTT.

There are several ways in which set theory can be encoded in type theory. See for instance the encodings presented in the Lego library [JM94] and in previous work on embedding Z in LEGO [Mah90]. Also of interest is Aczel's encoding of constructive set theory in Martin-Lof type theory [1].

As we have already explained, we are going to use an encoding of *finite* sets represented via lists. An advantage of using finite sets is we can define *decidable* membership and equality relations (provided we assume we are given decidable equality relations on the types of members of sets.) It is easier to reason about partial functions since membership of the domain of a partial function is decidable.

However, we would like to emphasise that we do not consider this to be the last word on encoding the set-theoretic aspect of Z in type theory. Our encoding of the rest of the Z notation is not dependent on this choice of method for encoding sets, so it is possible to use a more sophisticated representation of sets if this is required.

The encoding of finite sets as lists is straightforward and is shown in Figure 4–1. It is assumed that there is a decidable equality *eq* on the type of elements, z. The empty set is represented by the empty list *nil*, and the formation of a singleton set is represented by *cons*. The function *member* is used to represent the membership relation *In*. These definitions work correctly provided they are applied to lists which contain no duplicate elements. The set-forming operations (in particular, *Union*) are defined so as to never introduce duplicate elements.

Assume z : $Ztype$; eq : $(Typ\ z) \rightarrow (Typ\ z) \rightarrow bool$

$$
\begin{aligned}
&Null \stackrel{def}{=} nil\ (Typ\ z)\\
&Single \stackrel{def}{=} \boldsymbol{\lambda} x : Typ\ z.\ cons\ x\ Null\\
&In \stackrel{def}{=} member\ eq\\
&Subset \stackrel{def}{=} \boldsymbol{\lambda} s, s' : Typ\ (finset\_ty\ z).\\
&\quad list\_iter\ true\ (\boldsymbol{\lambda} x : Typ\ z.\ \boldsymbol{\lambda} prev : bool.\ andalso\ (In\ x\ s')\ prev)\\
&Set\_eq \stackrel{def}{=} \boldsymbol{\lambda} s, s' : Typ\ (finset\_ty\ z).\ andalso\ (Subset\ s\ s')\ (Subset\ s'\ s)\\
&Add\_one \stackrel{def}{=} \boldsymbol{\lambda} x : Typ\ z.\ \boldsymbol{\lambda} s : Typ\ (finset\_ty\ z).\ if\ (In\ x\ s)\ s\ (cons\ x\ s)\\
&Union \stackrel{def}{=} \boldsymbol{\lambda} s, s' : Typ\ (finset\_ty\ z).\ list\_iter\ s'\ Add\_one\ s\\
&Intersect \stackrel{def}{=} \boldsymbol{\lambda} s, s' : Typ\ (finset\_ty\ z).\\
&\quad list\_iter\ Null\\
&\qquad (\boldsymbol{\lambda} x : Typ\ z.\ \boldsymbol{\lambda} prev : (finset\_ty\ z).\ if\ (In\ x\ s')\ (Add\_one\ x\ prev)\ prev)\\
&\qquad s
\end{aligned}
$$

Discharge z, eq.

Figure 4–1: Finite sets encoded as lists

Assume $z, z' : Ztype$

Assume $eq : (Typ\ z) \rightarrow (Typ\ z) \rightarrow bool$

Assume $eq' : (Typ\ z') \rightarrow (Typ\ z') \rightarrow bool$

$$[eq_2 = \lambda x, y : Typ\ (prod\_ty\ z\ z').\ andalso\ (eq\ x.1\ y.1)\ (eq'\ x.2\ y.2)$$
$$rel\_ty \stackrel{def}{=} finset\_ty\ (prod\_ty\ z\ z')$$
$$RelIn \stackrel{def}{=} In\ eq_2$$
$$RelUnion \stackrel{def}{=} Union\ eq_2$$
$$Rel\_eq \stackrel{def}{=} Set\_eq\ eq_2$$
$$Dom \stackrel{def}{=} map(\lambda x : Typ\ (prod\_ty\ z\ z').\ x.1$$

Discharge z, z', eq, and eq'.

Figure 4–2: Finite relations encoded as lists

We use the wrapping functions described in the previous chapter to form partial versions of these operations.

In order to reason about the Birthday Book, we have developed a small library of lemmas and theorems about the set theoretic operations. Some of the contents of this library is listed (without proofs) in Appendix B.2.

4.1.1 Relations and functions

In Figure 4–2 we give our encoding of set-theoretic relations, and various operations relating to them. Following Z, we shall represent set-theoretic functions as relations which satisfy the property of being many-to-one. However, in Z' we do not allow such logical constraints to become part of the definition of types, so we cannot define a type of set-theoretic functions. To encode schema signatures which contain functions, we shall have to make the many-to-one property an explicit part of the schema predicate. The definitions of union, singleton formation, and equality on relations are used to represent the corresponding operations on functions, again with the proviso that appropriate conditions are added to the predicates of schemas in which these operations are used.

$$\begin{aligned}
&\mathit{fun_ty} \stackrel{def}{=} \mathit{rel_ty}\\
&\mathit{FunSingle} \stackrel{def}{=} \mathit{RelSingle}\\
&\mathit{FunUnion} \stackrel{def}{=} \mathit{RelUnion}\\
&\mathit{Fun_eq} \stackrel{def}{=} \mathit{Rel_eq}
\end{aligned}$$

4.1.2 Dealing with partial functions

In order to formalise partial set-theoretic functions within a total type theory such as UTT, we must make explicit the way in which we handle the cases where such functions are applied to arguments outside of their domain. The semantics of partially defined terms is an aspect of Z that has raised many interesting questions. While this topic is not a main focus of this thesis, it is one which we cannot avoid completely, so we shall discuss it briefly here. Though we shall phrase our discussion in terms of our formal encoding of Z, the issues that are being discussed are really questions about the semantics of Z itself.

We define an operation *Apply* which applies a set-theoretic function. This has the following type:

$$\begin{aligned}
&\mathbf{\Pi}\, z, z' \mid \mathit{Ztype}.\, \mathbf{\Pi}\, \mathit{eq} : (\mathit{Typ}\ z) \rightarrow (\mathit{Typ}\ z) \rightarrow \mathit{bool}.\\
&\quad (\mathit{Typ}\ (\mathit{Rel}\ z\ z')) \rightarrow (\mathit{Typ}\ z) \rightarrow (\mathit{Error} + (\mathit{Typ}\ z'))
\end{aligned}$$

(The full definition is given in Appendix C.1.) Here z and z' are the *Ztype*s of the domain and codomain of the function to be applied, and *eq* is a computational equality on $\mathit{Typ}\ z$. *Apply* performs function application by searching the list that represents the function until it finds the value to which the function is being applied. The *eq* argument is needed for comparison during this searching. If the search fails, then the value $\mathit{in}1\ \mathit{error}$ is returned. Dealing with partiality means, in the context of this thesis, dealing with the cases in which *Apply* returns this error value.

Suppose we wish to encode the following schema

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad S \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}} \\
f, g : \mathbb{N} \rightarrowtail\!\!\!\!\!\!+ \ \mathbb{N} \\
x : \mathbb{N} \\
\hline
f\ x < g\ x \\
\hline
\end{array}
$$

In translating the predicate of this schema, we must decide what to do in the cases where either $f\ x$ or $g\ x$ are undefined.

One possibility is to redefine the type of schema predicate so that a failed function would result in an error value. (This approach was suggested in the previous chapter as a means of handling failed *lookups*.) This is reminiscent of suggestions for adopting a 3-valued logic for dealing with undefined terms in specification languages (*e.g.*,[BCJ84]). For the reasons mentioned in Section 3.5.6 we shall not pursue this approach, but we believe it warrants further study.

The method we have adopted for dealing with partial functions is very simple, and is essentially one of those suggested in Section 2.5 of the ZRM: whenever we apply a function we also explicitly require that the value to which it is applied must lie within the domain of the function. This allows us to keep the type of schema predicate as *Binding* $\longrightarrow$ *Prop*. For instance the predicate part of the schema S will be translated as follows. First we define the wrapped version of *Apply*.

$$
\begin{array}{l}
APPLY \stackrel{def}{=} \text{[See Appendix C.2.]} \\
: \mathbf{\Pi}\, z_1, z_2 \,|\, Ztype.\, \mathbf{\Pi}\, eq \,|\, (Typ\ z_1) \rightarrow (Typ\ z_1) \rightarrow bool. \\
(Error + small\_item) \rightarrow (Error + small\_item) \rightarrow (Error + small\_item)
\end{array}
$$

We choose distinct natural numbers to represent the identifiers f, g, and x. We pair these with the appropriate decorations and *Ztype*s to form the signature items used in encoding the signature of S:

$$f\_item \stackrel{def}{=} ((\mathbf{0}, blank), fun\_ty\ nat\_ty\ nat\_ty) : sig\_item$$
$$g\_item \stackrel{def}{=} ((\mathbf{1}, blank), fun\_ty\ nat\_ty\ nat\_ty) : sig\_item$$
$$x\_item \stackrel{def}{=} ((\mathbf{2}, blank), nat\_ty) : sig\_item$$

$$S\_sig \stackrel{def}{=} [f\_item, g\_item, x\_item] : Signature$$

We now encode the predicate of S as follows:

$$\begin{array}{l} S\_pred \stackrel{def}{=} \boldsymbol{\lambda} b{:}Binding. \\ \quad [f = lookup\ f\_item\ b] \\ \quad [g = lookup\ g\_item\ b] \\ \quad [x = lookup\ x\_item\ b] \\ \quad\ (\exists\, n_1{:}nat.\, EQUAL\ (APPLY\ f\ x)\ (in\mathbf{2}\ (nat\_ty, n_1))) \wedge \\ \quad\ (\exists\, n_2{:}nat.\, EQUAL\ (APPLY\ g\ x)(in\mathbf{2}\ (nat\_ty, n_2))) \wedge \\ \quad\ (IS\_TRUE\ (LT\ (APPLY\ f\ x)\ (APPLY\ g\ x))) \end{array}$$

The signature and predicate are paired to complete the encoding of S:

$$S \stackrel{def}{=} (S\_sig, S\_pred) : Schema$$

4.2 Representing the given sets of the BirthdayBook

To represent the BirthdayBook specification in LEGO we first define the given types that it uses. We do this by defining an inductive type *GivenType* whose constructors are the names of the two given types. Using an inductive type like this has the disadvantage that the pool of given types is fixed once *GivenType* is defined. To add a new given type we must restart with a new definition of *GivenType*. The advantage is that we are provided with an elimination rule for *GivenType*. We cannot dispense with this, and simply declare or define new given types as we need them, because we need to know the totality of all given types in order to define the type *Ztype* (Definition 2).

$$\text{Inductive}[\mathit{GivenType}]$$
$$\text{Constructors}[\mathit{Name\_ty}, \mathit{Date\_ty}]$$

We use the elimination rule on *GivenType* to define a decidable equality:

$$\mathit{GivenType\_eq} \stackrel{def}{=} [\text{omitted}] : \mathit{GivenType} \longrightarrow \mathit{GivenType} \longrightarrow \mathit{bool}$$

We also define an operation *Typ_gtype* : which maps *GivenType*s to *Type*s. This operation is needed in order to define *Typ* (Definition 4.) We shall introduce two type parameters which we use as the range of *Typ_gtype*.

$$\mathit{Name}, \mathit{Date} : \mathit{Type}$$

$$\mathit{Typ\_gtype} \stackrel{def}{=} \mathit{GivenType\_rec}\ \mathit{Name}\ \mathit{Date}$$
$$: \mathit{GivenType} \longrightarrow \mathit{Type}$$

We shall assume that we have decidable equalities on these two types:

$$\mathit{Name\_eq} : \mathit{Name} \longrightarrow \mathit{Name} \longrightarrow \mathit{bool}$$
$$\mathit{Date\_eq} : \mathit{Date} \longrightarrow \mathit{Date} \longrightarrow \mathit{bool}$$

4.3 Putting it all together

Once the given types are defined we can load the definition of *Ztype*, which is parameterized over *GivenType*, and then all the remaining definitions of signatures, bindings, schemas, *et c.*

We then encode the signature items and signature of the BirthdayBook schema. As we did for the schema S, we select distinct natural numbers to represent the identifiers and pair these with the appropriate decorations and

Ztypes:

$$
\begin{aligned}
&known\_type \stackrel{def}{=} finset\_ty\ (Given\_ty\ Name\_ty) : Ztype \\
&known\_item \stackrel{def}{=} ((\mathbf{0}, blank), known\_type) : sig\_item \\
&birthday\_type \stackrel{def}{=} fun\_ty\ (Given\_ty\ Name\_ty)\ (Given\_ty\ Date\_ty) : Ztype \\
&birthday\_item \stackrel{def}{=} ((\mathbf{1}, blank), birthday\_type) : sig\_item \\
&BB\_sig \stackrel{def}{=} [known\_item, birthday\_item] : Signature
\end{aligned}
$$

We then define the predicate, making use of the wrapped versions of the set-theoretical constants, and pair this with the signature to form the schema.

Definition 25 BirthdayBook.

$$
\begin{aligned}
&BB\_pred \stackrel{def}{=} \boldsymbol{\lambda} B : Binding. \\
&\quad [known \stackrel{def}{=} lookup\ known\_item\ B] \\
&\quad [birthday \stackrel{def}{=} lookup\ birthday\_item\ B] \\
&\quad IS\_TRUE\ (EQUAL\ known\ (DOM\ birthday))
\end{aligned}
$$

$$
BirthdayBook \stackrel{def}{=} (BB\_sig, BB\_pred) : Schema
$$

4.4 Proving well-formedness

Theorem 8 BirthdayBook_well_formed.

$$
Well\_formed\ BirthdayBook
$$

Proof. First we must show that *BirthdayBook* is *Up_closed*:

$$
\begin{aligned}
&\forall\, b : Binding. \\
&\qquad (restricts\_to\_model\ BirthdayBook\ b) \Rightarrow (models\ BirthdayBook\ b)
\end{aligned}
$$

By doing introductions and expanding definitions we transform the proof context to the following:

$$b : Binding$$
$$H : \exists\, b' : Binding.\ ((restrict\ b\ BB\_sig) = (in\mathbf{2}\ b')) \wedge (BB\_pred\ b')$$

$$? : IS\_TRUE\ (EQUAL\ (lookup\ known\_item\ b)\ (DOM\ (lookup\ birthday\_item\ b)))$$

We then do some eliminations on H and add the following to the context:

$$t : Binding$$
$$H_2 : restrict\ b\ BB\_sig = in\mathbf{2}\ t$$
$$H_3 : BB\_pred\ t$$

By using Lemma 18 we can show that the results of looking up *known_item* and *birthday_item* in b are equal to those obtained by looking up these items in t. Rewriting with these equalities we transform the goal to:

$$? : IS\_TRUE\ (EQUAL\ (lookup\ known\_item\ t)\ (DOM\ (lookup\ birthday\_item\ t)))$$

This is the same as H_3, so the proof of up-closure is complete.

Now we must tackle the proof of down-closure. Unfortunately, we have not been able to refine this proof so as to make it as simple as that of up-closure. Our goal is:

$$? : \forall\, b : Binding.\ (models\ BirthdayBook\ b) \Rightarrow (succeeds\ (restrict\ b\ BirthdayBook.sig)) \Rightarrow (restricts\_to\_model\ BirthdayBook\ b)$$

We introduce a binding b, and then expand definitions to simplify the goal. Next we do case analyses on the results of looking up the items *known_item* and *birthday_item* in b. In the cases where one or other of these *lookup*s returns the value $in\mathbf{1}\ error$, by substituting this value into the goal we can reduce it to the form

$$absurd \Rightarrow (succeeds\ (restrict\ b\ BirthdayBook.sig)) \Rightarrow \\ restricts\_to\_model\ BirthdayBook\ b$$

because the wrapped functions *IS_TRUE*, *EQUAL*, and *DOM* collapse to *absurd* when applied to an error value. This completes the proof in these cases.

In the case where both of the *lookup*s succeed we derive the following proof context:

$$\begin{array}{l} t, t_1 : small\_item \\ H_1 : lookup\ known\_item\ b = in2\ t \\ H_2 : lookup\ birthday\_item\ b = in2\ t_1 \\ H_3 : IS\_TRUE\ (EQUAL\ (in2\ t))\ (DOM\ (in2\ t_1)) \\ H_3 : succeeds\ (restrict\ b\ BirthdayBook.sig) \\ \\ ? : restricts\_to\_model\ BirthdayBook\ b \end{array}$$

We expand the definition of *restricts_to_model*, and then do an existential introduction which gives us two goals:

$$\begin{array}{l} ?_1 : Binding \\ ?_2 : (restrict\ b\ BB\_sig = in2\ ?_1) \wedge (BB\_pred\ ?_1) \end{array}$$

The binding which we supply for goal $?_1$ is the following:

$$[(known\_item, t.2), (birthday\_item, t_1.2)]$$

We must then show that this binding is that obtained by *restrict*ing b to *BB_sig*, and that it satisfies the predicate *BB_sig*. We prove this by expanding the definitions of *restrict* and *BB_pred* and rewriting with the equalities H_1 and H_2. □

We have devoted so much attention to the proof of well-formedness for the schema *BirthdayBook* because this is an example of a routine proof obligation which will need to be discharged whenever a new schema is introduced. In proof-checkers which provide the ability to write user-defined tactics (such as HOL, Coq, or NuPrl[Con86]) routine proofs like these can be completely automated. The LEGO proof-checker, unfortunately, does not have this facility. However, the work required can be reduced by finding simple, reusable proof scripts for routine results. The proof of up-closure is simple, short, and easy to reuse because most of its work is encapsulated in the main lemma that is used. The proof of down-closure is less satisfactory, involving complicated equality manipulations and uses of the "Equiv" command to rewrite the goal. These make the proof less reusable. It seems likely that further analysis would enable us to find a more elegant proof.

Chapter 5

Encoding Z : Logical schema operations

In this chapter and the next we shall discuss the operations provided by Z for putting schemas together in a modular fashion. This chapter is concerned with the "logical" operations of schema conjunction, disjunction, universal and existential quantification, *et c.* while Chapter 6 deals with the conventions used for describing state changes in systems. The logical operations make up the syntax class of schema expressions which is described in Figure 5–1.

We shall represent the constructors for schema expressions by operations on *Schema*s. For instance, the operation of schema conjunction is represented by a function of type $Schema \rightarrow Schema \rightarrow Schema$. This encoding technique allows us to represent individual schema expressions, but does not provide the ability to quantify over the class of all schema expressions. (Our embedding could be extended to allow this, using techniques similar to those used in Sections 2.2.5 and 3.4.2, but we have not seen any pressing reasons for adding this extra complexity.)

In the rest of this chapter we discuss the semantics of the logical operations, give the details of their encodings, and use the encoding to prove some results about their metatheoretic properties. These theorems have all been formally verified using LEGO. A few of the proofs are described in this chapter, in order

Schema-Exp ::=	Schema		*Flat schema*
	|	Schema-Exp $\wedge$ Schema-Exp	*Schema conjunction*
	|	Schema-Exp $\vee$ Schema-Exp	*Schema disjunction*
	|	Schema-Exp $\Rightarrow$ Schema-Exp	*Schema implication*
	|	Schema-Exp $\Leftrightarrow$ Schema-Exp	*Schema equivalence*
	|	$\neg$ Schema-Exp	*Schema negation*
	|	Schema-Exp $\setminus$ Declaration	*Hiding*
	|	$\forall$ Schema-Exp $\bullet$ Schema-Exp	*Universal quantification*
	|	$\exists$ Schema-Exp $\bullet$ Schema-Exp	*Existential quantification*

Figure 5–1: The syntax of the logical schema operations

to illustrate the kinds of techniques used, while the remainder are described in Appendix A.

Results such as Theorems 11 and 13, *et c* concern the relationship between the operations on schemas and the different notions of model. Such results form the basis for the intended use of our system for supporting modular reasoning about Z specifications. The ability to state and to prove these theorems is a consequence of the expressive power of the encoding technique that we have used: these results cannot be stated in a "shallower" embedding such as that of [BG94]. However, we shall see that the use of a constructive type theory places some limitations on the results which can be proved.

5.1 The binary propositional operations

The semantics of the binary propositional operations ($\wedge, \vee, \Rightarrow$, and $\Leftrightarrow$) is described in the ZRM as follows:

> For one of the binary operations to be allowed, its two arguments must have type compatible signatures. The signatures are joined to form the signature of the result. The truth of its property in any

binding b is defined in terms of the truth in the argument schemas of the restrictions of b to their signatures. For example, the property of $S \lor T$ is true in a binding b if and only if either the property of S is true in the restriction of b to the signature of S or the property of T is true in the restriction of b to the signature of T (or both). The other operations follow the rules for propositional connectives [not shown].

5.1.1 Type compatibility

Two signatures are *type compatible* if each variable that is common to the two has the same type in both. This is a decidable, syntactic condition so we shall write a function in LEGO which checks it:

$$\mathit{type\_compatible\_fun} \stackrel{def}{=} \text{[See Appendix C.1]} : \mathit{Signature} \longrightarrow \mathit{Signature} \longrightarrow \mathit{bool}$$

From this we derive a relation:

$$\mathit{type\_compatible} \stackrel{def}{=} \boldsymbol{\lambda} s_1, s_2 : \mathit{Signature}\textbf{.}\ \mathit{is\_true}\ (\mathit{type\_compatible\_fun}\ s_1\ s_2)$$
$$: \mathit{Signature} \longrightarrow \mathit{Signature} \longrightarrow \mathit{Prop}$$

The advantage of defining the predicate in terms of a computational procedure for checking type compatibility is that in cases where we are required to prove that two defined (as opposed to hypothetical) *Signature*s are type-compatible, we can make the proof-checker do the work by checking whether the application of *type_compatible_fun* normalises to *true*. (In LEGO parlance, we use the command `Refine Eq_refl`.)

Well-formedness is preserved if a schema has its signature extended with a type-compatible signature which contains no repeated identifiers:

Theorem 9 Extend_well_formed.

$$\forall\, S : \mathit{Schema}\textbf{.}\ \forall\, \mathit{Sig} : \mathit{Signature}\textbf{.}$$
$$(\mathit{Well\_formed}\ S) \land (\mathit{unique\_idents}\ \mathit{Sig}) \land (\mathit{type\_compatible}\ S\textbf{.1}\ \mathit{Sig}) \Rightarrow$$
$$(\mathit{Well\_formed}\ (\mathit{join}\ S\textbf{.1}\ \mathit{Sig}, S\textbf{.2}))$$

Proof. By doing introductions and expanding the definition of Well_formed we arrive at the following proof state:

S : *Schema*

Sig : *Signature*

$H : (Up\_closed\ S) \wedge (Down\_closed\ S) \wedge (unique\_idents\ S.\mathbf{1})$

$H_1 : unique\_idents\ Sig$

$? : Up\_closed\ (join\ S.\mathbf{1}\ Sig, S.\mathbf{2})$

$?_1 : Down\_closed\ (join\ S.\mathbf{1}\ Sig, S.\mathbf{2})$

$?_2 : unique\_idents\ (join\ S.\mathbf{1}\ Sig)$

First we show up-closure. Some more introductions give us the context:

$b : Binding$

$H_2 : restricts\_to\_model\ (join\ S.\mathbf{1}\ Sig, S.\mathbf{2}) b$

$?_4 : models\ (join\ S.\mathbf{1}\ Sig, S.\mathbf{2}) b$

Since S is up-closed, we can prove this goal by showing

$?_4 : restricts\_to\_model\ (join\ S.\mathbf{1}\ Sig, S.\mathbf{2})\ b$

Now, we can show from hypothesis H_2 that the binding b can be successfully restricted to the signature $join\ S.\mathbf{1}\ Sig$, yielding a binding t and a proof $H_3 : S.\mathbf{2}\ t$. Lemmas 27 and 26 then allow us to conclude the following:

$H_4 \stackrel{def}{=} succeeds\ (restrict\ b\ S.\mathbf{1})$:

$H_5 \stackrel{def}{=} succeeds\ (restrict\ b\ Sig)$:

We can then use Lemma 24 to derive:

$H_6 \stackrel{def}{=} \ldots : restrict\ b\ S.\mathbf{1} = restrict\ t\ S.\mathbf{1}$

By rewriting the goal with this equality we get a new goal:

$?_5 : restricts\_to\_model\ (join\ S.\mathbf{1}\ Sig, S.\mathbf{2})\ t$

Next we apply the assumption that S is down-closed. This gives us the goals:

$$?_5 : \mathit{models}\ S\ t$$
$$?_6 : \mathit{succeeds}\ (\mathit{restrict}\ t\ S.1)$$

We have a proof of $?_5$ among our hypotheses (H_3). To prove $?_6$ we write with the equality H_6 and then use H_4. This completes the proof that ($\mathit{join}\ S.1\ \mathit{Sig}, S.2$) is up-closed.

The proof of down-closure is very similar to that of up-closure, so we omit its description. To show the final subgoal, $?_2$, we simply apply Lemma 22 which states that *join* preserves the property *unique_idents*. □

5.1.2 Schema conjunction

Schema conjunction was defined in the previous chapter. Here is that definition once more:

Definition 26 And.

$$\begin{aligned} &\mathit{And} \stackrel{def}{=} \lambda S, S' : \mathit{Schema}. \\ &\qquad (\mathit{join}\ S.\mathit{sig}\ S'.\mathit{sig},\ \lambda b : \mathit{Binding}.\ (S.\mathit{pred}\ b) \wedge (S'.\mathit{pred}\ b)) \\ &: \mathit{Schema} \rightarrow \mathit{Schema} \rightarrow \mathit{Schema} \end{aligned}$$

5.1.3 Theorems about schema conjunction

First we show that schema conjunction preserves well-formedness when applied to type compatible schemas:

Theorem 10 And_preserves_well_formedness.

$$\begin{aligned} &\forall\, S, T : \mathit{Schema}. \\ &\qquad (\mathit{Well\_formed}\ S) \wedge (\mathit{Well\_formed}\ T) \wedge (\mathit{type\_compatible}\ S.1\ T.1) \Rightarrow \\ &\qquad\quad \mathit{Well\_formed}\ (\mathit{And}\ S\ T) \end{aligned}$$

Proof. See Proof 2 of Appendix A. This proof is very similar to that of Theorem 9.

Next comes our main metatheorem about the *And* operation with respect to the *model* relationship between *Schemas* and *Bindings*. This theorem states that a *Binding* is a model of $S \wedge T$ if and only if it is a model of both S and T.

Theorem 11 And_model_char.

$$\forall\, S, T : Schema.\ \forall\, b : Binding.$$
$$(models\ S\ b) \wedge (models\ T\ b) \iff models\ (And\ S\ T)\ b$$

Proof. We introduce S, T, and b into the context. By expanding the definition of *models* we can reduce the remaining goal to the following:

$$?\ :\ ((S.\mathbf{2}\ b) \wedge (T.\mathbf{2}\ b)) \iff ((S.\mathbf{2}\ b) \wedge (T.\mathbf{2}\ b))$$

which is easily proved. □

Next we show that *And* is commutative with respect to the *model* relationship:

Theorem 12 And_model_commutes.

$$\forall\, S, T : Schema.\ \forall\, b : Binding.$$
$$models\ (And\ S\ T)\ b \Rightarrow models\ (And\ T\ S)\ b$$

Proof. This follows easily from Theorem 11. □

Similar results can be proved about restricting models. They apply only to schemas which are well-formed and type-compatible:

Theorem 13 And_restricting_model_char.

$$\forall\, S, T : Schema.\ \forall\, b : Binding.$$
$$(Well\_formed\ S) \wedge (Well\_formed\ T) \wedge (type\_compatible\ S.\mathbf{1}\ T.\mathbf{1}) \Rightarrow$$
$$(restricts\_to\_model\ S\ b) \wedge (restricts\_to\_model\ T\ b) \iff$$
$$restricts\_to\_model\ (And\ S\ T)\ b$$

Proof. By doing introductions we arrive at the following proof context:

$S, T : Schema$

$b : Binding$

$W_1 : Well\_formed\ S$

$W_2 : Well\_formed\ T$

$C : type\_compatible\ S.\mathbf{1}\ T.\mathbf{1}$

$? : ((restricts\_to\_model\ S\ b) \wedge (restricts\_to\_model\ T\ b)) \Rightarrow$
$\quad (restricts\_to\_model\ (And\ S\ T)\ b)$

$?_1 : (restricts\_to\_model\ (And\ S\ T)\ b) \Rightarrow$
$\quad ((restricts\_to\_model\ S\ b) \wedge (restricts\_to\_model\ T\ b))$

We shall only describe the proof of the first subgoal since both subgoals have very similar proofs. Our first step is to introduce the antecedent of the goal as a new hypothesis H:

$H : (restricts\_to\_model\ S\ b) \wedge (restricts\_to\_model\ T\ b)$

$?_3 : restricts\_to\_model\ (And\ S\ T)\ b$

To prove goal $?_3$ we use the fact that *And* preserves down-closure (Theorem 10) to reduce it to the following two subgoals:

$?_4 : models\ (And\ S\ T)\ b$

$?_5 : succeeds\ (restrict\ b\ (And\ S\ T).\mathbf{1})$

Theorem 11 (*And_model_char*) enables us to reduce the first goal to the following two goals:

$?_6 : models\ S\ b$

$?_7 : models\ T\ b$

These subgoals can be proved using the facts that S and T are up-closed (by hypotheses W_1 and W_2, respectively), and b is a restricting model of each of these schemas (by hypothesis H.)

To prove the outstanding subgoal, $?_5$, we use Lemma 23 which states that if a binding can be successfully restricted to each of two signatures sig_1 and sig_2, then it can be successfully restricted to the signature formed by *join*ing sig_1 and sig_2. □

Theorems 11 and 13 are very useful because they allow us to prove results about conjoined schemas in a modular fashion. For instance, if we know that all restricting models of a schema S have some property P, Theorem 13 allows us to conclude that, for all schemas T, all restricting models of $S \wedge T$ will also have property P. We shall see an example of this kind of reasoning in Chapter 7.

We show that schema conjunction is commutative with respect to restricting models:

Theorem 14 And_restricting_model_commutes.

$$\begin{array}{l} \forall\, S, T : Schema.\ \forall\, b : Binding. \\ \quad (Well\_formed\ S) \wedge (Well\_fomed\ T) \wedge (type\_compatible\ S.1\ T.1) \Rightarrow \\ \qquad (restricts\_to\_model\ (And\ S\ T)\ b \Rightarrow restricts\_to\_model\ (And\ T\ S)\ b) \end{array}$$

Proof. See Proof 3 in Appendix A

So far we have proved theorems about schema conjunction with respect to the *model* relationship and the *restricts_to_model* relationship. We shall not bother to show similar theorems about the *exactly_models* relationship. This is not because such theorems cannot be obtained, but because we believe they are neither as elegant nor as useful as the theorems about restricting models. As we have seen, these two definitions of the relationship between schemas and bindings are more or less equivalent: exact models are restricting models (Theorem 2) and restricting models yield exact models (Theorem 3). Restricting models also seem to give us the best definition of logical equivalence on schemas (Section 3.5.4). If we formulate the result of Theorem 13 in terms of exact models, instead of restricting models, we find that we must still talk about restricting

*Binding*s to *Signature*s, and the result is not so elegant. For example, here is one possible reformulation:

$$\begin{array}{l}\forall\, S, T : Schema.\ \forall\, b : Binding. \\ \quad (Well\_formed\ S) \wedge (Well\_formed\ T) \wedge (type\_compatible\ S.\mathbf{1}\ T.\mathbf{1}) \Rightarrow \\ \qquad (\exists\, b_1 : Binding.\ (restrict\ b\ S = in\mathbf{2}\ b_1) \wedge (exactly\_models\ S\ b_1) \wedge \\ \qquad \exists\, b_2 : Binding.\ (restrict\ b\ T = in\mathbf{2}\ b_1) \wedge (exactly\_models\ T\ b_1)) \iff \\ \qquad (exactly\_models\ (And\ S\ T)\ b)\end{array}$$

Now we give a result involving the *Has_prop* relationship:

Theorem 15 And_property.

$$\begin{array}{l}\forall\, S, T : Schema.\ \forall\, P : Predicate. \\ \quad ((Well\_formed\ S) \wedge (Well\_fomed\ T)) \Rightarrow \\ \quad (type\_compatible\ S.\mathbf{1}\ T.\mathbf{1}) \Rightarrow \\ \quad ((Has\_prop\ S\ P) \vee (Has\_prop\ T\ P)) \Rightarrow \\ \qquad\quad (Has\_prop\ (And\ S\ T)\ P)\end{array}$$

Proof. By expanding definitions and using the introduction tactic, we arrive at the following context:

$$\begin{array}{l} S, T : Schema \\ P : Predicate \\ H : (Well\_formed\ S) \wedge (Well\_formed\ T) \\ H_1 : type\_compatible\ S.\mathbf{1}\ T.\mathbf{1} \\ H_2 : (Has\_prop\ S\ P) \vee (Has\_prop\ T\ P) \\ b : Binding \\ H_3 : restricts\_to\_model\ b\ (And\ S\ T) \\ ? : P\ b \end{array}$$

Using Theorem 13 we can conclude:

$$H_4 : restricts\_to\_model\ b\ SH_5 : restricts\_to\_model\ b\ T$$

We now apply or-elimination to hypothesis H_2 and verify that the goal is satisfied in each case. □

5.1.4 Schema disjunction

Schema disjunction is encoded as follows:

Definition 27 Or.

$$\begin{aligned}&Or \stackrel{def}{=} \boldsymbol{\lambda} S, S' : Schema\boldsymbol{.}\\ &\qquad (join\ S\boldsymbol{.}sig\ S'\boldsymbol{.}sig,\ \ \boldsymbol{\lambda} b : Binding\boldsymbol{.}\ (S\boldsymbol{.}pred\ b) \vee (S'\boldsymbol{.}pred\ b))\\ &: Schema \longrightarrow Schema \longrightarrow Schema\end{aligned}$$

A consequence of this definition is that in order for a binding to be a restricting model of a disjunction $Or\ S\ T$, it must contain the components listed in the signatures of both S and T. At first sight, this seems to be at odds with the description of *Or* given in the ZRM:

> [...] the property of $S \vee T$ is true in a binding b if and only if either the property of S is true in the restriction of b to the signature of S or the property of T is true in the restriction of b to the signature of T (or both).

The anomaly disappears if we recall that bindings in Z are typed by schema signatures. The quoted extract does not explicitly state the type of the binding b to which it refers, but this can only be derived from the signature of the schema $S \vee T$. Hence, the binding b must contain the components referred to in the signatures of both S and T.

We show that *Or* preserves well-formedness.

Theorem 16 Or_preserves_well_formedness.

$$\begin{aligned}&\forall\, S, T : Schema\boldsymbol{.}\\ &\quad (Well\_formed\ S) \wedge (Well\_formed\ T) \wedge (type\_compatible\ S\boldsymbol{.}1\ T\boldsymbol{.}1) \Rightarrow\\ &\qquad Well\_formed\ (Or\ S\ T)\end{aligned}$$

Proof. See Proof 4 in Appendix A.

5.1.5 Theorems about Schema disjunction

Our first theorem about schema disjunction is analogous to that about schema conjunction, and is just as simple to prove.

Theorem 17 Or_model_char.

$$\forall\, S, T : Schema.\ \forall\, b : Binding.$$
$$(models\ S\ b) \vee (models\ T\ b) \iff models\ (Or\ S\ T)\ b$$

Proof. Trivial □

As a consequence, we show that *Or* is commutative with respect to the *model* relationship:

Theorem 18 Or_model_commutes.

$$\forall\, S, T : Schema.\ \forall\, b : Binding.$$
$$models\ (Or\ S\ T)\ b \iff models\ (Or\ T\ S)\ b$$

Dealing with the *restricts_to_model* relationship is more difficult. We cannot prove a simple "Or-introduction" result such as the following:

$$\forall\, S, T : Schema.\ \forall\, b : Binding.$$
$$(restricts\_to\_model\ S\ b) \vee (restricts\_to\_model\ T\ b)$$
$$\Rightarrow restricts\_to\_model\ (Or\ S\ T)\ b$$

This statement is false because a binding that is capable of being restricted to the signature of one of the disjuncts will not in general contain enough identifiers to allow it to be restricted to the signature of the disjunction. The best we can do is to restrict ourselves to talking about bindings that are large enough to allow both restrictions. We must also restrict ourselves to well-formed, type-compatible schemas.

Theorem* 19 *Or_restricting_model_intro.

$$\begin{array}{l}
\forall\, S, T : Schema.\ \forall\, b : Binding. \\
\quad (Well_formed\ S) \wedge (Well_formed\ T) \wedge (type_compatible\ S.\mathbf{1}\ T.\mathbf{1}) \wedge \\
\quad (succeeds\ (restrict\ b\ (join\ S.\mathbf{1}\ T.\mathbf{1}))) \Rightarrow \\
\quad ((restricts_to_model\ S\ b) \vee (restricts_to_model\ T\ b) \Rightarrow \\
\qquad restricts_to_model\ (Or\ S\ T)\ b)
\end{array}$$

Proof. See Proof 5 in Appendix A

We also show that restricting models of a disjunction are restricting models of each disjunct:

Theorem* 20 *Or_restricting_model_elim.

$$\begin{array}{l}
\forall\, S, T : Schema.\ \forall\, b : Binding. \\
\quad (Well_formed\ S) \wedge (Well_formed\ T) \wedge (type_compatible\ S.\mathbf{1}\ T.\mathbf{1}) \Rightarrow \\
\quad (restricts_to_model\ (Or\ S\ T)\ b) \Rightarrow \\
\qquad (restricts_to_model\ S\ b) \vee (restricts_to_model\ T\ b)
\end{array}$$

Proof. See Proof 6 in Appendix A

We show that *Or* is commutative with respect to the *restricts_to_model* interpretation:

Theorem* 21 *Or_restricting_model_commutes.

$$\begin{array}{l}
\forall\, S, T : Schema.\ \forall\, b : Binding. \\
\quad (Well_formed\ S) \wedge (Well_formed\ T) \wedge (type_compatible\ S.\mathbf{1}\ T.\mathbf{1}) \Rightarrow \\
\quad (restricts_to_model\ (Or\ S\ T)\ b \Rightarrow \\
\quad restricts_to_model\ (Or\ T\ S)\ b)
\end{array}$$

Proof. See Proof 7 in Appendix A

Finally, we give a result involving the *Has_prop* relationship:

Theorem* 22 *Or_property.

$$\begin{array}{l}\forall\, S, T : Schema.\ \forall\, P : Predicate.\\ \quad ((Well\_formed\ S) \wedge (Well\_fomed\ T) \Rightarrow\\ \quad (type\_compatible\ S.1\ T.1) \Rightarrow\\ \quad ((Has\_prop\ S\ P) \wedge (Has\_prop\ T\ P)) \Rightarrow\\ \qquad\qquad (Has\_prop\ (Or\ S\ T)\ P)\end{array}$$

Proof. By expanding definitions and using the introduction tactic, we arrive at the following context:

$$\begin{array}{l}S, T\ :\ Schema\\ P\ :\ Predicate\\ H\ :\ (Well\_formed\ S) \wedge (Well\_formed\ T)\\ H_1\ :\ type\_compatible\ S.1\ T.1\\ H_2\ :\ (Has\_prop\ S\ P) \wedge (Has\_prop\ T\ P)\\ b\ :\ Binding\\ H_3\ :\ restricts\_to\_model\ b\ (Or\ S\ T)\\ ?\ :\ P\ b\end{array}$$

Using Theorem 20 we can conclude:

$$H_4\ :\ (restricts\_to\_model\ b\ S) \vee (restricts\_to\_model\ b\ T)$$

We apply or-elimination to hypothesis H_4 and verify that the goal is satisfied in each case. □

5.1.6 Implication

We encode the implication operation on schemas:

Definition* 28 *Imply.

$$\begin{array}{l}Imply \stackrel{def}{=} \lambda S, S' : Schema.\\ \qquad (join\ S.sig\ S'.sig,\ \ \lambda b : Binding.\ (S.pred\ b) \rightarrow (S'.pred\ b))\\ : Schema \longrightarrow Schema \longrightarrow Schema\end{array}$$

5.1.7 Theorems about implication

We show that schema implicationpreserves the well-formedness property:

Theorem 23 Imply_preserves_well_formedness.

$$\forall\, S, T : Schema.\\ \quad (Well\_formed\ S) \wedge (Well\_formed\ T) \wedge (type\_compatible\ S.\mathbf{1}\ T.\mathbf{1}) \Rightarrow\\ \quad\quad Well\_formed\ (Imply\ S\ T)$$

Proof. See Proof 8 in Appendix A.

We prove the following simple result about *Imply* and the *model* relationship:

Theorem 24 Imply_model_char.

$$\forall\, S, T : Schema.\ \forall\, b : Binding.\\ \quad ((models\ S\ b) \Rightarrow (models\ T\ b)) \iff (models\ (Imply\ S\ T)\ b)$$

Proof. This follows easily from the definition of *Imply*. □

The analogous statement for restricting models is false. Given two schemas S and T and a binding b, the hypothesis

$$H\ :\ (restricts\_to\_model\ S\ b) \Rightarrow (restricts\_to\_model\ T\ b)$$

is not sufficient to allow us to conclude that b is a restricting model of *Imply* $S\ T$. The reason is that any binding which is not a restricting model of S will (vacuously) fit the hypothesis H. Such a binding will not, in general, be capable of being restricted to the signature of *Imply* $S\ T$. If we only consider bindings which are large enough to be restricted to this signature, we can prove the following result:

***Theorem* 25 *Imply_restricting_model_intro*.**

$$\begin{aligned}
&\forall\, S, T : Schema.\ \forall\, b : Binding.\\
&\quad (Down_closed\ S) \wedge (Up_closed\ T) \Rightarrow\\
&\quad (succeeds\ (restrict\ b\ (Imply\ S\ T).1)) \Rightarrow\\
&\qquad ((restricts_to_model\ S\ b) \Rightarrow (restricts_to_model\ T\ b)) \Rightarrow\\
&\qquad (restricts_to_model\ (Imply\ S\ T)\ b)
\end{aligned}$$

Proof. See Proof 9 in Appendix A.

We also prove an elimination result:

***Theorem* 26 *Imply_restricting_model_elim*.**

$$\begin{aligned}
&\forall\, S, T : Schema.\ \forall\, b : Binding.\\
&\quad (Up_closed\ S) \wedge (Down_closed\ T) \Rightarrow\\
&\qquad (restricts_to_model\ (Imply\ S\ T)\ b) \Rightarrow\\
&\qquad ((restricts_to_model\ S\ b) \Rightarrow (restricts_to_model\ T\ b))
\end{aligned}$$

Proof. Omitted, because it is very similar to the proof of Theorem 25.

5.2 Schema negation

The ZRM explains the semantics of schema negation as follows:

> The negation $\neg S$ of a schema S has the same signature as S but its property is true in just those bindings where the property of S is not true.

Here is our encoding of schema negation:

***Definition* 29 *Not*.**

$$\begin{aligned}
&Not \stackrel{def}{=} \lambda S : Schema.\ (S.sig,\ \lambda b : Binding.\ \neg(S.pred\ b))\\
&: Schema \longrightarrow Schema
\end{aligned}$$

5.2.1 Theorems about schema negation

We show that *Not* preserves well-formedness. The *unique_idents* property is obviously preserved, since *Not S* has the same signature as *S*. Up-closure and down-closure are also preserved:

Theorem 27 Not_up_closed.

$$\forall S : Schema .\ Down\_closed\ S \Rightarrow Up\_closed\ (Not\ S)$$

Proof. See Proof 10 in Appendix A.

Theorem 28 Not_down_closed.

$$\forall S : Schema .\ Up\_closed\ S \Rightarrow Down\_closed\ (Not\ S)$$

Proof. Omitted, because it is very similar to the proof of Theorem 27.

The next two theorems are simple results about models of negated schemas which follow directly from the definition of *Not*.

Theorem 29 Not_model_property1.

$$\forall S : Schema .\ \forall b : Binding .$$
$$(models\ S\ b) \Rightarrow \neg(models\ (Not\ S)\ b)$$

Proof. Trivial.

The converse of Theorem 29 is equivalent to the statement $\forall P : Prop .\ \neg\neg P \Rightarrow P$ which is not provable in LEGO's intuitionistic logic. (We have used LEGO to verify that the two statements are equivalent.)

Theorem 30 Not_model_property2.

$$\forall S : Schema .\ \forall b : Binding .$$
$$\neg(models\ S\ b) \iff (models\ (Not\ S)\ b)$$

Proof. Trivial.

Next we show that a binding cannot model both a schema and its negation.

Theorem 31 Non_contradiction.

$$\forall S : Schema.\ \forall b : Binding.\ \neg((models\ S\ b) \wedge (models\ (Not\ S)\ b))$$

Proof. Trivial.

The next result is an intuitionistically provable version of the Law of the Excluded Middle.

Theorem 32 Not_model_cases.

$$\forall S : Schema.\ \forall b : Binding.\ \neg\neg(models\ (Or\ S\ (Not\ S))\ b)$$

Proof. By expanding the definitions of *Or*, *Not*, *models*, *et c.*, we transform the goal to:

$$\neg\neg((S.2\ b) \vee \neg(S.2\ b))$$

By expanding the definitions of $\vee$ and $\neg$, we can prove this goal. The technique is well-known so we omit it here.

Now we come to some results about restricting models. First we show that a restricting model of a schema S cannot be a restricting model of $Not\ S$.

Theorem 33 Not_restricting_model_property1.

$$\forall S : Schema.\ \forall b : Binding.$$
$$(restricts\_to\_model\ S\ b) \Rightarrow \neg(restricts\_to\_model\ (Not\ S)\ b)$$

Proof. See Proof 11 in Appendix A.

Next we show that if a binding is a restricting model of $Not\ S$ then it cannot be a restricting model of S.

Theorem 34 Not_restricting_model_property2.

$$\forall\, S : Schema.\ \forall\, b : Binding.$$
$$(restricts\_to\_model\ (Not\ S)\ b) \Rightarrow \neg(restricts\_to\_model\ S\ b)$$

Proof. Omitted, since it is very similar to the proof of Theorem 33.

The converse of Theorem 33 is false, as we can prove in LEGO.

Theorem 35 Not_restricting_model_property3.

$$\neg(\forall\, S : Schema.\ \forall\, b : Binding.$$
$$\neg(restricts\_to\_model\ (Not\ S)\ b) \Rightarrow restricts\_to\_model\ S\ b)$$

Proof. By expanding the definition of $\neg$ and doing an introduction we arrive at the following proof context:

$$H : \forall\, S : Schema.\ \forall\, b : Binding.$$
$$\neg(restricts\_to\_model\ (Not\ S)\ b) \Rightarrow restricts\_to\_model\ S\ b$$

$$?\ :\ absurd$$

Take any schema S and any binding b which is not capable of being restricted to the signature of S. An example might be $S \stackrel{def}{=} ([(x, nat\_ty)], trueProp)$ and the empty binding nil_bin. First we prove, by computation, that nil_bin cannot be restricted to the signature of S:

$$H_1 \stackrel{def}{=} \ldots : fails\ (restrict\ nil\_bin\ S.\mathbf{1})$$

We use Lemma 38 to show that nil_bin is not a restricting model of S. Hypothesis H then allows us to conclude:

$$H_1 \stackrel{def}{=} \ldots : restricts\_to\_model\ S\ nil\_bin$$

Lemma 38 allows us to deduce from H_1 that nil_bin can be successfully restricted to the signature of S. However, H_1 shows that nil_bin cannot be restricted to the signature of S. We use the LEGO library theorem $in\mathbf{1}\_not\_in\mathbf{2}$ to put these pieces together to prove $absurd$. ☐

To prove Theorem 35 we used a schema together with a binding which was not capable of being restricted to the signature of that schema. If we disallow such bindings, can we prove the converse of Theorem 33? We cannot, for the same reason that we could not prove the converse of Theorem 29: the statement we get is equivalent to $\forall P : Prop.\ \neg\neg P \Rightarrow P$.

Using a similar argument to that of Theorem 35 we prove that the converse of Theorem 34 is also false. However, in this case, if we restrict ourselves to bindings for which restriction is successful we get the following result:

Theorem 36 Not_restricting_model_property4.

$$\forall S : Schema.\ \forall b : Binding.\ (succeeds\ (restrict\ b\ S.\mathbf{1})) \Rightarrow \\ \neg(restricts\_to\_model\ S\ b) \Rightarrow restricts\_to\_model\ (Not\ S)\ b$$

Proof. See Proof 12 in Appendix A.

Finally, we prove an intuitionistic version of the law of the excluded middle for restricting models.

Theorem 37 Not_restricting_model_cases.

$$\forall S : Schema.\ \forall b : Binding.\ (succeeds\ (restrict\ b\ S.\mathbf{1})) \Rightarrow \\ \neg\neg(restricts\_to\_model\ (Or\ S\ (Not\ S))\ b)$$

Proof. The proof is similar to that of Theorem 32.

5.3 The hiding operations

In this section we discuss the three operations $\backslash$, $\forall$, and $\exists$, which all have in common the feature that they hide some of the components of their argument schemas.

5.3.1 Hiding

The hiding operator $\setminus$ is described as follows in Section 2.2.3 of the ZRM.

> If S is a schema, and $x_1, \ldots, x_n$ are components of S then
>
> $$S \setminus (x_1, \ldots, x_n)$$
>
> is a schema. Its components are the components of S, except for $x_1, \ldots, x_n$, and they have the same types as in S. The property of this schema is true under exactly those bindings that are restrictions of bindings that satisfy the property of S.

5.3.2 Encoding the hiding operator

The ZRM states that the hiding operator can be written in terms of the existential quantifier. For instance, if we have the following schema

$$\begin{array}{|l}
\multicolumn{1}{l}{S} \\ \hline
x, y, z : \mathbb{N} \\ \hline
x = y \land z = 0 \\ \hline
\end{array}$$

then the schema $T \mathrel{\widehat{=}} S \setminus (x, z)$ can be written as

$$\begin{array}{|l}
\multicolumn{1}{l}{T} \\ \hline
y : \mathbb{N} \\ \hline
\exists\, x : \mathbb{N} .\ \exists\, z : \mathbb{N} .\ x = y \land z = 0 \\ \hline
\end{array}$$

We shall use this as the basis of our encoding of the hiding operator.

First, we define an auxiliary function *hide_sig* : *Signature* $\longrightarrow$ *Signature* $\longrightarrow$ *Signature*. (See Appendix C.1 for the full definition of *hide_sig*.) When applied to two signatures s_1 and s_2, this function returns the signature formed by removing from s_2 all signature items which happen to be in s_1. We use this to compute the signatures of the schemas produced by the hiding operator.

Next, we define another auxiliary function *join_bin* which puts two bindings together to form a new binding. This uses a third function *remove_occurs* : *Signature* $\longrightarrow$ *Binding* $\longrightarrow$ *Binding* which, when applied to a signature s and a binding b, returns the binding formed by removing all components from b whose identifier occurs in s. (The full definition of *remove_occurs* is in Appendix C.1.) We use this function to remove duplicate identifiers when we combine two bindings:

Definition 30 join_bin.

$$\begin{aligned}
&\mathit{join_bin} \stackrel{\mathit{def}}{=} \boldsymbol{\lambda} a, b : \mathit{Binding}.\ \mathit{append}\ a\ (\mathit{remove_occurs}\ (\mathit{extract_sig}\ a)\ b) \\
&: \mathit{Binding} \longrightarrow \mathit{Binding} \longrightarrow \mathit{Binding}
\end{aligned}$$

Here is our encoding of the hiding operator:

Definition 31 Hide.

$$\begin{aligned}
&\mathit{Hide} \stackrel{\mathit{def}}{=} \boldsymbol{\lambda} \mathit{sig} : \mathit{Signature}.\ \boldsymbol{\lambda} S : \mathit{Schema}. \\
&\quad (\mathit{hide_sig}\ \mathit{sig}\ S.\mathbf{1}, \\
&\quad\ \boldsymbol{\lambda} b : \mathit{Binding}.\ \exists\, b' : \mathit{Binding}. \\
&\qquad (\mathit{is_true}\ (\mathit{matches}\ \mathit{sig}\ b')) \wedge (S.\mathbf{2}\ (\mathit{join_bin}\ b\ b'))) \\
&: \mathit{Signature} \longrightarrow \mathit{Schema} \longrightarrow \mathit{Schema}
\end{aligned}$$

We prove that the *Hide* operation preserves well-formedness:

Theorem 38 Hide_preserves_well_formedness.

$$\forall\, s : \mathit{Signature}.\ \forall\, S : \mathit{Schema}.\ (\mathit{Well\_formed}\ S) \Rightarrow (\mathit{Well\_formed}\ (\mathit{Hide}\ s\ S))$$

Proof. See Proof 13 in Appendix A

5.3.3 Universal and existential quantification

A schema may be formed by universally or existentially quantifying one schema over another. These operations are described in the ZRM as follows:

If D is a declaration, P is a predicate, and S is a schema, then

$$\forall D \mid P \bullet S$$

is a schema. The schema S must have as components all the variables introduced by D and they must have the same types. The signature of the result contains all the components of S except those introduced by D, and they have the same types as in S. The property of the result is derived as follows: for any binding z for the signature of the result, consider all the extensions z' of z to the signature of S. If every such extension z' which satisfies [...] the predicate P also satisfies the property of S, then the original binding satisfies the property of $\forall D \mid P \bullet S$,

The schema $\exists D \mid P \bullet S$ has the same signature as $\forall D \mid P \bullet S$, but its property is true under a binding z if at least one of the extensions of z simultaneously satisfies [...] the predicate P and the property of S.

(We have removed references to the constraints associated with declarations in Z because, in our reduced version of Z, declarations do not carry constraints.)

5.3.4 Encoding the quantifiers

We use the function *hide_sig* to construct the signatures of the schemas formed by the universal and existential quantifier.

The predicate produced by the universal quantifier is encoded as follows: a binding b satisfies the predicate produced by *All* S T provided that, for all bindings b' which match the hidden signature $S.1$, if *join_bin* b' b satisfies the predicate of S then it also satisfies the predicate of T.

Definition* 32 *All.

$$
\begin{aligned}
& All \stackrel{def}{=} \boldsymbol{\lambda} S, T : Schema. \\
& \quad (hide\_sig\ S.\mathbf{1}\ T.\mathbf{1}, \\
& \quad\ \boldsymbol{\lambda} b : Binding.\ \forall\, b' : Binding.\ (is\_true\ (matches\ S.\mathbf{1}\ b')) \Rightarrow \\
& \qquad\qquad (S.\mathbf{2}\ (join\_bin\ b'\ b)) \Rightarrow (T.\mathbf{2}\ (join\_bin\ b'\ b))) \\
& : Schema \rightarrow Schema \rightarrow Schema
\end{aligned}
$$

The existential quantifier is encoded similarly:

Definition* 33 *Exists.

$$
\begin{aligned}
& Exists \stackrel{def}{=} \boldsymbol{\lambda} S, T : Schema. \\
& \quad (hide\_sig\ S.\mathbf{1}\ T.\mathbf{1}, \\
& \quad\ \boldsymbol{\lambda} b : Binding.\ \exists\, b' : Binding.\ ((is\_true\ (matches\ S.\mathbf{1}\ b')) \wedge \\
& \qquad\qquad (S.\mathbf{2}\ (join\_bin\ b'\ b)) \wedge (T.\mathbf{2}\ (join\_bin\ b'\ b)))) \\
& : Schema \rightarrow Schema \rightarrow Schema
\end{aligned}
$$

We conjecture that *All* and *Exists* preserve well-formedness.

5.4 Schema inclusion

In the grammar shown in Figure 3–1 we see that the declaration part of a schema may contain schema references, that is, the names of previously defined schemas. In this section we shall look at how to encode this kind of declaration. We shall group this with the logical operations because, as we shall see, forming declarations in this way is logically related to schema conjunction.

First let us look at the semantics of this kind of declaration. To give an example, if we have defined a schema S,

S
$x : \mathbb{N}$
$x > 0$

then the name of this schema can be used in the declaration of another schema:

$$\begin{array}{|l}
\multicolumn{1}{l}{T} \\
\hline
S \\
y : \mathbb{N} \\
\hline
y = x \\
\hline
\end{array}$$

According to Section 3.4 of the ZRM,

> [A schema reference] introduces the components of the schema as variables, with the same types as they have in the schema, and constrains their values to satisfy their property.

When several basic declarations are combined, as in the above example, repeated occurrences of the same identifier are merged in the resulting signature. Any identifier which appears more than once must be given the same type in every basic declaration in which it occurs.

The schema T is logically equivalent to the following schema:

$$\begin{array}{|l}
\multicolumn{1}{l}{T} \\
\hline
x, y : \mathbb{N} \\
\hline
x > 0 \land y = x \\
\hline
\end{array}$$

5.4.1 Encoding schema inclusion

The use of schema references as declarations can be thought of as a way of defining new schemas which "include" previously defined schemas within their definition. We shall represent such declarations by an operation *Include* which puts together a *Schema*, a *Signature*, and a *Predicate* to form a new *Schema*.

Definition 34 Include.

$$\begin{array}{l} \mathit{Include} \stackrel{\mathit{def}}{=} \boldsymbol{\lambda} S\!:\!\mathit{Schema}.\ \boldsymbol{\lambda} \mathit{sig}\!:\!\mathit{Signature}.\ \boldsymbol{\lambda} P\!:\!\mathit{Predicate}. \\ \qquad\qquad\qquad\qquad (\mathit{join}\ S.1\ \mathit{sig},\ \boldsymbol{\lambda} b\!:\!\mathit{Binding}.\ (S.2\ b) \wedge (P\ b)) \\ :\ \mathit{Schema} \rightarrow \mathit{Signature} \rightarrow \mathit{Predicate} \rightarrow \mathit{Schema} \end{array}$$

This definition allows only one *Schema* to be included at a time. To include several schemas, we first conjoin them all with *And*.

5.4.2 Theorems about schema inclusion

First we show two ways in which *Include* can be defined in terms of *And*. The first result shows that including a schema S within a signature *sig* with a predicate P is the same as conjoining S with the schema (sig, P).

Theorem 39 Include_equals_And1.

$$\begin{array}{l} \forall\, S\!:\!\mathit{Schema}.\ \forall\, \mathit{sig}\!:\!\mathit{Signature}.\ \forall\, P\!:\!\mathit{Predicate}. \\ \qquad\qquad\qquad \mathit{Include}\ S\ \mathit{sig}\ P = \mathit{And}\ S\ (\mathit{sig}, P) \end{array}$$

Proof. By conversion.

The second result shows that, under certain conditions, *Include S sig P* is the same as conjoining S with the schema $(\mathit{join}\ S.1\ \mathit{sig}, P)$. This is more useful than Theorem 39 because it will usually be the case that the predicate P will refer to items in the signature of the included schema S. In such cases (sig, P) will not be a well-formed schema, but $(\mathit{join}\ S.1\ \mathit{sig}, P)$ might be.

Theorem 40 Include_equals_And2.

$$\begin{array}{l} \forall\, S\!:\!\mathit{Schema}.\ \forall\, \mathit{sig}\!:\!\mathit{Signature}.\ \forall\, P\!:\!\mathit{Predicate}. \\ (\mathit{unique\_idents}\ S.1) \wedge (\mathit{unique\_idents}\ \mathit{sig}) \wedge (\mathit{type\_compatible}\ S.1\ \mathit{sig}) \Rightarrow \\ \quad \mathit{Include}\ S\ \mathit{sig}\ P = \mathit{And}\ S\ (\mathit{join}\ S.1\ \mathit{sig}, P) \end{array}$$

Proof. See Proof 14 in Appendix A.

By using Theorem 40 to rewrite *Include* in terms of *And*, we can use our metatheorems about *And* to prove the following metatheorems about *Include*:

Theorem 41 Include_well_formed.

$$\begin{array}{l}\forall\, S\!:\!Schema.\ \forall\, Sig\!:\!Signature.\ \forall\, P\!:\!Predicate.\\ (Well\_formed\ S) \wedge (type\_compatible\ S.\mathbf{1}\ Sig) \Rightarrow\\ (Well\_formed\ (join\ S.\mathbf{1}\ Sig, P)) \Rightarrow (Well\_formed\ (Include\ S\ Sig\ P))\end{array}$$

Proof. This follows from Theorems 40 and 10. □

Theorem 42 Include_model_property.

$$\begin{array}{l}\forall\, S\!:\!Schema.\ \forall\, Sig\!:\!Signature.\ \forall\, P\!:\!Predicate.\ \forall\, b\!:\!Binding.\\ \quad ((models\ S\ b) \wedge (P\ b)) \Longleftrightarrow (models\ (Include\ S\ Sig\ P)\ b)\end{array}$$

Proof. The proof is the same as that of Theorem 11. □

Theorem 43 Include_restricting_model_property.

$$\begin{array}{l}\forall\, S\!:\!Schema.\ \forall\, Sig\!:\!Signature.\ \forall\, P\!:\!Predicate.\ \forall\, b\!:\!Binding.\\ (Well\_formed\ S) \wedge (Well\_formed\ (join\ S.\mathbf{1}\ Sig, P)) \wedge\\ (type\_compatible\ S.\mathbf{1}\ Sig) \wedge (unique\_idents\ (extract\_sig\ b)) \Rightarrow\\ \quad ((restricts\_to\_model\ S\ b) \wedge (restricts\_to\_model\ (join\ S.\mathbf{1}\ Sig, P)\ b)) \Longleftrightarrow\\ \quad (restricts\_to\_model\ (Include\ S\ Sig\ P)\ b)\end{array}$$

Proof. This follows from Theorems 40 and 13. □

5.5 Formal description of our translation

At this point we have presented all the UTT definitions needed to give our semantics of Z′. We now complete the description of the semantics by describing formally how to translate Z′ specifications into their representations in UTT.

UTTenv = Global $\times$ SchemaDict

Global = Vars $\times$ Axioms

SchemaDict = UTTIdent $\twoheadrightarrow\!\!\!\!\!\!\shortmid\;\;$ *Schema*

Vars = UTTIdent $\twoheadrightarrow\!\!\!\!\!\!\shortmid\;\;$ *Ztype*

Axioms = UTTIdent $\twoheadrightarrow\!\!\!\!\!\!\shortmid\;\;$ *Prop*

UTTIdent = allowed identifiers in UTT

Figure 5–2: Semantic objects

5.5.1 Definition of the semantic objects

First of all we must define some semantic objects to capture those aspects of the representation which are not internally represented by definitions in UTT. All the definitions are parameterised over a set of given types, G, obtained from the prelude of a Z′ specification.

The definitions of the semantic objects are shown in figure 5–2, where, as in Z, the symbol $\twoheadrightarrow\!\!\!\!\!\!\shortmid\;\;$ is used to represent "finite, partial functions" in our informal meta-language.

5.5.2 Syntax annotations

To simplify the presentation of the translation, we shall add some annotations to the syntax of Z′ . Though we do not give the algorithms for doing so, these annotations can all be automatically computed for well-typed Z′ specifications.

- All axiomatic descriptions are labelled with new, unique identifiers. In other words, the first clause in the definition of the syntax of MAINSPEC (figure 3–2) becomes:

  `let` WORD : SCHEMA `end`

  These identifiers will be used as the names of the assumptions in UTT by which axiomatic descriptions are represented.

- Terms are annotated with types as shown below:

TERM	::=	$\text{IDENT}_{\text{TYPE}}$	(type of the identifier)
	|	$\emptyset$ [TYPE]	
	|	$\{\text{TERM}_1,\ldots,\text{TERM}_n\}_{\text{TYPE}}$	(base type of the set)
	|	(TERM,TERM)	
	|	$(\text{TERM (TERM)})_{\text{TYPE}}$	(argument type of first term)

  These annotations will be used to compute the correct *Ztype* arguments to give to *lookup*, *Equal* and *Apply*.

- In schema expressions formed by hiding, the identifiers to be hidden are annotated with their expected types. We shall simply re-use the phrase class DECL instead of introducing more syntax to represent these annotated identifiers. The syntax of these schema expressions becomes:
  SEXP \ (DECL)

- PREDs formed using $=$ and $\in$ are annotated with types:

PRED	::=	TERM $=_{\text{TYPE}}$ TERM
	|	TERM $\in_{\text{TYPE}}$ TERM

5.5.3 Translating the annotated syntax

We now show how to translate an annotated Z′ specification to a UTT environment. We shall assume that the prelude of the specification has been translated, so that the definitions of *GivenType* and *Ztype* have already been created. The rules in figures 5–3, 5–4, 5–5 and 5–6 show how to translate the main specification to obtain a UTTenv. The symbols $\bigcup$ and $\emptyset$ are generalised to mean, respectively, the component-wise union of two tuples of sets, and a tuple of empty sets.

For the sake of readability, we have not used subscripts to distinguish the semantic functions on different phrase classes, except for the cases where more than one semantic function happen to operate on the same phrase class.

Main specifications $[\![\ ]\!] : \mathrm{MAINSPEC} \rightarrow \mathrm{UTTenv} \rightarrow \mathrm{UTTenv}$

$$[\![\texttt{let}\ \mathrm{word}:\mathrm{schema}\ \texttt{end}, E]\!] = E \cup ([\![\mathrm{word}, \mathrm{schema}]\!]_{\mathrm{top}}, \emptyset)$$

$$[\![\texttt{let}\ \mathrm{word} = \mathrm{sexp}, E]\!] = E \cup ((\emptyset, \emptyset), \{\mathrm{word} \mapsto [\![\mathrm{sexp}, E]\!]\})$$

$$[\![\mathrm{mainspec}_1\ \texttt{in}\ \mathrm{mainspec}_2, E]\!] = [\![\mathrm{mainspec}_2, [\![\mathrm{mainspec}_1, E]\!]]\!]$$

Axiomatic descriptions $[\![\ ]\!]_{\mathrm{top}} : \mathrm{WORD} \rightarrow \mathrm{SCHEMA} \rightarrow \mathrm{Global}$

$$[\![\mathrm{word}, \mathrm{decl} \mid \mathrm{pred}]\!]_{\mathrm{top}} = ([\![\mathrm{decl}]\!]_{\mathrm{top}}, [\![\mathrm{word}, \mathrm{pred}]\!]_{\mathrm{top}})$$

Top declarations $[\![\ ]\!]_{\mathrm{top}} : \mathrm{DECL} \rightarrow \mathrm{Vars}$

$$[\![\mathrm{ident}:\mathrm{type}]\!]_{\mathrm{top}} = \{\mathrm{ident} \mapsto \mathit{Typ}\ [\![\mathrm{type}]\!]\})$$

$$[\![\mathrm{decl}_1\ ;\ \ \mathrm{decl}_2]\!]_{\mathrm{top}} = [\![\mathrm{decl}_1]\!]_{\mathrm{top}} \cup [\![\mathrm{decl}_2]\!]_{\mathrm{top}}$$

Top-level predicates $[\![\ ]\!]_{\mathrm{top}} : \mathrm{WORD} \rightarrow \mathrm{PRED} \rightarrow \mathrm{Axioms}$

$[\![\ ]\!]_{\mathrm{top}} : \mathrm{PRED} \rightarrow \mathit{Prop}$

$$[\![\mathrm{word}, \mathrm{pred}]\!]_{\mathrm{top}} = \{\mathrm{word} \mapsto [\![\mathrm{pred}]\!]_{\mathrm{top}}\}$$

$$[\![\mathrm{term}_1\ =_{\mathrm{type}}\ \mathrm{term}_2]\!]_{\mathrm{top}} = \mathit{IS\_TRUE}\ (\mathit{EQUAL}\ [\![\mathrm{type}]\!]\ [\![\mathrm{term}_1]\!]_{\mathrm{top}}\ [\![\mathrm{term}_2]\!]_{\mathrm{top}})$$

$$[\![\mathrm{term}_1 \in_{\mathrm{type}} \mathrm{term}_2]\!]_{\mathrm{top}} = \mathit{IS\_TRUE}\ (\mathit{IN}\ (\mathit{Equal}\ [\![\mathrm{type}]\!])\ [\![\mathrm{term}_1]\!]_{\mathrm{top}}\ [\![\mathrm{term}_2]\!]_{\mathrm{top}})$$

$$[\![\mathit{true}]\!]_{\mathrm{top}} = \mathit{trueProp}$$

$$[\![\mathit{false}]\!]_{\mathrm{top}} = \mathit{absurd}$$

$$[\![\neg\ \mathrm{pred}]\!]_{\mathrm{top}} = \ {}^{\sim}[\![\mathrm{pred}]\!]_{\mathrm{top}}$$

$$[\![\mathrm{pred}_1 \wedge \mathrm{pred}_2]\!]_{\mathrm{top}} = [\![\mathrm{pred}_1]\!]_{\mathrm{top}} \wedge [\![\mathrm{pred}_2]\!]_{\mathrm{top}}$$

$$[\![\mathrm{pred}_1 \vee \mathrm{pred}_2]\!]_{\mathrm{top}} = [\![\mathrm{pred}_1]\!]_{\mathrm{top}} \vee [\![\mathrm{pred}_2]\!]_{\mathrm{top}}$$

$$[\![\mathrm{pred}_1 \Rightarrow \mathrm{pred}_2]\!]_{\mathrm{top}} = [\![\mathrm{pred}_1]\!]_{\mathrm{top}} \Rightarrow [\![\mathrm{pred}_2]\!]_{\mathrm{top}}$$

$$[\![\exists\, \mathrm{ident} : \mathrm{type} \bullet \mathrm{pred}]\!]_{\mathrm{top}} = \exists\, \mathrm{ident}:\mathit{Typ}\ [\![\mathrm{type}]\!].\ [\![\mathrm{pred}]\!]_{\mathrm{top}}$$

$$[\![\forall\, \mathrm{ident} : \mathrm{type} \bullet \mathrm{pred}]\!]_{\mathrm{top}} = \forall\, \mathrm{ident}:\mathit{Typ}\ [\![\mathrm{type}]\!].\ [\![\mathrm{pred}]\!]_{\mathrm{top}}$$

Figure 5–3: Semantic rules (part 1)

Top level terms $[\![\ ]\!]_{\text{top}} : \text{TERM} \to \textit{small\_item}$

$[\![(\text{ident} : \text{type})]\!]_{\text{top}} = \textit{in}\mathbf{2}\ ([\![\text{type}]\!], \text{ident})$

$[\![\{\text{term}_1, \ldots, \text{term}_n\}_{\text{type}}]\!]_{\text{top}} =$

$\quad \textit{ADD\_ONE}\ (\textit{Equal}\ [\![\text{type}]\!])\ [\![\text{term}_1]\!]_{\text{top}}$

$\quad\quad (\ldots (\textit{ADD\_ONE}\ (\textit{Equal}\ [\![\text{type}]\!])\ [\![\text{term}_1]\!]_{\text{top}}\ (\textit{NULL}\ [\![\text{type}]\!])))$

$[\![\emptyset\ [\text{type}]]\!]_{\text{top}} = \textit{NULL}\ [\![\text{type}]\!]$

$[\![(\text{term}_1, \text{term}_2)]\!]_{\text{top}} = \textit{PAIR}\ [\![\text{term}_1]\!]_{\text{top}}\ [\![\text{term}_2]\!]_{\text{top}}$

$[\![\text{term}_1(\text{term}_2)_{\text{type}}]\!]_{\text{top}} =$

$\quad\quad \textit{APPLY}\ (\textit{Equal}\ [\![\text{type}]\!])\ [\![\text{term}_1]\!]_{\text{top}}\ [\![\text{term}_2]\!]_{\text{top}}$

Schema expressions $[\![\ ]\!] : \text{SEXP} \to \text{UTTenv} \to \text{Schema}$

$[\![\texttt{schema}\ \text{schema}\ \texttt{end}, E]\!] = [\![\text{schema}]\!]$

$[\![\text{sdes}, E]\!] = [\![\text{sdes}]\!], E_{\text{sdes}}$

$[\![\neg\text{sexp}, E]\!] = \textit{Not\_schema}\ [\![\text{sexp}, E]\!]$

$[\![\text{sexp}_1 \wedge \text{sexp}_2, E]\!] = \textit{And}\ [\![\text{sexp}_1, E]\!]\ [\![\text{sexp}_2, E]\!]$

$[\![\text{sexp}_1 \vee \text{sexp}_2, E]\!] = \textit{Or}\ [\![\text{sexp}_1, E]\!]\ [\![\text{sexp}_2, E]\!]$

$[\![\text{sexp}_1 \Rightarrow \text{sexp}_2, E]\!] = \textit{Imply}\ [\![\text{sexp}_1, E]\!]\ [\![\text{sexp}_2, E]\!]$

$[\![\text{sexp} \setminus \text{decl}, E]\!] = \textit{Hide}\ [\![\textit{decl}]\!]\ [\![\textit{sexp}, E]\!]$

$[\![\exists\ \text{schema} \bullet \text{sexp}, E]\!] = \textit{Exists}\ [\![\text{schema}]\!]\ [\![\text{sexp}, E]\!]$

$[\![\forall\ \text{schema} \bullet \text{sexp}, E]\!] = \textit{All}\ [\![\text{schema}]\!]\ [\![\text{sexp}, E]\!]$

$[\![\texttt{include}\ \text{sdes decl pred}, E]\!] =$

$\quad \textit{Include\_schema}$

$\quad\quad [\![\text{decl}]\!]\ [\![\text{pred}, \textit{join}\ [\![\text{decl}]\!]\ [\![\text{sdes}, E]\!]_{\text{SDES}}.\mathbf{1}]\!]\ [\![\text{sdes}, E]\!]_{\text{SDES}}$

Schema designators $[\![\ ]\!]_{\text{SDES}} : \text{SDES} \to \text{UTTenv} \to \textit{Schema}$

$[\![\text{word}, E]\!]_{\text{SDES}} = E.\mathbf{2}\ (\text{word})$

$[\![\text{sdes}', E]\!]_{\text{SDES}} = \textit{Prime}\ ([\![\text{sdes}]\!]_{\text{SDES}})$

Schema bodies $[\![\ ]\!] : \text{SCHEMA} \to \textit{Schema}$

$[\![\text{decl} \mid \text{pred}]\!] = ([\![\text{decl}]\!], [\![\text{pred}, [\![\text{decl}]\!]]\!])$

Figure 5–4: Semantic rules (part 2)

Declarations $[\![\ ]\!]$: DECL $\rightarrow$ *Signature*

$[\![\text{ident} : \text{type}]\!] = [([\![\text{ident}]\!], [\![\text{type}]\!])]$

$[\![\text{decl}_1;\ \text{decl}_2]\!] = \mathit{append}\ [\![\text{decl}_1]\!]\ [\![\text{decl}_2]\!]$

Predicates $[\![\ ]\!]$: PRED $\rightarrow$ *Signature* $\rightarrow$ (*Binding* $\rightarrow$ *Prop*)

$[\![\text{pred}, \mathit{sig}]\!] = \lambda b : \mathit{Binding}.\ [\![\text{pred}, \mathit{sig}]\!]_{\text{body}}$

$[\![\text{term}_1 \ =_{\text{type}} \ \text{term}_2, \mathit{sig}]\!]_{\text{body}} =$

$\quad \mathit{IS\_TRUE}\ (\mathit{EQUAL}\ [\![\text{type}]\!]\ [\![\text{term}_1, \mathit{sig}]\!]\ [\![\text{term}_2, \mathit{sig}]\!])$

$[\![\text{term}_1 \in_{\text{type}} \text{term}_2, \mathit{sig}]\!]_{\text{body}} =$

$\quad \mathit{IS\_TRUE}\ (\mathit{IN}\ (\mathit{Equal}\ [\![\text{type}]\!])\ [\![\text{term}_1, \mathit{sig}]\!]\ [\![\text{term}_2, \mathit{sig}]\!])$

$[\![\mathit{true}, \mathit{sig}]\!]_{\text{body}} = \mathit{trueProp}$

$[\![\mathit{false}, \mathit{sig}]\!]_{\text{body}} = \mathit{absurd}$

$[\![\neg\ \text{pred}, \mathit{sig}]\!]_{\text{body}} = \ {}^{\sim}[\![\text{pred}, \mathit{sig}]\!]_{\text{body}}$

$[\![\text{pred}_1 \wedge \text{pred}_2, \mathit{sig}]\!]_{\text{body}} = [\![\text{pred}_1, \mathit{sig}]\!]_{\text{body}} \wedge [\![\text{pred}_2, \mathit{sig}]\!]_{\text{body}}$

$[\![\text{pred}_1 \vee \text{pred}_2, \mathit{sig}]\!]_{\text{body}} = [\![\text{pred}_1, \mathit{sig}]\!]_{\text{body}} \vee [\![\text{pred}_2, \mathit{sig}]\!]_{\text{body}}$

$[\![\text{pred}_1 \Rightarrow \text{pred}_2, \mathit{sig}]\!]_{\text{body}} = [\![\text{pred}_1, \mathit{sig}]\!]_{\text{body}} \Rightarrow [\![\text{pred}_2, \mathit{sig}]\!]_{\text{body}}$

$[\![\exists\, \text{ident} : \text{type} \bullet \text{pred}, \mathit{sig}]\!]_{\text{body}} =$

$\quad \exists\, \text{ident} : \mathit{Typ}\ [\![\text{type}]\!].\ [\![\text{pred}, \mathit{sig} - [\![\text{ident}]\!]]\!]_{\text{body}}$

$[\![\forall\, \text{ident} : \text{type} \bullet \text{pred}, \mathit{sig}]\!]_{\text{body}} =$

$\quad \forall\, \text{ident} : \mathit{Typ}\ [\![\text{type}]\!].\ [\![\text{pred}, \mathit{sig} - [\![\text{ident}]\!]]\!]_{\text{body}}$

Identifiers $[\![\ ]\!]$: IDENT $\rightarrow$ *Ident*

This can be any function giving an isomorphism between the identifiers of a given Z′ specification and some finite subset of the UTT type *Ident* (= *nat*).

Figure 5–5: Semantic rules (part 3)

Terms $[\![\ ]\!] : \text{TERM} \rightarrow \mathit{Signature} \rightarrow \mathit{small\_item}$

$[\![\text{ident} : \text{type}, \mathit{sig}]\!] =$

$\quad (\mathit{lookup}\ ([\![\text{ident}]\!], [\![\text{type}]\!])\ b) \quad [\![\text{ident}]\!] \in \mathit{sig}$

$[\![\text{ident} : \text{type}, \mathit{sig}]\!] = \mathit{in}\mathbf{2}([\![\text{type}]\!], \text{ident}) \quad [\![\text{ident}]\!] \notin \mathit{sig}$

$[\![\{\text{term}_1, \ldots, \text{term}_n\}_{\text{type}}, \mathit{sig}]\!] =$

$\mathit{ADD\_ONE}\ (\mathit{Equal}\ [\![\text{type}]\!])\ [\![\text{term}_1, \mathit{sig}]\!]$

$\quad (\ldots (\mathit{ADD\_ONE}\ (\mathit{Equal}\ [\![\text{type}]\!])\ [\![\text{term}_1, \mathit{sig}]\!]\ (\mathit{NULL}\ [\![\text{type}]\!])))$

$[\![\emptyset\ [\text{type}], \mathit{sig}]\!] = \mathit{NULL}\ [\![\text{type}]\!]$

$[\![(\text{term}_1, \text{term}_2), \mathit{sig}]\!] = \mathit{PAIR}\ [\![\text{term}_1, \mathit{sig}]\!]\ [\![\text{term}_2, \mathit{sig}]\!]$

$[\![\text{term}_1(\text{term}_2)_{\text{type}}]\!] =$

$\quad \mathit{APPLY}\ (\mathit{Equal}\ [\![\text{type}]\!])\ [\![\text{term}_1, \mathit{sig}]\!]\ [\![\text{term}_2, \mathit{sig}]\!]$

Types $[\![\ ]\!] : \text{TYPE} \rightarrow \mathit{Ztype}$

$[\![\mathbb{N}]\!] = \mathit{nat\_ty}$

$[\![\mathbb{B}]\!] = \mathit{bool\_ty}$

$[\![\text{ident}]\!] = \mathit{given\_ty}\ \text{ident}$

$[\![\mathbb{F}\ \text{type}]\!] = \mathit{finset\_ty}\ [\![\mathit{type}]\!]$

$[\![\text{type}_1 \times \text{type}_2]\!] = \mathit{prod\_ty}\ [\![\text{type}_1]\!]\ [\![\text{type}_1]\!]$

Figure 5–6: Semantic rules (part 4)

5.6 Conclusion

We have encoded several of the logical operations on schemas provided by the Z notation. This allowed us, finally, to give a complete, formal definition of our semantics for Z' .

We used UTT to prove some meta-theoretical results about the relationship between the logical operations and our different notions of model for Z schemas. In chapter 7, we shall see how these theorems enable us to reason about Z specifications in a modular fashion.

Chapter 6

Encoding Z: Specifying operations

All the schemas that we have looked at so far have contained only undecorated identifiers in their signatures. Such schemas are conventionally understood as specifying the state space of an abstract datatype. For instance, the following schema specifies a state space containing two natural numbers, x and y:

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{Sch}} \\
\hline
x, y : \mathbb{N} \\
\hline
x > y \\
\hline
\end{array}$$

Operations map one state to another and can take input and produce output. They are specified by relationships between input and output variables and pairs of states representing the states before and after the execution of the operation. In Z, an operation upon the state space described by a schema S is specified by a schema which contains two copies of S, one of which has had all of its identifiers decorated with a $'$, and which represents the state after the operation is executed. In addition, the schema which specifies the operation may contain identifiers decorated with the symbols ? or !, representing the inputs and outputs of the operation. The notation S' is used to represent the operation of decorating all the identifiers in a schema S with a $'$. As an example, here is a schema which specifies an operation on the state space of the schema *Sch*. The operation takes as input a value *inc* and increments both x and y by this value.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{Op}} \\
\hline
Sch \\
Sch' \\
inc? : \mathbb{N} \\
\hline
x' = x + inc? \\
y' = y + inc? \\
\hline
\end{array}$$

In this chapter we shall look at a group of schema forming operations which relate to the specification of operations. We shall describe these operations, show how they may be encoded in type theory, and prove some theorems about the encoded operations.

6.1 Schema decoration

The operation of schema decoration was applied to the schema *Sch* in the definition of the schema *Op*. The meaning of this operation is described in Section 2.2.2 of the ZRM as follows:

> If S is a schema, then S' is the same as S, except that all the component names have been suffixed with the decoration $'$. The signature of S' contains a component x' for each component x of S, and the type of x' in S' is the same as the type of x in S.
>
> From a binding z for this new signature, a binding z_0 for the signature of S can be derived. In z_0, each component of S is given the value that x' takes in z, so that $z_0.x = z.x'$. The property of S' is true under z exactly if the property of S is true under the derived binding z_0.

Though it is possible to decorate schemas with decorations other than $'$, only the decoration with $'$ is meaningful for the purposes of specifying operations, so we shall look at this type of decoration only. There is no technical difficulty in extending our encoding to all other types of decoration.

6.1.1 Encoding schema decoration

We shall encode schema decoration as a function $\mathit{Prime} : \mathit{Schema} \rightarrow \mathit{Schema}$. It is easy to define a function to compute the signature of the new schema. First we define a function $\mathit{decorate\_item} : \mathit{Decor\_char} \rightarrow \mathit{sig\_item} \rightarrow \mathit{sig\_item}$ which decorates a single signature item. This simply appends the given decoration character to the decoration of the identifier of the given signature item. We use this to define an operation $\mathit{prime\_sig} : \mathit{Signature} \rightarrow \mathit{Signature}$ which decorates all the identifiers in a signature with the decoration character pr. We prove that $\mathit{prime\_sig}$ preserves the property $\mathit{unique\_idents}$.

Lemma 1 prime_sig_preserves_unique_idents.

$$\forall\, \mathit{Sig} : \mathit{Signature} \bullet (\mathit{unique\_idents}\ \mathit{Sig}) \Rightarrow (\mathit{unique\_idents}\ (\mathit{prime\_sig}\ \mathit{sig}))$$

Proof. By induction on Sig. The details are straightforward. □

We cannot compute the predicate of the primed schema in such a direct fashion. This is because our encoding of schema predicates does not provide us with access to the syntactic structure of these predicates. Schema predicates are represented as terms of type $\mathit{Binding} \rightarrow \mathit{Prop}$. The only way in which we can compute with such a term is to apply it to a $\mathit{Binding}$. This means that the only way in which we can modify a schema predicate is by modifying the *Binding*s to which it is applied.

To define the new predicate we shall take advantage of the semantic property of the Prime operation that was quoted above. The property of a primed schema holds true of a binding b, if and only if the property of the original schema holds true of the binding formed from b by removing a prime from the decoration of each identifier in b. We shall use this as the basis of our definition of the new predicate.

First we define a function $\mathit{primed\_part} : \mathit{Binding} \rightarrow \mathit{Binding}$ which forms a new binding from all of the primed components in a given binding. Next, we define a function $\mathit{post\_bin} : \mathit{Binding} \rightarrow \mathit{Binding}$ which, when applied to a binding

b, forms a new binding by stripping the final prime from the decorations of all the components of *primed_part b*. We prove that *post_bin* produces a binding which matches a signature s if it is applied to a binding which matches the signature *Prime s*:

Lemma 2 post_bin_lemma.

$$\forall\, S : Signature.\ \forall\, b : Binding.\\ (is\_true\ (matches\ (prime\_sig\ S)\ b)) \Rightarrow (is\_true\ (matches\ S\ (post\_bin\ b)))$$

The converse of this result is not true, since the *post_bin* operation discards components which are not decorated with a prime.

Now we can define the Prime operation on schemas:

Definition 35 Prime.

$$Prime \stackrel{def}{=} \lambda S : Schema.\ (prime\_sig\ S.\mathbf{1},\ \lambda b : Binding.\ S.\mathbf{2}\ (post\_bin\ b))\\ : Schema \longrightarrow Schema$$

6.1.2 Theorems about schema decoration

First we show that the Prime operation preserves well-formedness. We have already shown that the *unique_idents* property is preserved (Lemma 1), so now we prove that down-closure and up-closure are also retained.

Theorem 44 Prime_down_closed.

$$\forall\, S : Schema.\ Down\_closed\ S \Rightarrow Down\_closed\ (Prime\ S)$$

Proof. See Proof 15 of Appendix A.

Theorem 45 Prime_up_closed.

$$\forall\, S : Schema.\ Up\_closed\ S \Rightarrow Up\_closed\ (Prime\ S)$$

Proof. Omitted, since this proof is similar to that of Theorem 44

Next we prove two theorems which characterise the behaviour of the *Prime* operation. The first relates to the *model* relationship and the second is the equivalent result for the *restricts_to_model* relationship.

Theorem 46 Prime_model_char.

$$\forall\, S : Schema \bullet\ \forall\, b : Binding \bullet\ models\ S\ (post\_bin\ b) \iff models\ (Prime\ S)\ b$$

Proof. This is a trivial consequence of the definition of *Prime*.

Theorem 47 Prime_restricting_model_char.

$$\begin{array}{l}\forall\, S : Schema \bullet\ \forall\, b : Binding \bullet \\ \quad restricts\_to\_model\ S\ (post\_bin\ b) \iff restricts\_to\_model\ (Prime\ S)\ b\end{array}$$

Proof. See Proof 16 of Appendix A.

6.2 The Δ convention

The Δ convention is the basis for specifying state-changing operations in Z. When applied to a schema S, Δ produces a new schema containing S and S'. For example, the schema *Op* could equally well have been defined as:

$$\begin{array}{|l}
\multicolumn{1}{l}{Op} \\ \hline
\Delta Sch \\
inc? : \mathbb{N} \\ \hline
x' = x + inc? \\
y' = y + inc? \\ \hline
\end{array}$$

6.2.1 Encoding Δ

We encode Δ in terms of *And* and *Prime*:

Definition 36 Delta.

$$Delta \stackrel{def}{=} \lambda S\!:\!Schema.\, And\ S\ (Prime\ S)$$
$$:\ Schema \rightarrow Schema$$

6.2.2 Theorems about Delta

We would like to prove that *Delta* preserves well-formedness. The simplest way to prove this would be to use the facts that *And* and *Prime* preserve well-formedness, since *Delta* is defined in terms of these operations. However, it turns out that in order to use the result about *And* (Theorem 10) we need to place an extra syntactic condition on schemas. Theorem 10 applies only to schemas whose signatures are type compatible. The *prime_sig* operation does not always produce a signature which is type compatible with the signature to which it is applied. If a signature s contains both primed and unprimed versions of the same identifier, and these happen to be paired with unequal *Ztype*s, then s will not be type compatible with *prime_sig s*. We must exclude such signatures if we want to guarantee type compatibility. To do so we define a predicate *static_sig* : *Signature* $\rightarrow$ *Prop* which is true of a signature s if and only if no identifier occurs both primed and unprimed in s.

Lemma 3 static_sig_gives_type_compatibility.

$$\forall\, s\!:\!Signature.\ (unique\_idents\ s) \Rightarrow$$
$$(static\_sig\ s) \Rightarrow (type\_compatible\ s\ (prime\_sig\ s))$$

We prove that *Delta* preserves well-formedness when applied to schemas whose signatures satisfy the *static_sig* property:

Theorem 48 Delta_preserves_well_formedness.

$$\forall\, S\!:\!Schema.\ (static\_sig\ S.1) \Rightarrow (Well\_formed\ S) \Rightarrow (Well\_formed\ (Delta\ S))$$

Proof. This follows easily from Lemma 3 and the facts that *And* and *Prime* preserve well-formedness (Theorems 10, 44 and 45, and Lemma 1.) □

Because of the way that *Delta* is defined, we can use the theorems about *And* and *Prime* to reason in a modular fashion about schemas created by *Delta*.

6.3 Binding formation (θ)

Before we go on to discuss the next schema-forming operation, Ξ, we shall say something about θ-expressions (ZRM, p 60) because these are used in the definition of Ξ in the ZRM. We can think of the expression θS as denoting an operation which captures the portion (if it exists) of a given binding that matches the signature of a schema S. It is similar to the *restrict* operation that we have defined. However, θ differs from *restrict* in the way that it behaves when applied to schemas decorated with a ′. When applied to such a schema, θ captures the matching portion of a binding and then strips off the ′ from every identifier before returning the result. Hence the bindings returned by $\theta S'$ have the same signature as those returned by θS. In the case of S', θ behaves like *restrict* followed by *post_bin*.

In our encoding we have no way to distinguish between schemas formed by the *Prime* operation, and other schemas, so we cannot encode the θ operation directly. Instead we shall have to use the functions *restrict* or *restrict* followed by *post_bin*, as appropriate. We believe that being forced to separate the uses of θ into these two operations gives us a more perspicuous account of Z. We preserve the intended function of θ but we avoid such anomalies as the fact that θ is not preserved by schema renaming: if we define $T = S'$, where S is some schema, then θT produces bindings that are different from those produced by θS, because the primes are not stripped from the bindings produced by θT.

6.4 The Ξ convention

The Ξ convention is used in Z for specifying operations which cause no change in state. For example, these might be access operations which simply look up the value of an identifier in the state. When applied to a schema S, Ξ produces a schema that is the same as $\Delta\ S$ with the additional predicate:

$$\theta\ S = \theta\ S'$$

6.4.1 Encoding Ξ

In encoding Ξ we are faced with the question of choosing an appropriate equality predicate for bindings. This is not so much a question about encoding schemas correctly as it is a decision about the correct theory of the core language of Z. We have chosen to use the standard inductive equality, *Eq*, but this is by no means the only possibility, nor do we believe that this choice is the final word on equality for bindings. Further experimentation may well reveal that inductive equality is too strong for this purpose. It is certainly stronger than the set-theoretic equality that is used by Z, because this equates extensionally equal functions, which the inductive equality does not.

We shall encode *Xi* in terms of *Delta* and *Include*. To represent θ we use *restrict* and *post_bin* as described in Section 6.3.

Definition 37 Xi.

$$\begin{aligned} &Xi \stackrel{def}{=} \boldsymbol{\lambda} S{:}Schema\boldsymbol{.}\ Include\ (Delta\ S)\ nil\_sig \\ &\qquad\qquad (\boldsymbol{\lambda} b{:}Binding\boldsymbol{.}\ restrict\ (post\_bin\ b)\ S.\mathbf{1} = restrict\ b\ S.\mathbf{1}) \\ &: Schema \longrightarrow Schema \end{aligned}$$

6.4.2 Theorems about Xi

We have proved that *Xi* preserves well-formedness under the same conditions required for *Delta* to do so:

Theorem 49 Xi_preserves_well_formedness.

$$\forall\, S\!:\!Schema.\ (static\_sig\ S.1) \Rightarrow (Well\_formed\ S) \Rightarrow (Well\_formed\ (Xi\ S))$$

Proof. See Proof 17 of Appendix A.

6.5 Precondition schemas

When applied to a schema which specifies an operation, the precondition operator produces a schema which describes the precondition of that operation. It is described in the ZRM as follows:

> If S is a schema, and $x'_1, \ldots, x'_m$ are the components of S that have the decoration $'$, and $y_1!, \ldots, y_n!$ are the components that have the decoration $!$, then the schema 'pre S' is the result of hiding these variables of S:
>
> $$S \setminus (x'_1, \ldots, x'_m, y_1!, \ldots, y_n!).$$

To encode the precondition operator, we first define a function *after_sig* : *Signature* $\longrightarrow$ *Signature* which, when applied to a signature s, forms a new signature by gathering all the signature items in s which have the decoration *prime* or *shriek* as their final decoration. The precondition operator is then defined as follows:

Definition 38 Pre.

$$Pre \stackrel{def}{=} \lambda S\!:\!Schema.\, Hide\ (after\_sig\ S.1)\ S$$
$$:\ Schema \longrightarrow Schema$$

We show that *Pre* preserves well-formedness:

Theorem 50 Pre_preserves_well_formedness.

$$\forall\, s\!:\!Signature.\ \forall\, S\!:\!Schema.\ (Well\_formed\ S) \Rightarrow (Well\_formed\ (Pre\ S))$$

Proof. First we show that *after_sig* produces a sub-signature of any signature to which it is applied. We also show that *after_sig* preserves the property *unique_idents*. We then use the fact that *Hide* preserves well-formedness (Theorem 38) to complete the proof. □

6.6 Sequential composition

Two schemas S and T which specify operations may be put together by sequential composition ($\fatsemi$) to form a new schema. The formal definition of this operation is given as follows in the ZRM:

> For the composition $S \fatsemi T$ to be defined, for each word x such that x' is a component of S and x itself is a component of T, the types of these two components must be the same. We call x a *matching state variable*. Also, the types of any other components they share (including inputs, outputs, and state variables that do not match) must be the same.
>
> The schema $S \fatsemi T$ has all the components of S and T, except for the components x of S and x' of T, where x is a matching state variable. If *State* is a schema containing just the matching state variables, then $S \fatsemi T$ is defined as
>
> $$\begin{array}{l} \exists\, State'' \bullet \\ \quad (\exists\, State' \bullet [S;\ State'' \mid \theta State' = \theta State'']) \land \\ \quad (\exists\, State \bullet [T;\ State'' \mid \theta State = \theta State'']). \end{array}$$

We can understand this schema as follows: for a binding b_1 to satisfy the predicate of $S \mathbin{;} T$, there must exist another binding b_2 which matches the hidden state variables of S and T. The binding b_2 may be thought of as the state in which S terminates and T begins. If b_2 is decorated with primes and combined with b_1, the result must satisfy the predicate of S. If b_2 is combined with b_1 the result must satisfy the predicate of T.

6.6.1 Encoding sequential composition

We can encode the formal definition of sequential composition directly, using the functions *restrict* and *post_bin* to represent the $\boldsymbol{\theta}$ operation as required. First, we define a function $matching\_vars : Signature \longrightarrow Signature \longrightarrow Signature$, which computes the matching state variables of two signatures. In other words, when applied to two signature s_1 and s_2, *matching_vars* returns a signature containing all the identifiers in s_2 which appear primed in s_1. Then, we define *Compose* as follows:

Definition* 39 *Compose.

$$
\begin{array}{l}
Compose \stackrel{def}{=} \boldsymbol{\lambda} S, T : Schema.\\
\quad [State = (matching\_vars\ S.\mathbf{1}\ T.\mathbf{1}, \boldsymbol{\lambda}\_ : Binding.\ trueProp)]\\
\quad [State' = Prime\ State]\\
\quad [State'' = Prime\ State']\\
\quad Exist\ State''\\
\qquad\quad (And\ (Exist\ State'\\
\qquad\qquad\qquad (Include\ (And\ S\ State'')\ nil\_sig\\
\qquad\qquad\qquad (\boldsymbol{\lambda} b : Binding.\\
\quad restrict\ (post\_bin\ b)\ State.\mathbf{1} = restrict\ (post\_bin\ (post\_bin\ b))\ State.\mathbf{1})))\\
\qquad\quad (Exist\ State\\
\qquad\qquad\qquad (Include\ (And\ T\ State'')\ nil\_sig\\
\qquad\qquad\qquad (\boldsymbol{\lambda} b : Binding.\\
\quad restrict\ b\ State.\mathbf{1} = restrict\ (post\_bin\ (post\_bin\ b))\ State.\mathbf{1}))))\\
: Schema \longrightarrow Schema \longrightarrow Schema
\end{array}
$$

The compose operation is meaningful only if applied to schemas whose signatures are consistent for composition, as described in the quoted extract, as well as being type-compatible. Two signatures S and T are consistent for composition if any identifier that occurs primed in S and unprimed in T is paired with the same type in both occurrences. We formalise this as a predicate $compose\_consistent : Signature \rightarrow Signature \rightarrow Prop$. We conjecture that $Compose$ preserves well-formedness if applied to schemas which are type-compatible and have the property $compose\_consistent$.

6.7 Conclusion

We have encoded some of the schema-forming operations conventionally used in Z for the specification of operations. We have used LEGO to prove some results about the metatheory of these operations, though much work remains to be done in investigating this subject.

Chapter 7

A Specification Example Continued

In this chapter we present the rest of the Birthday Book specification which was introduced in Chapter 4. The specification makes use of several of the schema-forming operations discussed in Chapters 5 and 6. We show how this specification can be encoded in LEGO and then use the encoding to formally verify a result about one of the specified operations, AddBirthday. We then encode an extension to the original specification, which specifies a robust version of the AddBirthday operation in which failure conditions are taken into account. We show that our theorem about AddBirthday can be carried over to the robust operation.

This chapter shows examples of two kinds of formal reasoning about Z specifications that are made possible by our encoding. Theorem 55 shows how we can reason at the level of the core language and Theorem 57 shows how we can take advantage of our metatheorems about the schema-level operators to prove results about schemas in a modular style.

All of the Z code in this chapter is taken from the ZRM. The proof of Theorem 55 is a formalisation of a proof in the ZRM.

7.1 The Z specification

The first schema specifies an operation which adds a new birthday to the set of birthdays recorded by the system.

AddBirthday

$$
\begin{array}{l}
\Delta BirthdayBook \\
name? : NAME \\
date? : DATE \\
\hline
name? \notin known \\
birthday' = birthday \cup \{name? \mapsto date?\}
\end{array}
$$

The next operation looks up a name in the set of birthdays, and returns the birthday associated with that name if it is in the set.

FindBirthday

$$
\begin{array}{l}
\Xi BirthdayBook \\
name? : NAME \\
date! : DATE \\
\hline
name? \in known \\
date! = birthday(name?)
\end{array}
$$

The next schema specifies an alarm operation which, when supplied with the current date, returns the set of all birthdays which fall on that date.

Remind

$$
\begin{array}{l}
\Xi BirthdayBook \\
today? : DATE \\
cards! : \mathbb{F}\ NAME \\
\hline
\forall n : NAME \bullet n \in cards! \Leftrightarrow birthday(n) = today?
\end{array}
$$

The original version of this schema in the ZRM made use of set comprehension in the predicate:

$$cards! = \{n : known \mid birthday(n) = today?\}$$

We have not allowed set comprehension in the syntax of Z' because it is difficult to see how to add this to our representation of finite sets as lists.

The initial state of the BirthdayBook system is specified by the schema:

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{InitBirthdayBook}} \\
\hline
BirthdayBook \\
\hline
known = \emptyset \\
\hline
\end{array}$$

7.2 Encoding the specification

We first define all of the signature items needed to encode this specification:

$$\begin{array}{l}
name_type \stackrel{def}{=} Given_ty\ Name_ty : Ztype \\
name_item \stackrel{def}{=} ((\mathbf{2}, query), name_type) : sig_item \\
date_type \stackrel{def}{=} Given_ty\ Date_ty : Ztype \\
date_item \stackrel{def}{=} ((\mathbf{3}, shriek), date_type) : sig_item \\
today_type \stackrel{def}{=} Given_ty\ Date_ty : Ztype \\
today_item \stackrel{def}{=} ((\mathbf{4}, query), today_type) : sig_item \\
cards_type \stackrel{def}{=} finset_ty\ (Given_ty\ Name_ty) : Ztype \\
cards_item \stackrel{def}{=} ((\mathbf{5}, shriek), cards_type) : sig_item \\
known'_item \stackrel{def}{=} decorate_item\ pr\ known_item : sig_item \\
birthday'_item \stackrel{def}{=} decorate_item\ pr\ birthday_item : sig_item
\end{array}$$

To encode the schema AddBirthday, we first define its signature and predicate and then put all the parts together using the operations *Include* and *Delta*.

Definition 40 AddBirthday.

$$AB\_sig \stackrel{def}{=} [name\_item, date\_item]$$

$$\begin{aligned}
AB_pred \stackrel{def}{=}\ & \lambda b : Binding. \\
& [name = lookup\ name_item\ b] \\
& [date = lookup\ date_item\ b] \\
& [known = lookup\ known_item\ b] \\
& [birthday = lookup\ birthday_item\ b] \\
& [birthday' = lookup\ birthday'_item\ b] \\
& \quad (IS_FALSE\ (IN\ name\ known)) \wedge \\
& \quad (IS_TRUE\ (EQUAL\ birthday' \\
& \qquad\qquad (FUN_UNION\ birthday\ (FUN_SINGLE\ name\ date))))
\end{aligned}$$

$$AddBirthday \stackrel{def}{=} Include\ (Delta\ BirthdayBook)\ AB\_sig\ AB\_pred$$

We prove that the schema *AddBirthday* is well-formed.

Theorem 51 AddBirthday_well_formed.

$$Well\_formed\ AddBirthday$$

Proof. We show first that the schema ($AddBirthday.1, AB\_pred$) is well-formed. This is a flat schema, like the schema *BirthdayBook*, and the proof of its well-formedness is basically similar to the proof that *BirthdayBook* is well-formed (Theorem 8).

Since *Delta* preserves well-formedness (Theorem 48) and *BirthdayBook* is well-formed, we can prove that *Delta BirthdayBook* is well-formed. By applying the result that *Include* preserves well-formedness (Theorem 41) to these two facts we complete the proof. □

Next we define the schema *FindBirthday*. To handle the application of the partial function *birthday*, we add an existence condition, as described in Section 4.1.2.

Definition 41 FindBirthday.

$$FB\_sig \stackrel{def}{=} [name\_item, date\_item]$$

$$\begin{aligned}
&FB_pred \stackrel{def}{=} \boldsymbol{\lambda} b : Binding\textbf{.}\\
&\quad [name = lookup\ name_item\ b]\\
&\quad [date = lookup\ date_item\ b]\\
&\quad [known = lookup\ known_item\ b]\\
&\quad [birthday = lookup\ birthday_item\ b]\\
&\quad\ (\exists\, d : Date\textbf{.}\, APPLY\ birthday\ name = in\mathbf{2}\ (Given_ty\ Date_ty, d))\ \wedge\\
&\quad\ (IS_TRUE\ (IN\ name\ known))\ \wedge\\
&\quad\ (IS_TRUE\ (EQUAL\ date\ (APPLY\ birthday\ name)))
\end{aligned}$$

$$FindBirthday \stackrel{def}{=} Include\ (Xi\ BirthdayBook)\ FB\_sig\ FB\_pred$$

Theorem 52 FindBirthday_well_formed.

$$Well\_formed\ FindBirthday$$

Proof. The proof is very similar to the proof of Theorem 51. First we show that the schema (*FindBirthday*$\mathbf{.1}$, *FB_pred*) is well-formed. We then use the fact that *Xi* preserves well-formedness to prove that *Xi BirthdayBook* is well-formed. (To do this we are required to verify that the signature of *BirthdayBook* has the property *static_sig*.) We use Theorem 41 to complete the proof. □

Here is our encoding of the schema Remind.

Definition* 42 *Remind.

$$Remind\_sig \stackrel{def}{=} [today\_item, cards\_item]$$

$$\begin{array}{l} Remind\_pred \stackrel{def}{=} \boldsymbol{\lambda} b : Binding. \\ \quad [today = lookup\ today\_item\ b] \\ \quad [cards = lookup\ cards\_item\ b] \\ \quad [known = lookup\ known\_item\ b] \\ \quad [birthday = lookup\ birthday\_item\ b] \\ \quad \forall\, n : Typ\ (givenT\ Name).\ [N = (givenT\ Name, n)] \\ \quad\ \ (IS\_TRUE\ (IN\ N\ cards)) \iff \\ \quad\ \ ((IS\_TRUE\ (IN\ N\ known)) \wedge \\ \quad\ \ \ (IS\_TRUE\ (EQUAL\ (APPLY\ birthday\ N)\ today))) \end{array}$$

$$Remind \stackrel{def}{=} Include\ (Xi\ BirthdayBook)\ Remind\_sig\ Remind\_pred$$

Theorem* 53 *Remind_well_formed.

$$Well\_formed\ Remind$$

Proof. Omitted. The proof is similar to that of Theorem 52.

Definition* 43 *InitBirthdayBook.

$$\begin{array}{l} Init\_BB\_pred \stackrel{def}{=} \boldsymbol{\lambda} b : Binding. \\ \quad [known = lookup\ known\_item\ b] \\ \quad\ IS\_TRUE\ (EQUAL\ known\ NULL) \end{array}$$

$$InitBirthdayBook \stackrel{def}{=} Include\ BirthdayBook\ nil\_sig\ Init\_BB\_pred$$

Theorem* 54 *InitBirthdayBook_well_formed.

$$Well\_formed\ InitBirthdayBook$$

Proof. We prove that the schema $(InitBirthdayBook.\mathbf{1}, Init\_BB\_pred)$ is well-formed, and then use the facts that *Include* preserves well-formedness (Theorem 41) and *BirthdayBook* is well-formed (Theorem 8) to complete the proof. □

7.3 A theorem about AddBirthday

The ZRM gives an informal proof of the following statement: the AddBirthday operation causes the set of names known to the system to be augmented with the new name supplied to the operation.

$$known' = known \cup \{name?\}$$

We shall prove this result formally in LEGO. First we formalise the statement of what we want to prove. Our goal is to show that all bindings which are restricting models of the schema *AddBirthday* satisfy the following predicate:

Definition 44 P.

$$\begin{aligned}
&P \stackrel{def}{=} \lambda b : Binding. \\
&\quad [known = lookup\ known_item\ b] \\
&\quad [known' = lookup\ known'_item\ b] \\
&\quad [name = lookup\ name_item\ b] \\
&\qquad IS_TRUE\ (EQUAL\ known'\ (UNION\ known\ (SINGLE\ name))) \\
&: Binding \rightarrow Prop
\end{aligned}$$

The formal proof goes in two stages. The first stage is essentially overhead resulting from our encoding of Z while the main proof is in the second stage.

The first stage consists of proving Lemma 4. This states that if a binding b is a restricting model of the *BirthdayBook* schema, then the signature items *known_item* and *birthday_item* can be successfully looked up in b, and the values so obtained satisfy the predicate *BB_pred* if they are substituted in place of the *lookup*s. We anticipate that this kind of proof obligation — where we show that a binding that is a restricting model of a schema does indeed provide witnesses that make the predicate of that schema hold true — will arise frequently in the use of this system. For this reason we have tried to streamline the proof script so that it can be reused easily.

Lemma 4 AddBirthday_lemma. [1]

$$
\begin{array}{l}
\forall\, b\!:\!\mathit{Binding}.\ (\mathit{restricts\_to\_model}\ \mathit{AddBirthday}\ b)\ \Rightarrow \\
\quad \exists\, \mathit{known\_v}\!:\!\mathit{Typ}\ \mathit{known\_type}.\ \exists\, \mathit{birthday\_v}\!:\!\mathit{Typ}\ \mathit{birthday\_type}. \\
\quad \exists\, \mathit{known'\_v}\!:\!\mathit{Typ}\ \mathit{known\_type}.\ \exists\, \mathit{birthday'\_v}\!:\!\mathit{Typ}\ \mathit{birthday\_type}. \\
\quad \exists\, \mathit{name\_v}\!:\!\mathit{Typ}\ \mathit{name\_type}.\ \exists\, \mathit{date\_v}\!:\!\mathit{Typ}\ \mathit{date\_type}. \\
\qquad (\mathit{lookup}\ \mathit{known\_item}\ b = \mathit{in}\mathbf{2}\ (\mathit{known\_type}, \mathit{known\_v}))\ \wedge \\
\qquad (\mathit{lookup}\ \mathit{birthday\_item}\ b = \mathit{in}\mathbf{2}\ (\mathit{birthday\_type}, \mathit{birthday\_v}))\ \wedge \\
\qquad (\mathit{lookup}\ \mathit{known'\_item}\ b = \mathit{in}\mathbf{2}\ (\mathit{known\_type}, \mathit{known'\_v}))\ \wedge \\
\qquad (\mathit{lookup}\ \mathit{birthday'\_item}\ b = \mathit{in}\mathbf{2}\ (\mathit{birthday\_type}, \mathit{birthday'\_v}))\ \wedge \\
\qquad (\mathit{lookup}\ \mathit{name\_item}\ b = \mathit{in}\mathbf{2}\ (\mathit{name\_type}, \mathit{name\_v}))\ \wedge \\
\qquad (\mathit{lookup}\ \mathit{date\_item}\ b = \mathit{in}\mathbf{2}\ (\mathit{date\_type}, \mathit{date\_v}))\ \wedge \\
\qquad (\mathit{is\_true}\ (\mathit{Set\_eq}\ \mathit{known\_v}\ (\mathit{Dom}\ \mathit{birthday\_v})))\ \wedge \\
\qquad (\mathit{is\_true}\ (\mathit{Set\_eq}\ \mathit{known'\_v}\ (\mathit{Dom}\ \mathit{birthday'\_v})))\ \wedge \\
\qquad (\mathit{is\_false}\ (\mathit{In}\ \mathit{name\_v}\ \mathit{known\_v}))\ \wedge \\
\qquad (\mathit{is\_true}\ (\mathit{Fun\_eq}\ \mathit{birthday'\_v} \\
\qquad\qquad\qquad\qquad (\mathit{FunUnion}\ \mathit{birthday\_v}\ (\mathit{FunSingle}\ \mathit{name\_v}\ \mathit{date\_v}))))
\end{array}
$$

Proof. We begin by introducing a binding b and the hypothesis:

$$H\ :\ \mathit{restricts\_to\_model}\ \mathit{AddBirthday}\ b$$

From H, we can easily show that b can successfully be restricted to the signature $\mathit{AddBirthday}.\mathbf{1}$:

$$
\begin{array}{l}
b'\ :\ \mathit{Binding} \\
H_1\ :\ \mathit{restrict}\ \mathit{AddBirthday}.\mathbf{1}\ b = \mathit{in}\mathbf{2}\ b' \\
H_2\ :\ \mathit{AddBirthday}.\mathbf{2}\ b'
\end{array}
$$

---

[1] The use of unwrapped versions of core functions (*Dom*, rather than *DOM*, *et c*) in the statement of this lemma may be confusing. The statement of the lemma is actually computationally equivalent to the corresponding statement in which only wrapped functions are used, because the wrapped functions are being applied to values of the "correct" type. See Section 3.5.7 for explanations.

We can then use Lemma 12 (*restrict_works_then_lookup_works*) to show that *known_item*, *birthday_item*, et c. can all be successfully looked up in b. Then we use Lemma 6 (*lookup_success_lemma*) to obtain witnesses for all the existentials in the goal, and to prove the equalities. For example, by applying Lemma 6 to our proof that *known_item* can be successfully looked up in b, we obtain the following:

$$
\begin{aligned}
& t : \mathit{Typ}\ \mathit{known\_type} \\
& H_3 : \mathit{lookup}\ \mathit{known\_item}\ b = \mathit{in}\mathbf{2}\ (\mathit{known\_type}, t)
\end{aligned}
$$

Similarly, we obtain values t_1, t_2, t_3, t_4 and t_5 and proofs that these are the results of looking up *birthday_item*, *known′_item*, *birthday′_item*, *name_item*, and *date_item*, respectively, in b.

Now we must show that these values satisfy the predicate $\mathit{AddBirthday}.\mathbf{2}$ if they are substituted in place of the appropriate *lookup*s:

$$
\begin{aligned}
? :\ & (\mathit{is\_true}\ (\mathit{Set\_eq}\ t\ (\mathit{Dom}\ t_1)))\ \wedge \\
& (\mathit{is\_true}\ (\mathit{Set\_eq}\ t_2\ (\mathit{Dom}\ t_3)))\ \wedge \\
& (\mathit{is\_false}\ (\mathit{In}\ t_4\ t_1))\ \wedge \\
& (\mathit{is\_true}\ (\mathit{Fun\_eq}\ t_3\ (\mathit{FunUnion}\ t_1\ (\mathit{FunSingle}\ t_4\ t_5))))
\end{aligned}
$$

By rewriting with the equalities such as H_3 that were obtained in the last step, we can transform the goal to:

$$
\begin{aligned}
?_1 :\ & (\mathit{IS\_TRUE}\ (\mathit{EQUAL}\ (\mathit{lookup}\ \mathit{known\_item}\ b) \\
& \qquad\qquad\qquad (\mathit{DOM}\ (\mathit{lookup}\ \mathit{birthday\_item}\ b))))\ \wedge \\
& (\mathit{IS\_TRUE}\ (\mathit{EQUAL}\ (\mathit{lookup}\ \mathit{known'\_item}\ b) \\
& \qquad\qquad\qquad (\mathit{DOM}\ (\mathit{lookup}\ \mathit{birthday'\_item}\ b))))\ \wedge \\
& (\mathit{IS\_FALSE}\ (\mathit{IN}\ (\mathit{lookup}\ \mathit{name\_item}\ b)\ (\mathit{lookup}\ \mathit{known\_item}\ b)))\ \wedge \\
& (\mathit{IS\_TRUE}\ (\mathit{EQUAL}\ (\mathit{lookup}\ \mathit{birthday'\_item}\ b) \\
& \qquad\qquad\qquad (\mathit{FUN\_UNION}\ (\mathit{lookup}\ \mathit{birthday\_item}\ b) \\
& \qquad\qquad\qquad\qquad (\mathit{FUN\_SINGLE}\ (\mathit{lookup}\ \mathit{name\_item}\ b) \\
& \qquad\qquad\qquad\qquad\qquad (\mathit{lookup}\ \mathit{date\_item}\ b)))))
\end{aligned}
$$

Lemma 18 (*lookup_restrict_equals_lookup_orig*) allows us to replace all of the *lookup*s in b by *lookup*s in the binding b' obtained by restricting b. Lemma 8 then allows us to replace both of the *lookup*s in b' of primed signature items (such as $known'\_item$) by *lookup*s of the unprimed item in $post\_bin\ t$. These rewrites bring the goal into a form where it matches the hypothesis H_2. Refining by this hypothesis completes the proof. □

Now we can use this lemma to prove the main result.

Theorem 55 AddBirthday_prop.

$$Has\_prop\ AddBirthday\ P$$

Proof. We introduce a binding b and a hypothesis:

$$H : restricts\_to\_model\ AddBirthday\ b$$

By applying Lemma 4 to this hypothesis, and then doing several existential eliminations, introductions, and eliminations, we arrive at the following proof context:

$$\begin{array}{l}
known_v, known'_v : Typ\ known_type \\
birthday_v, birthday'_v : Typ\ birthday_type \\
name_v : Typ\ name_type \\
date_v : Typ\ date_type \\
H_1 : lookup\ known_item\ b = in\mathbf{2}\ (known_v\ known_type) \\
\vdots \\
H_6 : lookup\ date_item\ b = in\mathbf{2}\ (date_v\ date_type) \\
H_7 : is_true\ (Set_eq\ known_v\ (Dom\ birthday_v)) \\
H_8 : is_true\ (Set_eq\ known'_v\ (Dom\ birthday'_v)) \\
H_9 : is_false\ (In\ name_v\ known_v) \\
H_{10} : is_true\ (Fun_eq\ birthday'_v \\
\qquad\qquad\qquad\qquad (FunUnion\ birthday_v\ (FunSingle\ name_v\ date_v)))
\end{array}$$

By rewriting with the equalities $H_1 \ldots H_6$, we transform the goal to:

$$? : is\_true\ (Set\_eq\ known'\_v\ (Union\ known\_v\ (Single\ name\_v)))$$

We use the fact that *Set_eq* (Lemma 41) is transitive to transform this to the following two goals:

$$?_1 : \mathit{is\_true}\ (\mathit{Set\_eq}\ (\mathit{Union}\ (\mathit{Dom}\ \mathit{birthday\_v})\ (\mathit{Single}\ \mathit{name\_v}))\ (\mathit{Union}\ \mathit{known\_v}\ (\mathit{Single}\ \mathit{name\_v})))$$

$$?_2 : \mathit{is\_true}\ (\mathit{Set\_eq}\ \mathit{known'\_v}\ (\mathit{Union}\ (\mathit{Dom}\ \mathit{birthday\_v})\ (\mathit{Single}\ \mathit{name\_v})))$$

To prove $?_1$, we use the fact that *Union* respects set equality (Lemma 42). This gives us two goals:

$$?_3 : \mathit{is\_true}\ (\mathit{Set\_eq}\ (\mathit{Dom}\ \mathit{birthday\_v})\ \mathit{known\_v})$$

$$?_4 : \mathit{is\_true}\ (\mathit{Set\_eq}\ (\mathit{Single}\ \mathit{name\_v})\ (\mathit{Single}\ \mathit{name\_v}))$$

To prove $?_3$ we use hypothesis H_7 and the fact that *Set_eq* is symmetric (Lemma 40). To prove $?_4$ we use the fact that *Set_eq* is reflexive (Lemma 39). This completes the proof of subgoal $?_1$.

Now we work on subgoal $?_2$. *Single name_v* is computationally identical to *Dom* (*FunSingle name_v date_v*). We rewrite the goal using this fact, and then use the transitivity of *Set_eq* to transform it into the following two subgoals:

$$?_5 : \mathit{is\_true}\ (\mathit{Set\_eq}\ (\mathit{Union}\ (\mathit{Dom}\ \mathit{birthday\_v})\ (\mathit{Dom}\ (\mathit{FunSingle}\ \mathit{name\_v}\ \mathit{date\_v})))\ (\mathit{Dom}\ (\mathit{FunUnion}\ \mathit{birthday\_v}\ (\mathit{FunSingle}\ \mathit{name\_v}\ \mathit{date\_v}))))$$

$$?_6 : \mathit{is\_true}\ (\mathit{Set\_eq}\ (\mathit{Dom}\ (\mathit{FunUnion}\ \mathit{birthday\_v}\ (\mathit{FunSingle}\ \mathit{name\_v}\ \mathit{date\_v})))\ \mathit{known'\_v})$$

Goal $?_5$ is proved by refining by Lemma 45. To prove goal $?_6$ we again use the fact that *Set_eq* is transitive. Our new subgoals are:

$$?_7 : \mathit{is\_true}\ (\mathit{Set\_eq}\ (\mathit{Dom}\ (\mathit{birthday'\_v}))\ \mathit{known'\_v})$$

$$?_8 : \mathit{is\_true}\ (\mathit{Set\_eq}\ (\mathit{Dom}\ (\mathit{FunUnion}\ \mathit{birthday\_v}\ (\mathit{FunSingle}\ \mathit{name\_v}\ \mathit{date\_v})))\ (\mathit{Dom}\ (\mathit{birthday}'_v)))$$

Goal $?_7$ is proved by refining by H_8 and the fact that *Set_eq* is transitive. To prove goal $?_8$ we use Lemma 44 to reduce the goal to

$$?_9 : \mathit{is\_true}\ (\mathit{Fun\_eq}\ (\mathit{FunUnion}\ \mathit{birthday\_v}\ (\mathit{FunSingle}\ \mathit{name\_v}\ \mathit{date\_v}))\ \mathit{birthday'\_v})$$

We prove goal $?_9$ by using H_{10} and the fact that *Fun_eq* is symmetric (Lemma 43.) $\square$

7.4 Specifying a robust system

The ZRM gives a specification of a robust version of the BirthdayBook system, in which allowances are made for the failure of the various operations. We shall focus on the modified specification of the operation AddBirthday.

In the robust system, each operation produces an output *result*! which indicates whether or not that operation was successful. The values that can be assigned to this output come from a type *REPORT* which is defined as follows:

$$REPORT ::= ok \mid already\_known$$

The following schema, *Success*, specifies a state in which the *result* output has the value *ok*.

Success

$result! : REPORT$

$result! = ok$

The next schema, *AlreadyKnown*, specifies an operation which acts upon the state space of the birthday book and takes an input *name*?, and which produces the result *already_known* if *name*? is among the set of known names in the birthday book.

AlreadyKnown

$\Xi BirthdayBook$
$name? : NAME$
$result! : REPORT$

$name? \in known$
$result! = already\_known$

The robust version of the AddBirthday operation is defined as follows:

$$RAddBirthday \mathrel{\widehat{=}} (AddBirthday \wedge Success) \vee AlreadyKnown$$

7.4.1 Encoding the schema RAddBirthday

We represent the type *REPORT* by extending the type *GivenType* with a new constructor *Report_ty*, and defining a new inductive type *Report*, with constructors *ok* and *already_known*, to represent the semantics of this new *GivenType*. (In other words, the function *Typ* maps *given_ty Report_ty* to the type *Report*.) We also define a decidable equality $Report\_eq : Report \rightarrow Report \rightarrow bool$.

We define the signature item *result_item*:

$$result\_item \stackrel{def}{=} ((\mathbf{6}, shriek), given\_ty\ Report\_ty) : sig\_item$$

We encode the schema *Success* as follows:

Definition* 45 *Success.

$$\begin{array}{l} Success\_sig \stackrel{def}{=} [result\_item] : Signature \\ Success\_pred \stackrel{def}{=} \boldsymbol{\lambda} b\!:\!Binding\textbf{.}\ [result = lookup\ result\_item\ b] \\ \quad IS\_TRUE\ (EQUAL\ result\ OK) \\ : Predicate \\ Success \stackrel{def}{=} (Success\_sig, Success\_pred) : Schema \end{array}$$

We can prove that *Success* is well-formed. The proof is obtained by slightly modifying the proof of the fact that *BirthdayBook* is well-formed (Theorem 8).

Next we encode the schema *AlreadyKnown*:

Definition 46 AlreadyKnown.

$$
\begin{aligned}
&AK\_sig \stackrel{def}{=} [name\_item, result\_item] : Signature\\
&AK\_pred \stackrel{def}{=} \boldsymbol{\lambda} b : Binding.\\
&\quad [name = lookup\ name\_item\ b]\\
&\quad [known = lookup\ known\_item\ b]\\
&\quad [result = lookup\ result\_item\ b]\\
&\quad\ (IS\_TRUE\ (IN\ name\ known)) \wedge\\
&\quad\ (IS\_TRUE\ (EQUAL\ result\ ALREADY\_KNOWN))\\
&: Predicate
\end{aligned}
$$

$$AlreadyKnown \stackrel{def}{=} Include\ (Xi\ BirthdayBook)\ AK\_sig\ Ak\_pred : Schema$$

We prove that *AlreadyKnown* is well-formed.

Theorem 56 AlreadyKnown_well_formed.

$$Well\_formed\ AlreadyKnown$$

Proof. The proof is a slight modification of the proof of Theorem 52. □

Finally, we encode the schema *RAddBirthday*:

Definition 47 RAddBirthday.

$$RAddBirthday \stackrel{def}{=} Or\ (And\ AddBirthday\ Success)\ AlreadyKnown : Schema$$

Since this is composed of well-formed schemas (Theorems 51 and 56) and the operations *Or* and *And* preserve well-formedness (Theorems 16 and 10), we can easily show that it is well-formed.

7.4.2 RAddBirthday has property P

We use our metatheorems about schema operations to prove a version of Theorem 55 for the robust version of the AddBirthday operation. We show that bindings which are restricting models of *RAddBirthday* either satisfy the property P or represent failure as specified by *AlreadyKnown*. First, we define this new property:

$$RobustP \stackrel{def}{=} \boldsymbol{\lambda} b : Binding.\ (P\ b) \vee (restricts\_to\_model\ b\ AlreadyKnown)$$

Theorem 57 RAddBirthday_prop.

$$Has\_prop\ RAddBirthday\ RobustP$$

Proof. We refine by Theorem 22 which gives us the following subgoals:

$$?_1 : Has\_prop\ (And\ AddBirthday\ Success)\ RobustP$$
$$?_2 : Has\_prop\ AlreadyKnown\ RobustP$$

To prove the first subgoal we use Theorem 15:

$$?_3 : (Has\_prop\ AddBirthday\ RobustP) \vee (Has\_prop\ Success\ RobustP)$$

We prove this by using Theorem 55 to prove the first disjunct. We are then left with subgoal $?_2$ which is trivial. $\square$

Chapter 8

Speculations

In this chapter we speculate on ways of extending our encoding with notions of implementation and refinement.

The type theory UTT can be thought of as a rudimentary programming language in which terms are evaluated by normalisation. This has been exploited by others as a means of developing UTT and LEGO into a tool for reasoning about programs ([Hof92,Sch95,McK92]). In this chapter we shall look at one way in which this can be harnessed in order to define a notion of implementation for Z specifications. We first choose a UTT type which seems to provide a notion of program that is complementary to our representation of Z schemas. We call this type *Program*. We define an implementation relationship between the types *Schema* and *Program*. We shall then give a more conventional syntax for a simple programming language and show how terms in this language can be translated into the type *Program*.

8.1 Defining a type to represent programs

As we have seen in Chapter 6, when an operation is specified by a Z schema, a single signature is used to specify the inputs, outputs, and the state spaces before and after that operation is executed. A binding which matches the signature of such a schema will contain values representing all of these components. The decoration on an individual signature item (or binding item) identifies which of these components that item belongs to.

This leads to a natural notion of "program". A program can be thought of as a computable function which, when supplied with bindings which represent its inputs and the state before its execution, returns two more bindings, representing its outputs and the state after execution. In a type theory such as UTT, where all functions are computable, this can be represented by the type $(Binding \times Binding) \rightarrow (Binding \times Binding)$.

In practise, when we attempt to encode programs as terms of this type, we shall need to use the *lookup* function in order to obtain the values associated with identifiers in bindings. This means that we need to consider what happens when a program is applied to a pair of bindings which do not contain all the identifiers required by that program. In such a case some of the applications of *lookup* will fail, and this must be handled somehow. This is essentially the same issue which we dealt with in regard to schema predicates in Section 3.5.6. Our solution there was to have predicates and relations "collapse" to the value *absurd* when applied to a failed *lookup*. For programs, we shall adopt one of the alternate approaches discussed in Section 3.5.6: when a *lookup* fails we shall cause the program to return an error value. Hence we define the type *Program* as follows:

Definition 48 Program.

$$Program \stackrel{def}{=} (Binding \times Binding) \rightarrow Error + (Binding \times Binding) : Type$$

It would be possible for us to use instead a solution analogous to that used for schema predicates. We could have programs return some arbitrarily chosen pair of bindings in the case where *lookup*s fail. However, there is little advantage in using this technique for programs. Schemas are combined in various ways by a multitude of operations, so it quickly becomes tedious to have to deal with error results when encoding these operations. In contrast, we do not anticipate having many operations which combine programs so the cost of using error results to handle failure is not so great. The advantage of using error results is that we have a more informative model: by looking at the result returned by applying a program we can tell whether any *lookup*s failed when that result was computed. This information is lost if programs return binding pairs whether or not their *lookup*s succeed.

8.2 The programming language

We have identified a type *Program* which can serve as a model of a programming language. Now we shall look at what this programming language might be, and how it may be translated into terms of the type *Program*. Figure 8–1 gives a syntax for a very simple, imperative programming language, which has been devised so as to be easy to translate into type theory. (For this reason, we do not have while loops, for example.) Our programming language is intended only to give a taste of what is possible in formalising a notion of implementation in Z. More sophisticated programming languages than this can be encoded in type theory. (For example, it is possible to define a restricted form of general recursion for programs which are provably terminating.)

A program in our language consists of a sequence of procedure declarations and variable declarations, followed by a list of statements which make up the program body. A procedure declaration begins with a procedure name followed by two lists of declarations. The first list contains the names of the input and output variables used by that procedure. Output variables are preceded by the keyword `var`. (It is not possible for a variable to be simultaneously an input and an output variable.) The second list of declarations contains the variables declared locally to that procedure. The body of a procedure is a sequence of statements which may be assignments (to variables or to array elements), for loops, or if statements. A procedure is executed by being called with a list of values which supply the inputs to that program. We shall not go into the formalities of the semantics of the programming language.

```
Program = Decl;...Decl;Body
ProcDecl =
   procedure ProcName (IoDecl;...;IoDecl)
                 VarDecl;...;VarDecl;
                  ProgBody
IoDecl = InputDecl | OutDecl
InputDecl = VarDecl
OutDecl = var VarDecl
VarDecl = Ident:Type
Decl = VarDecl | ProcDecl
Body = begin Statement;...;Statement end
ProgBody = begin ProgStatement;...;ProgStatement end
Statement = Ident := Expression |
              Ident [Expression] := Expression
              for Ident := Expression to Expression do Statement |
              if Expression then Statement else Statement |
              begin Statement;...;Statement end|
ProcCall = ProcName (Expression;...;Expression)
ProgStatement = Statement | ProcCall
Expression = Ident | Integer | Boolean |
               (Op Expression) | (Expression BinOp Expression) |
               Ident [Expression]
Op = not
BinOp = + | = | < | and | or
```

The definitions of Ident and Type, the typechecking rules and the evaluation rules are all omitted.

Figure 8–1: The programming language

Our programming language can be modelled in UTT via the type *Program*. (Note that not all *Programs* correspond to terms in the programming language.) Procedures and programs will both be represented by the type *Program*. Here we describe the key points of the translation.

We shall re-use the types *Ident* and *Ztype* as representations of the identifiers and types of our programming language. The type of arrays of elements of some type `t` is translated as the *Ztype*, *fun_ty nat_ty t*, where *t* is the *Ztype* representing `t`. Let us suppose that we have a procedure `P`, which we wish to represent as a term of type *Program*. Suppose that some identifier `i` is used as an Expression within the body of `P`. We shall call such an Expression an *identifier reference*. The key to our translation of procedures lies in the way identifier references are translated. Suppose that the `i` is translated as an *Ident*, *i*. The translation of the identifier reference `i` depends upon where the identifier `i` is declared in the procedure `P`. If `i` is among the input declarations of `P`, then the reference `i` is translated by looking up *i* in the input binding. If `i` is a local declaration in `P` then it is represented as a locally defined variable in UTT, and the reference `i` is translated as a reference to that variable. Finally, if `i` is not among the declarations of `P`, then the reference `i` is translated by looking up *i* in the binding representing the state before the execution of `P`.

The rest of the encoding of Expressions, Statements, *et c*, follows naturally from the way identifier references are encoded. Operations are encoded via suitably wrapped (in the sense of Section 3.5.6) versions of the corresponding operations in the LEGO library. For statements are encoded as wrapped versions of *nat_rec* or *bool_rec*, as appropriate. If statements are encoded by wrapping the LEGO library *if* operation. Assignments are encoded by the way in which the output and post-state bindings are constructed. Some examples which illustrate the process are given in Section 8.3.

It is straightforward to translate the composition of statements which make up a procedure body. Doing the same for a program body is not so straightforward, since the class ProgStatement includes procedure calls, which are state-changing operations. It is possible to define a composition operation in type theory, which computes the composition of two procedures. However, we shall not discuss this any further.

8.3 Examples

The programs which we shall use as examples are based on code that is presented in the ZRM as implementations of the operations of the Birthday Book system. Later we shall look at defining an implementation relationship, but, for the moment, we shall just consider the translation of these programs into terms of type *Program*.

The programs operate within a global state which contains an integer variable `hwm` (read as "high water mark") and two arrays `name`, of type `NAME` and `date`, of type `DATE`. (Let us assume that the programming language provides some means of defining type abbreviations, and that `NAME` and `DATE` are two previously defined type abbreviations.) This state represents the Birthday Book. The variable `hwm` records the number of birthdays currently in the system. The array `name` holds all the names in the system: the birthday associated with `name[i]` is found by looking up `date[i]`.

The first program is intended to implement the AddBirthday operation:

```
procedure AddBirthday (name:NAME; date:DATE);
begin
   hwm := hwm + 1;
   names[hwm] := name;
   dates[hwm] := date
end
```

To encode this program in UTT, we first encode the identifiers `hwm`, `names`, and `dates` as *Ident*s called *hwm_i*, *names_i*, and *dates_i*, respectively. We pair these with the appropriate *Ztype*s (*nat_ty*, *fun_ty nat_ty Name*, and *fun_ty nat_ty Date*, respectively) to form signature items called *names_item*, *dates_item* and *hwm_item*.

We also define primed versions of these signature items, and call them $names'\_item$, $dates'\_item$, and hwm'_item.

The encoding makes use of a number of auxiliary functions. The first is $mk\_sum\_bin\_item$ (abbreviated as $msbi$) which has type $Ident \longrightarrow (Error + small\_item) \longrightarrow (Error + bin\_item)$. This simply adds a given $Ident$ to a $small\_item$ to produce a bin_item, returning the value $in1\ error$ if it is applied to $in1\ error$. We also define partial versions of the $cons$ and pairing operations applied to bindings:

$$CONSBIN \stackrel{def}{=} [SeeAppendixC.\mathbf{2}]$$
$$: (Error + bin\_item) \longrightarrow (Error + Binding) \longrightarrow (Error + Binding)$$
$$PAIRBIN \stackrel{def}{=} [SeeAppendixC.\mathbf{2}]$$
$$: (Error + Binding) \longrightarrow (Error + Binding) \longrightarrow (Error + (Binding \times Binding))$$

We shall also need to define an update operator on finite functions. The wrapped version of this operator is called $UPDATE$.

The program `AddBirthday` is encoded as a $Program$, $AddBirthday\_prog$:

Definition 49 AddBirthday_prog.

$$
\begin{aligned}
&\mathit{AddBirthday\_prog} \stackrel{def}{=} \lambda \mathit{pre\_state}, \mathit{input} : \mathit{BInding}. \\
&\quad [\mathit{HWM} = \mathit{lookup\ hwm\_item\ pre\_state}] \\
&\quad [\mathit{NAME} = \mathit{lookup\ name\_item\ input}] \\
&\quad [\mathit{DATE} = \mathit{lookup\ date\_item\ input}] \\
&\quad [\mathit{NAMES} = \mathit{lookup\ names\_item\ input}] \\
&\quad [\mathit{DATES} = \mathit{lookup\ dates\_item\ input}] \\
&\quad [\mathit{HWM\_ITEM} = \mathit{msbi\ hwm\_i}\ (\mathit{PLUS\ HWM\ ONE})] \\
&\quad [\mathit{NAMES\_ITEM} = \mathit{msbi\ names\_i} \\
&\qquad\quad (\mathit{UPDATE\ NAMES}\ (\mathit{PLUS\ HWM\ ONE})\ \mathit{NAME})] \\
&\quad [\mathit{DATES\_ITEM} = \mathit{msbi\ names\_i} \\
&\qquad\quad (\mathit{UPDATE\ DATES}\ (\mathit{PLUS\ HWM\ ONE})\ \mathit{DATE})] \\
&\quad [\mathit{post\_state} = \mathit{CONSBIN\ HWM\_ITEM}\ (\mathit{CONSBIN\ NAMES\_ITEM} \\
&\qquad\qquad\qquad\qquad\qquad (\mathit{CONSBIN\ DATES\_ITEM\ NIL\_BIN}))] \\
&\quad [\mathit{output} = \mathit{NIL\_BIN}] \\
&\qquad (\mathit{PAIRBIN\ post\_state\ output}) \\
&: \mathit{Program}
\end{aligned}
$$

AddBirthday first looks up all of the input and pre-state identifiers in the appropriate bindings. It then computes binding items for the post-state, using wrapped operations as appropriate. The binding items are put together to form the post-state binding and the output binding (which, in this case, is empty), and these two bindings are returned as the result. In the case where any of the *lookup*s fail, the value $\mathit{in}\mathbf{1}\ \mathit{error}$ will be returned instead.

This next program is intended to implement the FindBirthday operation. (Note: this is not the program given in the ZRM, since that program uses a while loop which is not part of our programming language.)

```
procedure FindBirthday (name:NAME; var date:Date);
   var i : INTEGER;
begin
   for i := 1 to hwm do
```

```
  begin
    if names[i] = name then date := dates[i]
  end
end
```

To encode the for-loop in `FindBirthday` we shall use a wrapped version of the recursion operator *nat_rec*:

$$
\begin{aligned}
&FOR \stackrel{def}{=} \text{[See Appendix C.2]}\\
&: \mathbf{\Pi}\, t \,|\, Type\textbf{.}\ t \longrightarrow (nat \longrightarrow t \longrightarrow t) \longrightarrow (Error + small\_item) \longrightarrow (Error + t)
\end{aligned}
$$

The first argument taken by *FOR* is a starting value for recursion, and the second argument is the step function. The third argument gives the number of iterations to perform. If this argument is $in\mathbf{1}\ error$, or has a *Ztype* other than *nat_ty*, then the result returned is $in\mathbf{1}\ error$, otherwise the result is computed using *nat_rec*.

The if statement in `FindBirthday` is encoded using a wrapped version of the function *if*:

$$
\begin{aligned}
&IF \stackrel{def}{=} \text{[See Appendix C.2]}\\
&: \mathbf{\Pi}\, t \,|\, TYPE\textbf{.}\ (Error + small\_item) \longrightarrow (Error + t) \longrightarrow (Error + t) \longrightarrow (Error + t)
\end{aligned}
$$

Here is the encoding of the program `FindBirthday`:

Definition 50 FindBirthday_prog.

$$
\begin{array}{l}
\mathit{FindBirthday\_prog} \stackrel{\mathit{def}}{=} \boldsymbol{\lambda} \mathit{pre\_state}, \mathit{input}\!:\!\mathit{Binding}.\\
\quad [\mathit{NAMES} = \mathit{lookup\ names\_item\ pre\_state}]\\
\quad [\mathit{DATES} = \mathit{lookup\ dates\_item\ pre\_state}]\\
\quad [\mathit{HWM} = \mathit{lookup\ hwm\_item\ pre\_state}]\\
\quad [\mathit{NAME} = \mathit{lookup\ name\_item\ input}]\\
\quad [\mathit{preDATE} =\\
\qquad \mathit{FOR}\ (\mathit{in}\mathbf{1}\ \mathit{Error})\\
\qquad\quad (\boldsymbol{\lambda} i\!:\!\mathit{nat}.\ \boldsymbol{\lambda} x\!:\!\mathit{Error} + \mathit{small\_item}.\\
\qquad\qquad\quad \mathit{IF}\ (\mathit{EQUAL\ NAME}\ (\mathit{APPLY\ NAMES}\ (\mathit{in}\mathbf{2}\ (\mathit{nat\_ty}, i))))\\
\qquad\qquad\qquad (\mathit{APPLY\ DATES}\ (\mathit{in}\mathbf{2}\ (\mathit{nat\_ty}, i)))\\
\qquad\qquad\qquad x)]\\
\quad [\mathit{DATE} = \mathit{case}\ (\boldsymbol{\lambda}\_\!:\!\mathit{Error}.\ \mathit{in}\mathbf{1}\ \mathit{Error})\\
\qquad\qquad\quad (\boldsymbol{\lambda} x\!:\!\mathit{Error} + \mathit{small\_item}.\ x)\\
\qquad\qquad\quad \mathit{preDATE}]\\
\quad [\mathit{DATE\_ITEM} = \mathit{msbi\ date\_i\ DATE}]\\
\quad [\mathit{post\_state} = \mathit{in}\mathbf{2}\ \mathit{pre\_state}]\\
\quad [\mathit{output} = \mathit{CONSBIN\ DATE\_ITEM\ NIL\_BIN}]\\
\qquad (\mathit{PAIRBIN\ post\_state\ output})\\
: \mathit{Program}
\end{array}
$$

The final program which we shall translate is the simplest. It initialises the Birthday Book system:

```
procedure InitBirthdayBook;
begin
  hwm := 0
end
```

This is encoded as follows:

Definition 51 InitBirthdayBook_prog.

$$\begin{array}{l} \mathit{InitBirthdayBook\_prog} \stackrel{def}{=} \boldsymbol{\lambda}\_, \_ : \mathit{Binding}. \\ \quad [\mathit{HWM\_ITEM} = \mathit{in}\mathbf{2}\ (\mathit{hwm\_item}, \mathit{zero})] \\ \quad [\mathit{post\_state} = \mathit{CONSBIN\ HWM\_ITEM\ NIL\_BIN}] \\ \quad [\mathit{output} = \mathit{in}\mathbf{2}\ \mathit{NIL\_BIN}] \\ \qquad (\mathit{PAIRBIN\ post\_state\ output}) \\ : \mathit{Program} \end{array}$$

8.4 Refinement and implementations in the ZRM

Before we go on to discuss how to define an implementation relationship between programs and schemas, we shall say something about how this is handled in the ZRM. The Z notation itself does not incorporate a programming language or a notion of implementation. However, the ZRM suggests, by means of examples, a method of dealing with these concepts.

In brief, the method suggested in the ZRM goes like this. We begin with an abstract specification. We formulate a concrete specification of the same system, relating the state spaces and the operations in the two specifications by means of *abstraction relations* defined as schemas. (Moving from the abstract to the concrete specification may take place in a series of steps.) The concrete specification is written so as to be as close as possible to a programming language. For instance, the only types used are ones that are available in programming languages. The concrete schemas are then implemented directly by programs written in some programming language.

The weakest step in this process is the final one. The programming language is outside of the formalism of Z, so it is unclear how to verify that a program correctly implements a concrete schema. In some cases (such as those given in the ZRM) this will be obvious, but in other cases this may represent a significant proof obligation.

8.4.1 An example

This is an extract from the example used in the ZRM to illustrate how schemas can be refined to implementations. Again, the system used for the example is the Birthday Book.

First, a decision is made about how to implement the Birthday Book. The system will be implemented via two arrays:

```
names : Array[1...] of NAME;
dates : Array[1...] of DATE;
```

The arrays used in the implementation can be modelled in Z by functions from a set $\mathbb{N}_1$ of strictly positive integers to *NAME* or *DATE*:

$$names : \mathbb{N}_1 \rightarrow NAME$$
$$dates : \mathbb{N}_1 \rightarrow DATE$$

The ith element of the array is represented by the value of the appropriate function applied to the argument i. Assignment to array elements is modelled by function update.

Next, the ZRM gives a concrete specification of the state space of the Birthday Book system. The types used in this schema model the types that will be used in the implementation.

*BirthdayBook***1**

$names : \mathbb{N}_1 \rightarrow NAME$

$dates : \mathbb{N}_1 \rightarrow DATE$

$hwm : N$

$\forall\, i, j : \mathbf{1} \ldots hwm \bullet i \neq j \Rightarrow names\ (i) \neq names\ (j)$

The relationship between the schemas *BirthdayBook***1** and *BirthdayBook* is described by an abstraction schema (See Section 8.6).

The next schema gives a concrete specification of the AddBirthday operation:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{AddBirthday}\mathbf{1}} \\
\hline
\Delta \textit{BirthdayBook}\mathbf{1} \\
\textit{name}? : \textit{NAME} \\
\textit{date}? : \textit{DATE} \\
\hline
\forall\, i : \mathbf{1} \ldots \textit{hwm} \bullet \textit{name}? \neq \textit{names}(i) \\
\quad \textit{hwm}' = \textit{hwm} + \mathbf{1} \\
\quad \textit{names}' = \textit{names} \oplus \{\textit{hwm}' \mapsto \textit{name}?\} \\
\quad \textit{dates}' = \textit{dates} \oplus \{\textit{hwm}' \mapsto \textit{date}?\} \\
\hline
\end{array}
$$

Again, this is related to the abstract schema, AddBirthday, by means of an abstraction schema, which we shall not show.

The initial state of the concrete system is specified by the following schema.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{InitBirthdayBook}\mathbf{1}} \\
\hline
\textit{BirthdayBook}\mathbf{1} \\
\hline
\textit{hwm} = \mathbf{0} \\
\hline
\end{array}
$$

The ZRM then presents the programs `AddBirthday` and `InitBirthdayBook` as implementations of the operations specified by the schemas *AddBirthday*$\mathbf{1}$ and *InitBirthdayBook*$\mathbf{1}$. However, no justification is given for this claim. In the next section we shall see how this relationship can be formally defined using UTT.

8.5 The implementation relationship

We shall define an implementation relationship between *Schema*s and *Program*s. First, we define some simple predicates on *Binding*s. (The details of the definitions are omitted.)

$$\textit{is\_pre\_state}, \textit{is\_post\_state}, \textit{is\_input}, \textit{is\_output} : \textit{Binding} \rightarrow \textit{Prop}$$

The predicate *is_pre_state* is true of a *Binding* provided that none of the identifiers in that *Binding* are decorated in any way. The other predicates, *is_post_state*, *is_input*, and *is_output*, check that all identifiers are decorated with, respectively, *prime*, *query*, and *shriek*.

Definition 52 Implements.

$$
\begin{array}{l}
\mathit{Implements} \stackrel{\mathit{def}}{=} \boldsymbol{\lambda} S : \mathit{Schema}.\ \boldsymbol{\lambda} P : \mathit{Program}. \\
\quad \forall\, b, b_1 : \mathit{Binding}. \\
\qquad\qquad ((\mathit{is\_pre\_state}\ b) \wedge (\mathit{is\_input}\ b_1) \wedge \\
\qquad\qquad (\mathit{restricts\_to\_model}\ (\mathit{pre}\ S)\ (\mathit{join\_bin}\ b\ b_1))) \Rightarrow \\
\qquad\quad \exists\, b', b_2 : \mathit{Binding}. \\
\qquad\qquad (P\ b\ b_1 = \mathit{in}2\ (b', b_2)) \wedge \\
\qquad\qquad (\mathit{restricts\_to\_model}\ S \\
\qquad\qquad\quad (\mathit{join\_bin}\ b\ (\mathit{join\_bin}\ b_1\ (\mathit{join\_bin}\ (\mathit{decorate\_bin}\ \mathit{pr}\ b')\ b_2)))) \\
: \mathit{Schema} \longrightarrow \mathit{Binding} \longrightarrow \mathit{Prop}
\end{array}
$$

This definition states that a program P implements a schema S provided that the following conditions hold. Whenever P is applied to a pre-state binding b and an input binding b_1 which together satisfy the precondition schema *pre* S, the application succeeds, producing a post-state binding b' and an output binding b_2. If the output binding is decorated with primes, and all four bindings are then joined together, the resulting binding is a restricting model of the schema S.

Further work needs to be carried out to test whether this is a useful way of defining the implementation relationship, and to discover how difficult it is, in practice, to prove that a *Schema* and a *Program* are related under this relation. Another important area of exploration is whether the proof of the implementation relationship can be done in a *modular* fashion, exploiting the modularity in the structure of both specifications and programs.

8.6 Further Work: refinement

In the Birthday Book example, abstraction schemas are used to relate the abstract and concrete specifications of the system. For example, the schemas *BirthdayBook* and *BirthdayBook*1 are related by the following schema:

$$
\begin{array}{|l}
\textit{Abs} \\
\hline
\textit{BirthdayBook} \\
\textit{BirthdayBook}\mathbf{1} \\
\hline
\textit{known} = \{i : \mathbf{1} \ldots \textit{hwm} \bullet \textit{names}\ (i)\} \\
\forall\, i : \mathbf{1} \ldots \textit{hwm} \bullet \textit{birthday}\ (\textit{names}\ (i)) = \textit{dates}\ (i) \\
\hline
\end{array}
$$

This schema describes a relationship between the state spaces described by the two schemas *BirthdayBook* and *BirthdayBook*1. However, it is unclear from examining the schema *Abs* what exactly needs to be proved in order to verify that *BirthdayBook*1 is a correct refinement of *BirthdayBook*.

In the specification language VDM [Jon86], clearly defined proof obligations are used to express a refinement (or *reification*) relationship between two data types. For example, in order to show that a data type *Rep* is an adequate representation of a more abstract datatype, *Abs*, the following proof obligation must be met:

$$
\begin{aligned}
&\exists\, retr : Rep \rightarrow Abs. \\
&\qquad \forall\, a \in Abs.\ \exists\, r \in Rep. retr(r) = a
\end{aligned}
$$

Other proof obligations are used to express a refinement relationship between operations on data types.

The VDM notion of refinement has been extensively studied ([HJ88,JHS86]) and is relatively well understood. It is therefore useful to see whether a similar definition can be applied to specifications in Z. Expressed in the language of our encoding, here is such a definition:

Definition 53 Refines.

$$
\begin{aligned}
&\mathit{Refines} \stackrel{def}{=} \lambda A, C : \mathit{Schema}.\\
&\quad \exists\, \mathit{retr} : \mathit{Binding} \rightarrow \mathit{Binding}.\\
&\qquad (\forall\, b : \mathit{Binding}.\ (\mathit{exactly\_models}\ b\ C) \Rightarrow (\mathit{exactly\_models}\ (\mathit{retr}\ b)\ A)) \wedge\\
&\qquad (\forall\, a : \mathit{Binding}.\ (\mathit{exactly\_models}\ a\ A) \Rightarrow\\
&\qquad\qquad\qquad\qquad (\exists\, c : \mathit{Binding}.\ (\mathit{exactly\_models}\ c\ C) \wedge (\mathit{retr}\ c = a)))\\
&: \mathit{Schema} \rightarrow \mathit{Schema} \rightarrow \mathit{Prop}
\end{aligned}
$$

This states that a schema C is a refinement of a schema A, provided that the following conditions are satisfied. There must exist a retrieval function $\mathit{retr} : \mathit{Binding} \rightarrow \mathit{Binding}$ which maps exact models of C to exact models of A. For every exact model a of A, there must exist a concrete representation c which is an exact model of C, such that $\mathit{retr}\ c = a$.

Further work needs to be done to study this notion of refinement. For example, are exact models the appropriate notion of model to use here, or should we use restricting models instead? How are the two definitions different?

Another interesting question is what are the consequences of defining the refinement relationship in a constructive type theory? How does this relate to the set-theoretic definition used in VDM?

The proof obligations involved in refining operations should also be defined and studied.

8.7 Conclusion

We have seen that it is possible to define a programming language in UTT in such a way that programs and schemas become part of the same formalism. This allows us to formally define an implementation relationship between the two. The work in this chapter is intended only to suggest a way towards achieving this goal; further work needs to be done to test the concepts that have been defined. It would also be interesting to see if the principles of [Mor90] could be integrated into our work.

Chapter 9

Conclusions

9.1 Extending the encoding

The subset of Z which we have encoded contains a number of serious restrictions. How difficult would it be to remove these?

To enrich our core language, we would need a good encoding of set theory in type theory. Unfortunately, this seems to be difficult to achieve. An alternative strategy would be to dispense with the idea of set theory altogether, and to use a core language more closely based on the type theory. This would take us rather far from Z, but might be useful in its own right.

It is difficult to see how to reintroduce the free use of terms as types, and of schemas as types without having to resort to a deep embedding. The latter would, in any case, conflict with our desire to maintain separate core and module languages.

In the Z notation, it is possible, and often useful, for schemas to be parameterised over types. These parameters may then be used to build the types used in the schema signature. Unfortunately, it is difficult to see how to add such a feature to our encoding. We would need to incorporate parameterization and parameter substitution into our syntactic types (*Ztype*) and into the semantic function (*Typ*). It is not obvious to us how to do this.

9.2 Comments about LEGO

To develop our work into a realistic tool for users, based upon the LEGO system it would be nice to have: user-definable tactics to help discharge routine proof obligations (type-compatibility, well-formedness, etc); an interface to implement the translation of Z' specifications to LEGO input; a search feature (like that of HOL) to cut down the amount of time time spent searching for the names of theorems and definitions within the libraries or within the user's own files.

9.3 Comments about Z

The Z notation is modular, in the sense that one can prove theorems about schemas in a compositional way, using metatheorems about the schema operators. (Modularity in the more traditional sense of "information hiding" is also present, in the form of the hiding operator.) These metatheorems suggest that the schema operators of Z provide a potentially useful mechanism for organising theories.

The specification of operations, however, seems to be treated as an afterthought. The decorations on variables make no difference to their treatment in the official semantics even though these decorations have an important meaning in terms of a user's understanding of a Z specification.

The Z notation, therefore, seems to be composed of two separate languages whose relationship with each other is difficult to understand. There is a *static* language in which theories (schemas) may be combined using logical operators, and there is a *dynamic* part consisting of the conventions for defining operations. It is unclear how these two parts interact: more concretely, for example, it is not obvious what theorems to prove relating, say, the *Prime* and the *And* operations.

As a consequence of the way in which Z deals with the specification of operations , definitions of "refinement" and "implementation" do not seem to arise naturally, though, as we have seen, it is not impossible to define such notions.

9.4 The main contributions of this thesis

- We have demonstrated that type theory is expressive and affords a variety of representation techniques that is much richer than the shallow/deep dichotomy provided by less expressive systems like HOL.

- A significant portion the Z notation has been formalised in type theory. This provides a basis for reasoning formally about the Z notation and about specifications expressed in Z.

- We have shown that, within certain constraints it is possible to reason about Z specifications at a modular level. Evidence for this is provided by various theorems about the schema operations. An example is given to illustrate the kind of modular proof that is supported.

- Type theory provides a broad spectrum language incorporating a programming language. We have used this to illustrate how the Z notation can be enriched by means of a programming language that is within the same formal system. This gives us a basis for formalising a notion of implementation between programs and Z specifications.

Appendix A

Proof Descriptions

A.1 Proofs of theorems in Chapter 3

Proof 1 $\mathbf{Equiv_1\_implies\_Equiv_2}$.

Goal: $\forall\, S, T : Schema.\ (Equiv_1\ S\ T) \Rightarrow (Equiv_2\ S\ T)$

We introduce two schemas S and T. We then do a case analysis on whether S and T have equal signatures. In the first case, we have:

H : $is\_true\ (sig\_eq\ S.1\ T.1)$

By expanding definitions and doing introductions, we transform the proof context to the following:

H_1 : $\forall\, b : Binding.\ (exactly\_models\ S\ b) \Longleftrightarrow (exactly\_models\ T\ b)$

b : $Binding$

$?_1$: $(restricts\_to\_model\ S\ b) \Rightarrow (restricts\_to\_model\ T\ b)$

$?_2$: $(restricts\_to\_model\ T\ b) \Rightarrow (restricts\_to\_model\ S\ b)$

The proofs of Goals $?_1$ and $?_2$ are similar so we shall only show the first of these. We expand the definition of *restricts_to_model* and then do an introduction:

H_2 : $\exists\, t : Binding.\ (restrict\ b\ S.1 = in2\ t) \wedge (S.2\ t)$

$?_3$: $\exists\, t : Binding.\ (restrict\ b\ T.1 = in2\ t) \wedge (T.2\ t)$

By existential elimination on H_2, we obtain a binding t which is the result of restricting b to $S.1$ and which satisfies the predicate $S.2$. We use this binding t as our witness to prove goal $?_3$. By doing and-introduction on the remaining goal we obtain:

$?_4$: *restrict* b $T.1 = in2$ t

$?_5$: $T.2$ t

Since t is obtained by restricting b to $S.1$, and $S.1$ equals $T.1$ by hypothesis H, we can prove goal $?_4$. To prove goal $?_5$, we observe that t is an exact model of S because it was obtained by restriction (*cf* Theorem 3). By hypothesis H_1, t is therefore an exact model, and hence a model, of T.

Now we consider the case where S and T have unequal signatures. Lemmas 36 and 37 show that in this, case, neither S nor T can have any restricting models. Hence they are equivalent under $Equiv_2$. □

A.2 Proofs of theorems in Chapter 5

A.2.1 The propositional operations

In this section, we shall assume that we have two schemas S and T with the following properties:

H : *Well_formed* S

H_1 : *Well_formed* T

H_2 : *type_compatible* $S.1$ $T.1$

Proof 2 And_preserves_well_formedness.

Goal: *Well_formed* (*And* S T)

We must show the following three goals:

$$? : \mathit{unique\_idents}\ (\mathit{And}\ S\ T).\mathbf{1}$$
$$?_1 : \mathit{Up\_closed}\ (\mathit{And}\ S\ T)$$
$$?_2 : \mathit{Down\_closed}\ (\mathit{And}\ S\ T)$$

Goal ? follows from the fact that *join* preserves *unique_idents* (Lemma 22). The proofs of goals $?_2$ and $?_3$ are very similar, so we shall only describe the first of these. We introduce a binding b and a hypothesis H_1 stating that b is a restricting model of $\mathit{And}\ S\ T$. By existential elimination on this we obtain:

$$t : \mathit{Binding}$$
$$H_3 : \mathit{restrict}\ b\ (\mathit{join}\ S.\mathbf{1}\ T.\mathbf{1}) = \mathit{in}\mathbf{2}\ t$$
$$H_4 : (S.\mathbf{2}\ t) \wedge (T.\mathbf{2}\ t)$$

We use and introduction to reduce the remaining goal to the following:

$$?_3 : S.\mathbf{2}\ b$$
$$?_4 : T.\mathbf{2}\ b$$

To prove goal $?_3$ we use the assumption that S is up-closed, so that it is sufficient to show that b is a restricting model of S:

$$?_5 : \exists\, b' : \mathit{Binding}.\ (\mathit{restrict}\ b\ S.\mathbf{1}) \wedge (S.\mathbf{2}\ b')$$

We can show that $\mathit{restrict}\ b\ S.\mathbf{1}$ is equal to $\mathit{restrict}\ t\ S.\mathbf{1}$ (Lemmas 27, 26, and 24.) By rewriting with this equality we reduce the goal to showing that t is a restricting model of S:

$$?_6 : \exists\, b' : \mathit{Binding}.\ (\mathit{restrict}\ t\ S.\mathbf{1}) \wedge (S.\mathbf{2}\ b')$$

We can prove this because we know that t is a model of S (hypothesis H_4 and that S is down-closed. This concludes the proof of goal $?_3$. Goal $?_4$ is similar, using the facts that T is up-closed and down-closed. We have therefore shown that $\mathit{And}\ S\ T$ is up-closed.

The proof of down-closure is very similar and is omitted. □

At this point we discharge the schemas S and T, and all the assumptions listed at the beginning of this section. This has the effect of causing all the theorems we have proved to be universally quantified over S and T, and to have the discharged assumptions added to their hypotheses. (We shall need these fully quantified theorems for the next proof.) We then make the same declarations afresh:

H : *Well_formed S*

H_1 : *Well_formed T*

H_2 : *type_compatible* $S.\mathbf{1}$ $T.\mathbf{1}$

Proof 3 And_restricting_model_commutes.

Goal:

$\forall\, b : Binding.\ (restricts\_to\_model\ (And\ S\ T)\ b) \Rightarrow$
$(restricts\_to\_model\ (And\ T\ S)\ b)$

By doing introductions we obtain the following proof state:

b : *Binding*

H_3 : *restricts_to_model* (*And S T*) *b*

? : *restricts_to_model* (*And T S*) *b*

Theorem 11 tells us that we can prove this goal by showing:

$?_1$: *restricts_to_model S b*

$?_2$: *restricts_to_model T b*

(We use the fact that the type-compatibility relationship is symmetric (Lemma 35).) Theorem 11 then enables us to obtain proofs of each of these goals from hypothesis H_3. □

Proof 4 Or_preserves_well_formedness.

Goal: *Well_formed* (*Or S T*)

Again, we know that *Or* preserves the *unique_idents* property because *join* preserves it (Lemma 22). The proofs that up-closure and down-closure are preserved are again very similar. We shall describe the latter this time. By doing introductions, and expanding definitions, we transform the proof state to the following:

$b : \mathit{Binding}$
$H_3 : (S.\mathbf{2}\ b) \vee (T.\mathbf{2}\ b)$
$H_4 : \mathit{succeeds}\ (\mathit{restrict}\ b\ (\mathit{Or}\ S\ T).\mathbf{1})$
$? : \exists\, b' : \mathit{Binding}.\ (\mathit{restrict}\ b\ (\mathit{Or}\ S\ T).\mathbf{1} = \mathit{in}\mathbf{2}\ b') \wedge ((S.\mathbf{2}\ b') \vee (T.\mathbf{2}\ b'))$

Hypothesis H_4 gives us a binding b' and a proof that this is obtained by restricting b to $(\mathit{Or}\ S\ T).\mathbf{1}$. We shall show that b' is a model of $\mathit{Or}\ S\ T$.

We proceed by doing an or elimination on H_3. In the first case:

$H_5 : S.\mathbf{2}\ b$

we shall prove the goal by proving the left hand disjunct:

$?_1 : S.\mathbf{2}\ b'$

Since S is up-closed, we can prove this by showing that b' is a restricting model of S.

$?_2 : \exists\, t : \mathit{Binding}.\ (\mathit{restrict}\ b'\ S.\mathbf{1} = \mathit{in}\mathbf{2}\ t) \wedge (S.\mathbf{2}\ t)$

Lemmas 28 and 29 enable us to show that $\mathit{restrict}\ b'\ S.\mathbf{1}$ equals $\mathit{restrict}\ b\ S.\mathbf{1}$. (We omit the details of this part of the proof.) Rewriting with this equality leaves us with the goal of showing that b is a restricting model of S. This follows from the facts that S is down-closed, and b is a model of S (hypothesis H_3).

In the case where b is a model of T, we show, by a similar argument, that b' is a model of T. This completes the proof of down-closure. □

Proof 5 Or_restricting_model_intro.

Goal:

$$\forall\, b\!:\!Binding.\ (succeeds\ (restrict\ b\ (Or\ S\ T).\mathbf{1})) \Rightarrow$$
$$((restricts\_to\_model\ S\ b) \vee (restricts\_to\_model\ T\ b) \Rightarrow$$
$$restricts\_to\_model\ (Or\ S\ T)\ b)$$

By doing introductions we arrive at the following proof state:

b : $Binding$

H_3 : $succeeds\ (restrict\ b\ (Or\ S\ T).\mathbf{1})$

H_4 : $(restricts\_to\_model\ S\ b) \vee (restricts\_to\_model\ T\ b)$

$?$: $restricts\_to\_model\ (Or\ S\ T)\ b$

Since $Or\ S\ T$ is down-closed our goal can be proved by showing:

$?_1$: $(S.\mathbf{2}\ b) \vee (T.\mathbf{2}\ b)$

We do or-elimination on hypothesis H_4. If b is a restricting model of S, then, since S is up-closed, we can prove $S.\mathbf{2}\ b$. Similarly, if b is a restricting model of T, we show $T.\mathbf{2}\ b$. □

Proof 6 Or_restricting_model_elim.

Goal:

$$\forall\, b\!:\!Binding.\ (restricts\_to\_model\ (Or\ S\ T)\ b) \Rightarrow$$
$$(restricts\_to\_model\ S\ b) \vee (restricts\_to\_model\ T\ b)$$

We introduce a binding b and a proof H_3 that b is a restricting model of $Or\ S\ T$. Since $Or\ S\ T$ is up-closed, b is also a model of $Or\ S\ T$. We shall prove the goal by doing or-elimination on this fact. In the first case, we know that b is a model of S:

H_4 : $S.\mathbf{2}\ b$

$?$: $(restricts\_to\_model\ S\ b) \vee (restricts\_to\_model\ T\ b)$

We use the fact that S is down-closed to prove the left hand disjunct. (We need Lemma 27 to show that b can be restricted to the signature of S.)

Similarly, in the case where we know that b is a model of T, we prove the right hand disjunct of Goal ?, using Lemma 26 and the assumption that T is down-closed. □

Once more, we discharge and then reintroduce our global assumptions (S, T, H, H_1, and H_2).

Proof 7 Or_restricting_model_commutes.

Goal:

$\forall\, b : \mathit{Binding}\textbf{.}\ (\mathit{restricts\_to\_model}\ (\mathit{Or}\ S\ T)\ b \Rightarrow \mathit{restricts\_to\_model}\ (\mathit{Or}\ T\ S)\ b)$

We introduce a binding b and a proof H_3 that b is a restricting model of $\mathit{Or}\ S\ T$. Theorem 20 (Proof 6) enables us to split our proof into two cases. In the first case, the proof state is as follows:

H_4 : $\mathit{restricts\_to\_model}\ S\ b$

? : $\mathit{restricts\_to\_model}\ (\mathit{Or}\ T\ S)$

We use Theorem 19 (Proof 5) to prove this goal. In order to satisfy the conditions for this theorem, we need Lemma 35 (to show that $T\textbf{.1}$ and $S\textbf{.1}$ are type-compatible) and Lemmas 23, 26 and 27 (to show that b is capable of being restricted to the signature $(\mathit{Or}\ T\ S)\textbf{.1}$.

The proof of the second case is similar to that of the first. □

Proof 8 Imply_preserves_well_formedness.

Goal: $\mathit{Well\_formed}\ (\mathit{Imply}\ S\ T)$

We shall only describe the proof that $\mathit{Imply}\ S\ T$ is up-closed. By expanding definitions, and doing introductions and eliminations, we transform the proof context to:

b, t : $\mathit{Binding}$

H_3 : $\mathit{restrict}\ b\ (\mathit{Imply}\ S\ T)\textbf{.1} = \mathit{in}\textbf{2}\ t$

H_4 : $(S\textbf{.2}\ t) \Rightarrow (T\textbf{.2}\ t)$

H_5 : $S\textbf{.2}\ b$

? : $T\textbf{.2}\ b$

Since T is up-closed, we can prove the goal by showing that b is a restricting model of T:

$$?_1 : \exists\, b' : Binding.\ (restrict\ b\ T.\mathbf{1} = in\mathbf{2}\ b') \wedge (T.\mathbf{2}\ b')$$

Lemmas 25, 27 and 26 allow us to show that $restrict\ \ b\ \ T.\mathbf{1}$ is equal to $restrict\ \ t\ \ T.\mathbf{1}$. We can therefore prove the goal by showing that t is a restricting model of T:

$$?_2 : \exists\, b' : Binding.\ (restrict\ t\ T.\mathbf{1} = in\mathbf{2}\ b') \wedge (T.\mathbf{2}\ b')$$

Since T is down-closed, we can prove this by showing:

$$?_3 : T.\mathbf{2}\ t$$

Hypothesis H_4 enables us to prove this by showing:

$$?_4 : S.\mathbf{2}\ t$$

Since S is up-closed, we can prove this by showing that t is a restricting model of S. We use Lemmas 24, 27 and 26 to show that $restrict\ t\ S.\mathbf{1}$ equals $restrict\ b\ S.\mathbf{1}$. Rewriting with this transforms our goal to:

$$?_5 : restricts\_to\_model\ S\ b$$

Since S is down-closed, and we know (hypothesis H_5) that b is a model of S, the proof is complete. □

Proof 9 Imply_restricting_model_intro.

Goal:

$$\forall\, b : Binding.\ (succeeds\ (restrict\ b\ (Imply\ S\ T).\mathbf{1})) \Rightarrow$$
$$((restricts\_to\_model\ S\ b) \Rightarrow (restricts\_to\_model\ T\ b)) \Rightarrow$$
$$(restricts\_to\_model\ (Imply\ S\ T)\ b)$$

We introduce hypotheses to obtain the following proof state:

$b : Binding$

$H_3 : succeeds\ (restrict\ b\ (Imply\ S\ T).\mathbf{1})$

$H_4 : (restricts\_to\_model\ S\ b) \Rightarrow (restricts\_to\_model\ T\ b)$

$? : restricts\_to\_model\ (Imply\ S\ T)\ b$

Hypothesis H_3 enables us to obtain a binding t which is the result of restricting b to the signature of *Imply* S T. Our goal is reduced to showing that t is a model of *Imply* S T:

H_5 : $S.\mathbf{2}\ t$

$?_2$: $T.\mathbf{2}\ t$

Since T is up-closed, we can prove goal $?_2$ by showing that t is a restricting model of T. Lemmas 27, 26 and 25 enable us to show that *restrict* t $T.\mathbf{1}$ equals *restrict* b $T.\mathbf{1}$. So the goal can be rewritten as:

$?_6$: *restricts_to_model* T b

To prove this we use hypothesis H_4. Now we must show that b is a restricting model of S. Lemmas 27, 26 and 24 can be used to prove that *restrict* b $S.\mathbf{1}$ equals *restrict* t $S.\mathbf{1}$. The goal then becomes:

$?_7$: *restricts_to_model* S t

This follows from hypothesis H_5 and the assumption that S is down-closed. □

Proof 10 Not_up_closed.

Goal: *Well_formed* (*Not* S)

By doing introductions we transform the proof state to:

b : *Binding*

H_3 : *restricts_to_model* (*Not* S) b

$?$: $(S.\mathbf{2}\ b) \Rightarrow$ *absurd*

Hypothesis H implies that restricting b to the signature (*Not* S)$.\mathbf{1}$, which is the same as $S.\mathbf{1}$, yields a binding t and a proof:

H_4 : $(S.\mathbf{2}\ t) \Rightarrow$ *absurd*

To prove goal $?$, we introduce its antecedent, and then use hypothesis H_4 to prove the remaining goal. This gives us the following proof state:

H_5 : $S.\mathbf{2}\ b$

$?_1$: $S.\mathbf{2}\ t$

Since S is down-closed, H_5 implies that b is a restricting model of S. We know that the binding obtained by restricting b to $S.1$ is t. We can therefore conclude that t has the property $S.2$. □

Proof 11 Not_restricting_model_property1.

Goal:

$\forall\, b\!:\!Binding.\ (restricts\_to\_model\ S\ b) \Rightarrow \neg(restricts\_to\_model\ (Not\ S)\ b)$

By expanding the definition of *not* and doing introductions we obtain the following proof context:

b : *Binding*

H_3 : $restricts\_to\_model\ S\ b$

H_4 : $restricts\_to\_model\ (Not\ S)\ b$

$?$: *absurd*

Since *Not S* and S have the same signature, the same binding t is obtained by restricting b to each of them. Hypotheses H_3 and H_4 allow us to prove $S.2\ t$ and $\neg(S.2\ t)$, respectively. Hence we can prove *absurd*. □

Proof 12 Not_restricting_model_property4.

Goal:

$$\forall\, b\!:\!Binding.\ (succeeds\ (restrict\ b\ S.1)) \Rightarrow$$
$$\neg(restricts\_to\_model\ S\ b) \Rightarrow restricts\_to\_model\ (Not\ S)\ b$$

Doing introductions, and expanding the definition of $\neg$, gives us the following proof context:

b : *Binding*

H_3 : $succeeds\ (restrict\ b\ S.1)$

H_4 : $(restricts\_to\_model\ S\ b) \Rightarrow absurd$

$?$: $restricts\_to\_model\ (Not\ S)\ b$

From H_3 we obtain a binding t that is the result of restricting b to the signature of S. Since *Not S* has the same signature as S, our goal reduces to showing:

$?_1$: $(S.2\ t) \Rightarrow absurd$

To prove this we assume $S.\mathbf{2}\ t$. From this we can show that b is a restricting model of S, since we know that t is obtained by restricting b to the signature of S. Hypothesis H_4 then allows us to prove *absurd*. □

We discharge all our global assumptions at this point.

A.2.2 The hiding operations

Proof 13 Hide_preserves_well_formedness.

Goal:
$\forall\, s\!:\!Signature.\ \forall\, S\!:\!Schema.\ (Well\_formed\ S) \Rightarrow (Well\_formed\ (Hide\ s\ S))$

We introduce a signature s, a schema S, and a hypothesis H stating that S is well-formed. We must now show the following:

$? : unique\_idents\ (Hide\ S).\mathbf{1}$
$?_1 : Down\_closed\ (Hide\ S)$
$?_2 : Up\_closed\ (Hide\ S)$

It is easy to show that the *hide_sig* operation preserves the property *unique_idents*. The proofs of subgoals $?_1$ and $?_2$ are similar, so we shall only describe the latter. By expanding definitions, and doing introductions, we obtain the following proof state:

$b : Binding$
$H_1 : \exists\, b'\!:\!Binding.\ ((restrict\ b\ (hide\_sig\ s\ S.\mathbf{1})) = in\mathbf{2}\ b') \wedge$
$\quad (\exists\, b''\!:\!Binding.\ (is\_true\ (matches\ s\ b'')) \wedge (S.\mathbf{2}\ (join\_bin\ b''\ b')))$
$?_3 : \exists\, b'\!:\!Binding.\ (is\_true\ (matches\ s\ b')) \wedge (S.\mathbf{2}\ (join\_bin\ b'\ b))$

By existential elimination on hypothesis H_1, we obtain two bindings b' and b''. We use b'' as a witness in proving goal $?_3$. We must now show:

$?_4 : is\_true\ (matches\ s\ b'')$
$?_5 : S.\mathbf{2}\ (join\_bin\ b''\ b)$

We know from hypothesis H_1 that b'' matches s, so goal $?_4$ is taken care of. Since S is up-closed, we can prove goal $?_5$ by showing that $join\_bin\ b''\ b$ is a restricting model of S:

$$?_6 : \exists\, b_1 : Binding.\ (restrict\ (join\_bin\ b''\ b_1)\ S.1 = in2\ b_1) \wedge (S.2\ b_1)$$

We now use Lemma 33 which allows us to deduce from that $join\_bin\ b''\ b'$ can be successfully restricted to the signature of S. We then use the fact that S is down-closed to show that the binding $join\_bin\ b''\ b'$ is a restricting model of S. By existential elimination on this we get:

$$b_1 : Binding$$
$$H_2 : restrict\ (join\_bin\ b''\ b')\ S.1 = in2\ b_1$$
$$H_3 : S.2\ b_1$$

We use the binding b_1 as a witness for proving goal $?_6$. We know from hypothesis H_3 that b_1 satisfies the property of S. Lemma 34 enables us to prove the remaining subgoal. □

A.3 Include

Proof 14 Include_equals_And2.

Goal:

$$\forall\, S : Schema.\ \forall\, sig : Signature.\ \forall\, P : Predicate.$$
$$(unique\_idents\ S.1) \wedge (unique\_idents\ sig) \wedge (type\_compatible\ S.1\ sig) \Rightarrow$$
$$Include\ S\ sig\ P = And\ S\ (join\ S.1\ sig, P)$$

We introduce a schema S, a signature sig, a predicate P, and three hypotheses H, H_1, and H_2. We must now show that $Include\ S\ sig\ P$ is equal to $And\ S\ (join\ S.1\ sig, P)$. These two schemas have exactly the same predicate $(\lambda b : Binding.\ (S.2\ b) \wedge (P\ b))$, so all that is required is to prove that their signatures are the same. Lemma 19 shows that this is indeed the case. □

A.4 Proofs of theorems in Chapter 6

We assume that we have a well-formed schema, S which has a static signature:

S : *Schema*

H : *Up_closed* S

H_1 : *Down_closed* S

H_2 : *unique_idents* $S.\mathbf{1}$

H_3 : *static_sig* $S.\mathbf{1}$

Proof 15 Prime_down_closed.

Goal: *Down_closed* (*Prime* S)

By expanding definitions and doing introductions we transform the goal to:

b : *Binding*

H_4 : $S.\mathbf{2}$ (*post_bin* b)

H_5 : *succeeds* (*restrict* b (*Prime* S)$.\mathbf{1}$)

$? : \exists\, b_1 : \mathit{Binding}.\ (\mathit{restrict}\ b\ (\mathit{Prime}\ S).\mathbf{1}) \wedge (S.\mathbf{2}\ (\mathit{post\_bin}\ b_1))$

From H_5 we obtain a binding b_1 with at proof that this is equal to the restriction of b to the signature of *Prime* S. We shall show that *post_bin* b_1 satisfies the predicate $S.\mathbf{2}$. Since we know that S is down-closed, we can infer from hypothesis H_4 that *post_bin* b is a restricting model of S. Lemma 31 enables us to prove that *post_bin* b_1 is the binding obtained by restricting *post_bin* b to the signature of S. Since *post_bin* b is a restricting model of S, its restriction to $S.\mathbf{1}$ must satisfy the predicate of S. □

Proof 16 Prime_restricting_model_char.

Goal:

$$\forall\, b : \mathit{Binding}.$$
$$\mathit{restricts\_to\_model}\ S\ (\mathit{post\_bin}\ b) \iff \mathit{restricts\_to\_model}\ (\mathit{Prime}\ S)\ b$$

We introduce a binding b. First we must show that if *post_bin* b is a restricting model of S, then b is a restricting model of *Prime* S. We transform the proof state to:

b_1 : *Binding*

H_4 : *restrict* (*post_bin* b) $S.\mathbf{1} = in\mathbf{2}\ b_1$

H_5 : $S.\mathbf{2}\ b_1$

$? : \exists\, b_2 : (\mathit{restrict}\ b\ (\mathit{decorate\_sig}\ \mathit{pr}\ S.\mathbf{1}) = in\mathbf{2}\ b_2) \wedge (S.\mathbf{2}\ (\mathit{post\_bin}\ b_2))$.

We shall use the binding *decorate_bin pr* b_1 as the witness to prove the goal ?. Lemma 32 shows us that this binding is indeed obtained by restricting b to the signature *decorate_sig pr* $S.\mathbf{1}$. We are left to show:

$?_1 : S.\mathbf{2}\ (\mathit{post\_bin}\ (\mathit{decorate\_bin}\ \mathit{pr}\ b_1))$

Lemma 30 tells us that *decorate_bin* is the right inverse of *post_bin*, so this reduces to showing that b_1 satisfies the predicate $S.\mathbf{2}$. This is hypothesis H_5. This completes the forward half of the proof.

Next, we must show that if b is a restricting model of *Prime* S, then *post_bin* b is a restricting model of S. By means of introductions and eliminations we transform the proof state to the following:

b_1 : *Binding*

H_4 : *restrict* b (*Prime* S)$.\mathbf{1} = in\mathbf{2}\ b_1$

$H_5 : (\mathit{Prime}\ S).\mathbf{2}\ b_1 ? : \exists\, b_2 : \mathit{Binding}.\ (\mathit{restrict}\ (\mathit{post\_bin}\ b)\ S.\mathbf{1} = \mathit{in\_2}\ b_2) \wedge (S.\mathbf{2}\ b_2)$

We use the binding *post_bin* b_1 as the witness in proving this goal. Lemma 31 tells us that this is the binding obtained by restricting *post_bin* b to the signature of S. Hypothesis H_5 tells us that this binding has the property $S.\mathbf{2}$. □

Proof 17 Xi_preserves_well_formedness.

Goal: *Well_formed* (*Xi* S)

The *Xi* operation is defined in terms of *Include*. We shall prove the goal by using Theorem 41, which gives conditions for *Include* to produce well-formed schemas. The proof is reduced to the following subgoals:

$? : \mathit{Well\_formed}\ (\mathit{Delta}\ S)$

$?_1 : \mathit{type\_compatible}\ (\mathit{Delta}\ S).\mathbf{1}\ \mathit{nil\_sig}$

$?_2 : \mathit{Well\_formed}\ (\mathit{join}\ (\mathit{Delta}\ S.\mathbf{1})\ \mathit{nil\_sig},$
$\qquad\qquad \lambda b : \mathit{Binding}.\ \mathit{restrict}\ b\ S.\mathbf{1} = \mathit{restrict}\ (\mathit{post\_bin}\ b)\ S.\mathbf{1})$

To prove goal ? we use Theorem 48 which states that *Delta* preserves well-formedness. All signatures are type-compatible with *nil_sig*, so goal $?_1$ is proved. Since $S.\mathbf{1}$ has unique identifiers, and *Delta* preserves this property, we can show that the the signature $\mathit{join}\ (\mathit{Delta}\ S.\mathbf{1})\ \mathit{nil\_sig}$ also has unique identifiers. We are left with having to show down-closure and up-closure for the schema in Goal $?_2$. The proofs of both properties are similar, so we shall only describe the proof of down-closure.

Since $(\mathit{Delta}\ S).\mathbf{1}$ has unique identifiers, we can use Lemma 20 to simplify the signature of the schema. We then do introductions and eliminations to obtain the following proof context:

$b, b_1 : \mathit{Binding}$

$H_4 : \mathit{restrict}\ (\mathit{post\_bin}\ b)\ S.\mathbf{1} = \mathit{restrict}\ b\ S.\mathbf{1}$

$H_5 : \mathit{restrict}\ b\ (\mathit{Delta}\ S).\mathbf{1} = \mathit{in}\mathbf{2}\ b_1$

$? : \exists\, b_2 : \mathit{Binding}.\ (\mathit{restrict}\ b\ (\mathit{Delta}\ S).\mathbf{1} = \mathit{in}\mathbf{2}\ b_2) \wedge$
$\qquad\qquad (\mathit{restrict}\ (\mathit{post\_bin}\ b_2)\ S.\mathbf{1} = \mathit{restrict}\ b_2\ S.\mathbf{1})$

We use the binding b_1 as a witness to prove this goal. The goal becomes:

$?_1 : \mathit{restrict}\ (\mathit{post\_bin}\ b_1)\ S.\mathbf{1} = \mathit{restrict}\ b_1\ S.\mathbf{1}$

We prove this by rewritings and other manipulations involving Lemmas 24, 27, 26, 31 and 25. □

Appendix B

Lemmas

B.1 Functions used in the main encoding

B.1.1 Lemmas about lookup

Lemma 5 lookup_x_equals_x.

$$\forall\, b : \mathit{Binding}.\ \forall\, s : \mathit{sig\_item}.\ \forall\, z : \mathit{Ztype}.\ \forall\, v : \mathit{Typ}\ z.$$
$$(\mathit{lookup}\ s\ b = \mathit{in}\mathbf{2}\ (z, v)) \Rightarrow (s.\mathbf{2} = z)$$

Lemma 6 lookup_success_lemma.

$$\forall\, x : \mathit{sig\_item}.\ \forall\, b : \mathit{Binding}.\ (\mathit{succeeds}\ (\mathit{lookup}\ x\ b)) \Rightarrow$$
$$\exists\, y : \mathit{Typ}\ x.\mathbf{2}.\ \mathit{lookup}\ x\ b = (x.\mathbf{2}, y)$$

Lemma 7 lookup_member_equiv.

$$\forall\, x : \mathit{sig\_item}.\ \forall\, b : \mathit{Binding}.$$
$$(\mathit{succeeds}\ (\mathit{lookup}\ x\ b)) \iff (\mathit{is\_true}\ \mathit{member}\ \mathit{sig\_item\_eq}\ x\ (\mathit{extract\_sig}\ b))$$

Lemma 8 lookup_orig_is_lookup_undecorated_post.

$$\forall\, b : \mathit{Binding}.\ \forall\, x : \mathit{sig\_item}.$$
$$(\mathit{last\_decoration}\ x.\mathbf{1}.\mathbf{2} = \mathit{prime}) \Rightarrow$$
$$(\mathit{lookup}\ x\ b = \mathit{lookup}\ (\mathit{undecorate\_item}\ x)\ (\mathit{post\_bin}\ b))$$

B.1.2 Lemmas about restrict

Lemma 9 restrict_to_tail.

$$\forall\, sig : Signature.\ \forall\, x : sig\_item.\ \forall\, b : Binding.$$
$$succeeds\ (restrict\ b\ (cons\ x\ sig)) \Rightarrow succeeds\ (restrict\ b\ sig)$$

Lemma 10 restrict_equals_sig.

$$\forall\, sig : Signature.\ \forall\, b : Binding.$$
$$(case\ \boldsymbol{\lambda}\_ : Error.\ trueProp$$
$$\boldsymbol{\lambda} b' : Binding.\ extract\_sig\ b' = sig$$
$$(restrict\ b\ sig))$$

Lemma 11 member_restrict_implies_member_orig.

$$\forall\, sig : Signature.\ \forall\, b : Binding.\ \forall\, x : sig\_item.$$
$$case\ \boldsymbol{\lambda}\_ : Error.\ trueProp$$
$$\boldsymbol{\lambda} b' : Binding.\ (is\_true\ (member\ x\ (extract\_sig\ b'))) \Rightarrow$$
$$(is\_true\ (member\ x\ (extract\_sig\ b)))$$
$$(restrict\ b\ sig)$$

Lemma 12 restrict_works_then_lookup_works.

$$\forall\, sig : Signature.\ \forall\, b : Binding.\ \forall\, x : bin\_item.$$
$$(is\_true\ (member\ sig\_item\_eq\ x\ sig)) \Rightarrow$$
$$(succeeds\ (restrict\ b\ sig)) \Rightarrow (succeeds\ (lookup\ x\ b))$$

Lemma 13 restrict_larger_binding.

$$\forall\, sig : Signature.\ \forall\, b : Binding.\ \forall\, x : bin\_item.$$
$$succeeds\ (restrict\ b\ S) \Rightarrow succeeds\ (restrict\ (cons\ x\ )\ S)$$

Lemma 14 restrict_own_sig.

$$\forall\, b : Binding.\ succeeds\ (restrict\ b\ (extract\_sig\ b))$$

Lemma 15 restrict_own_sig2.

$$\forall\, b\!:\!Binding.\ unique\_idents\ (extract\_sig\ b) \Rightarrow \\ restrict\ b\ (extract\_sig\ b) = in\mathbf{2}\ b$$

Lemma 16 restrict_matching_sig.

$$\forall\, sig\!:\!Signature.\ \forall\, b\!:\!Binding. \\ is\_true\ (matches\ sig\ b) \Rightarrow succeeds\ (restrict\ b\ sig)$$

Lemma 17 restriction_matches.

$$\forall\, sig\!:\!Signature.\ \forall\, b\!:\!Binding. \\ (succeeds\ (restrict\ b\ sig)) \Rightarrow is\_true\ (case\ \boldsymbol{\lambda}\_\!:\!Error.\,false \\ \boldsymbol{\lambda}b'\!:\!Binding.\ matches\ sig\ b' \\ (restrict\ b\ sig))$$

Lemma 18 lookup_restrict_equals_lookup_orig.

$$\forall\, Sig\!:\!Signature.\ \forall\, x\!:\!sig\_item.\ \forall\, b, c\!:\!Binding. \\ (restrict\ b\ Sig = in\mathbf{2}\ c) \Rightarrow \\ (is\_true\ (member\ sig\_item\_eq\ x\ Sig)) \Rightarrow \\ lookup\ x\ b = lookup\ x\ c$$

B.1.3 Lemmas about join

Lemma 19 join_twice_left.

$$\forall\, sig, sig'\!:\!Signature.\ (type\_compatible\ sig\ sig') \Rightarrow \\ (unique\_idents\ sig) \Rightarrow (unique\_idents\ sig') \Rightarrow \\ join\ sig\ (join\ sig\ sig') = join\ sig\ sig'$$

Lemma 20 join_s_nil.

$$\forall\, sig\!:\!Signature.\ (unique\_idents\ sig) \Rightarrow join\ s\ nil\_sig = s$$

Lemma 21 join_cons_cases.

$$\begin{array}{l} \forall\, sig, sig' : Signature.\ \forall\, x : sig\_item. \\ \quad (join\ (cons\ x\ sig)\ sig' = join\ sig\ sig') \lor \\ \quad (join\ (cons\ x\ sig)\ sig' = cons\ x\ (join\ sig\ sig')) \end{array}$$

Lemma 22 join_has_unique_idents.

$$\begin{array}{l} \forall\, sig, sig' : Signature.\ (unique\_idents\ sig) \land (unique\_idents\ sig') \Rightarrow \\ \qquad\quad (unique\_idents\ (join\ sig\ sig')) \end{array}$$

Lemma 23 restrict_join.

$$\begin{array}{l} \forall\, sig, sig' : Signature.\ \forall\, b : Binding. \\ \quad (succeeds\ (restrict\ b\ sig)) \Rightarrow (succeeds\ (restrict\ b\ sig')) \Rightarrow \\ \qquad succeeds\ (restrict\ b\ (join\ sig\ sig')) \end{array}$$

Lemma 24 restrict_join_l.

$$\begin{array}{l} \forall\, sig, sig' : Signature.\ \forall\, b : Binding. \\ \quad (type\_compatible\ sig\ sig') \Rightarrow \\ \quad (unique\_idents\ sig) \Rightarrow (unique\_idents\ sig') \Rightarrow \\ \quad (succeeds\ (restrict\ b\ sig)) \Rightarrow (succeeds\ (restrict\ b\ sig')) \Rightarrow \\ \qquad case\ (\boldsymbol{\lambda}\_ : Error.\ absurd) \\ \qquad\quad (\boldsymbol{\lambda} b' : Binding.\ restrict\ b'\ sig = restrict\ b\ sig) \\ \qquad\quad (restrict\ b\ (join\ sig\ sig')) \end{array}$$

Lemma 25 restrict_join_r.

$$\begin{array}{l} \forall\, sig, sig' : Signature.\ \forall\, b : Binding. \\ \quad (unique\_idents\ sig) \Rightarrow (unique\_idents\ sig') \Rightarrow \\ \quad (succeeds\ (restrict\ b\ sig)) \Rightarrow (succeeds\ (restrict\ b\ sig')) \Rightarrow \\ \qquad case\ (\boldsymbol{\lambda}\_ : Error.\ absurd) \\ \qquad\quad (\boldsymbol{\lambda} b' : Binding.\ restrict\ b'\ sig' = restrict\ b\ sig') \\ \qquad\quad (restrict\ b\ (join\ sig\ sig')) \end{array}$$

Lemma 26 restrict_join_back_r.

$$\forall\, sig, sig' : Signature.\ \forall\, b : Binding.$$
$$(succeeds\ (restrict\ b\ (join\ sig\ sig'))) \Rightarrow succeeds\ (restrict\ b\ sig')$$

Lemma 27 restrict_join_back_l.

$$\forall\, sig, sig' : Signature.\ \forall\, b : Binding.$$
$$(type\_compatible\ sig\ sig') \Rightarrow$$
$$(unique\_idents\ sig) \Rightarrow (unique\_idents\ sig') \Rightarrow$$
$$(succeeds\ (restrict\ b\ (join\ sig\ sig'))) \Rightarrow succeeds\ (restrict\ b\ sig)$$

Lemma 28 restrict_twice_lemma.

$$\forall\, s : Signature.\ \forall\, b : Binding.\ (unique\_idents\ s) \Rightarrow$$
$$case\ (\lambda\_ : unit.\ trueProp)$$
$$(\lambda b' : Binding.\ restrict\ b'\ s = restrict\ b\ s)$$
$$(restrict\ b\ S)$$

Lemma 29 restrict_split_lemma.

$$\forall\, s, t : Signature.\ \forall\, b, b_1, b_2, b_3 : Binding.$$
$$(restrict\ b\ s = in2\ b_1) \Rightarrow$$
$$(restrict\ b_1\ t = in2\ b_3) \Rightarrow$$
$$(restrict\ b\ t = in2\ b_2) \Rightarrow$$
$$b_2 = b_3$$

B.1.4 Lemmas about post_bin

Lemma 30 post_bin_inverse_decorate_bin.

$$\forall\, b : Binding.\ post\_bin\ (decorate\_bin\ pr\ b) = b$$

Lemma 31 post_bin_restrict_decorated_lemma.

$$\forall\, sig : Signature.\ \forall\, b_1, b_2 : Binding.$$
$$(restrict\ b_1\ (decorate\_sig\ pr\ sig) = in2\ b_2) \Rightarrow$$
$$(restrict\ (post\_bin\ b)\ sig = in2\ (post\_bin\ b'))$$

Lemma 32 restrict_post_bin_decorate_lemma.

$$\forall\, sig : Signature.\ \forall\, b_1, b_2 : Binding.$$
$$(restrict\ (post\_bin\ b)\ sig = in2\ b_2) \Rightarrow$$
$$(restrict\ b_1\ (decorate\_sig\ pr\ sig) = in2\ (decorate\_bin\ pr\ b_2))$$

B.1.5 Lemmas about join_bin

Lemma 33 join_bin_lemma1.

$$\forall\, s, t : Signature.\ \forall\, b, b_1, b_2 : Binding.$$
$$(restrict\ b\ (hide\_sig\ s\ t) = in2\ b_1) \Rightarrow$$
$$(is\_true\ (matches\ s\ b_2)) \Rightarrow$$
$$(succeeds\ (restrict\ (join\_bin\ b_2\ b_1)\ t))$$

Lemma 34 join_bin_lemma2.

$$\forall\, s, t : Signature.\ \forall\, b, b_1, b_2, b_3 : Binding.$$
$$(restrict\ b\ (hide\_sig\ s\ t) = in2\ b_1) \Rightarrow$$
$$(restrict\ (join\_bin\ b_2\ b_1)\ t = in2\ b_3) \Rightarrow$$
$$(restrict\ (join\_bin\ b_2\ b)\ t = in2\ b_3)$$

B.1.6 Other lemmas

Lemma 35 type_compatible_sym.

$$\forall\, s, t : Signature.\ (unique\_idents\ s) \wedge (unique\_idents\ t) \Rightarrow$$
$$(type\_compatible\ s\ t \iff type\_compatible\ t\ s)$$

Lemma 36 unequal_sigs_thm$_1$.

$$\forall\, S, T : Schema.\ ((is\_false\ (sig\_eq\ S.1\ T.1)) \wedge (Equiv_1\ S\ T)) \Rightarrow$$
$$\forall\, b : Binding.\ \neg(exactly\_models\ S\ b)$$

Lemma 37 unequal_sigs_thm$_2$.

$$\forall\, S, T : \mathit{Schema}.\ ((\mathit{is\_false}\ (\mathit{sig\_eq}\ S.\mathbf{1}\ T.\mathbf{1})) \wedge (\mathit{Equiv}_1\ S\ T)) \Rightarrow \forall\, b : \mathit{Binding}.\ \neg(\mathit{exactly\_models}\ T\ b)$$

Lemma 38 restricting_models_restrict.

$$\forall\, S : \mathit{Signature}.\ \forall\, b : \mathit{Binding}.\ (\mathit{restricts\_to\_model}\ S\ b) \Rightarrow (\mathit{succeeds}\ (\mathit{restrict}\ b\ S.\mathbf{1}))$$

B.2 Sets and Functions

This file shows an extract from a library of lemmas about finite sets and relations encoded as lists. The lemmas shown are those that have been used in proving Theorem 51 of Chapter 7.

Assume $z, z' : \mathit{Ztype}; \mathit{eq} : (\mathit{Typ}\ z) \rightarrow (\mathit{Typ}\ z) \rightarrow \mathit{bool};$ $\mathit{eq}' : (\mathit{Typ}\ z') \rightarrow (\mathit{Typ}\ z') \rightarrow \mathit{bool}.$

Lemma 39 Set_eq_refl. $\forall\, s : \mathit{Typ}\ (\mathit{finset\_ty}\ z).\ \mathit{is\_true}\ (\mathit{Set\_eq}\ \mathit{eq}\ s\ s)$

Lemma 40 Set_eq_sym. $\forall\, s, t : \mathit{Typ}\ (\mathit{finset\_ty}\ z).\ (\mathit{Set\_eq}\ \mathit{eq}\ s\ t) = (\mathit{Set\_eq}\ \mathit{eq}\ t\ s)$

Lemma 41 Set_eq_trans. $\forall\, s, t, u : \mathit{Typ}\ (\mathit{finset\_ty}\ z).$
$(\mathit{is\_true}\ (\mathit{Set\_eq}\ \mathit{eq}\ s\ t)) \Rightarrow (\mathit{is\_true}\ (\mathit{Set\_eq}\ \mathit{eq}\ t\ u)) \Rightarrow (\mathit{is\_true}\ (\mathit{Set\_eq}\ \mathit{eq}\ s\ u))$

Lemma 42 Union_resp_Set_eq. $\forall\, s, t, u, v : \mathit{Typ}\ (\mathit{finset\_ty}\ z).$
$(\mathit{is\_true}\ (\mathit{Set\_eq}\ \mathit{eq}\ s\ t)) \Rightarrow (\mathit{is\_true}\ (\mathit{Set\_eq}\ \mathit{eq}\ u\ v)) \Rightarrow$
$(\mathit{is\_true}\ (\mathit{Set\_eq}\ \mathit{eq}\ (\mathit{Union}\ \mathit{eq}\ s\ u)\ (\mathit{Union}\ \mathit{eq}\ t\ v)))$

Lemma 43 Fun_eq_sym. $\forall\, s, t : \mathit{Typ}\ (\mathit{fun\_ty}\ z\ z').\ \mathit{Fun\_eq}\ \mathit{eq}\ \mathit{eq}'\ s\ t = \mathit{Fun\_eq}\ \mathit{eq}\ \mathit{eq}'\ t\ s$

Lemma 44 Dom_resp_Fun_eq. $\forall\, s, t : \mathit{Typ}\ (\mathit{fun\_ty}\ z\ z').$
$(\mathit{is\_true}\ (\mathit{Fun\_eq}\ \mathit{eq}\ \mathit{eq}'\ s\ t)) \Rightarrow (\mathit{is\_true}\ (\mathit{Set\_eq}\ \mathit{eq}\ (\mathit{Dom}\ s)\ (\mathit{Dom}\ t)))$

Lemma 45 Dom_Union_lemma. $\forall\, s, t : Typ\ (fun\_ty\ z\ z')$.
$(is\_true\ (Set\_eq\ eq\ (Dom\ (FunUnion\ eq\ eq'\ s\ t))\ (Union\ eq\ (Dom\ s)\ (Dom\ t))))$

Discharge z, z', eq, eq'.

Appendix C

Function definitions

C.1 General function definitions

This is the elimination rule for the inductive type *Ztype*.

Definition 54 Ztype_elim.

$$
\begin{aligned}
&\mathit{Ztype\_elim} : \mathbf{\Pi}\, F : \mathit{Ztype} \rightarrow \mathit{Type}.\\
&\quad (F\ \mathit{nat\_ty}) \rightarrow\\
&\quad (F\ \mathit{bool\_ty}) \rightarrow\\
&\quad (\mathbf{\Pi}\, x : \mathit{GivenType}.\, F\ (\mathit{given\_ty}\ x)) \rightarrow\\
&\quad (\mathbf{\Pi}\, z : \mathit{Ztype}.\, (F\ z) \rightarrow (F\ (\mathit{finset\_ty}\ z))) \rightarrow\\
&\quad (\mathbf{\Pi}\, z, z_1 : \mathit{Ztype}.\, (F\ z) \rightarrow (F\ z_1) \rightarrow (F\ (\mathit{prod\_ty}\ z\ z_1))) \rightarrow\\
&\qquad (\mathbf{\Pi}\, z : \mathit{Ztype}.\, F\ z)
\end{aligned}
$$

We define a reduced form of the elimination rule *Ztype_elim*.

Definition 55 Ztype_dep_enum.

$$
\begin{aligned}
&\mathit{Ztype\_dep\_enum} \stackrel{\mathit{def}}{=} \textit{[omitted]}\\
&: \mathbf{\Pi}\, F\!:\!\mathit{Ztype} \to \mathit{Type}.\\
&\quad (F\ \mathit{nat\_ty}) \to\\
&\quad (F\ \mathit{bool\_ty}) \to\\
&\quad (\mathbf{\Pi}\, x\!:\!\mathit{GivenType}.\, F\ (\mathit{given\_ty}\ x)) \to\\
&\quad (\mathbf{\Pi}\, z\!:\!\mathit{Ztype}.\, F\ (\mathit{finset\_ty}\ z)) \to\\
&\quad (\mathbf{\Pi}\, z, z_1\!:\!\mathit{Ztype}.\, F\ (\mathit{prod\_ty}\ z\ z_1)) \to\\
&\qquad (\mathbf{\Pi}\, z\!:\!\mathit{Ztype}.\, F\ z)
\end{aligned}
$$

Definition 56 Error.

$$
\begin{aligned}
&\mathit{Error} \stackrel{\mathit{def}}{=} \mathit{unit} : \mathit{Type}\\
&\mathit{error} \stackrel{\mathit{def}}{=} \mathit{void} : \mathit{Error}
\end{aligned}
$$

Definition 57 succeeds,fails.

$$
\begin{aligned}
&\mathit{succeeds} \stackrel{\mathit{def}}{=} \mathit{is\_in}\mathbf{1} \mid \mathit{Error} : \mathbf{\Pi}\, t \,|\, \mathit{Type}.\, (\mathit{Error} + t) \to \mathit{Prop}\\
&\mathit{fails} \stackrel{\mathit{def}}{=} \mathit{is\_in}\mathbf{2} \mid \mathit{Error} : \mathbf{\Pi}\, t \,|\, \mathit{Type}.\, (\mathit{Error} + t) \to \mathit{Prop}
\end{aligned}
$$

When applied to a signature item x, and a signature, *sig*, *mem_and_comp* returns a pair of booleans. The first of these is equal to *true* if the identifier $x.\mathbf{1}$ occurs in *sig*, and is *false* otherwise. The second boolean is equal to *false* if the first occurence of $x.\mathbf{1}$ in *sig* is paired with a *Ztype* that is unequal to $x.\mathbf{2}$, and is *true* otherwise.

Definition 58 mem_and_comp.

$$
\begin{aligned}
&\mathit{mem\_and\_comp} \stackrel{\mathit{def}}{=} \boldsymbol{\lambda} x\!:\!\mathit{sig\_item}.\\
&\quad \mathit{list\_rec}\ (\mathit{false}, \mathit{true})\\
&\qquad\qquad (\boldsymbol{\lambda} h\!:\!\mathit{sig\_item}.\, \boldsymbol{\lambda}\_\!:\!\mathit{Signature}.\, \boldsymbol{\lambda} \mathit{prev}\!:\!\mathit{bool} \times \mathit{bool}.\\
&\qquad\qquad\quad \mathit{if}\ (\mathit{Ident\_eq}\ x.\mathbf{1}\ h.\mathbf{1})\\
&\qquad\qquad\qquad (\mathit{true}, \mathit{Ztype\_eq}\ x.\mathbf{2}\ h.\mathbf{2})\\
&\qquad\qquad\qquad \mathit{prev})\\
&: \mathit{sig\_item} \to \mathit{Signature} \to (\mathit{bool} \times \mathit{bool})
\end{aligned}
$$

***Definition* 59 *type_compatible_fun*.**

$$
\begin{aligned}
&tcf\_aux \stackrel{def}{=} \boldsymbol{\lambda} s, t : Signature.\\
&\quad list\_iter\ (t, true)\\
&\qquad (\boldsymbol{\lambda} n : sig\_item.\ \boldsymbol{\lambda} prev : Signature \times bool.\\
&\qquad\ [tmp = mem\_and\_comp\ n\ prev.\mathbf{1}]\\
&\qquad\quad if\ tmp.\mathbf{1}\\
&\qquad\qquad (if\ tmp.\mathbf{2}\ prev\ (prev.\mathbf{1}, false))\\
&\qquad\qquad (cons\ n\ prev.\mathbf{1}, prev.\mathbf{2}))
\end{aligned}
$$

$$
\begin{aligned}
&type\_compatible\_fun \stackrel{def}{=} \boldsymbol{\lambda} s, t : Signature.\ (tcf\_aux\ s\ t).\mathbf{2}\\
&: Signature \longrightarrow Signature \longrightarrow bool
\end{aligned}
$$

***Definition* 60 *hide_sig*.**

$$
\begin{aligned}
&hide\_sig \stackrel{def}{=} \boldsymbol{\lambda} sig : Signature.\\
&\quad list\_iter\ nil\_sig\\
&\qquad (\boldsymbol{\lambda} h : sig\_item.\ \boldsymbol{\lambda} prev : Signature.\\
&\qquad\qquad if\ (member\ sig\_item\_eq\ h\ sig)\ prev\ (cons\ h\ prev))\\
&: Signature \longrightarrow Signature \longrightarrow Signature
\end{aligned}
$$

***Definition* 61 *remove_occurs*.**

$$
\begin{aligned}
&remove\_occurs \stackrel{def}{=} \boldsymbol{\lambda} sig : Signature.\\
&\quad list\_rec\ nil\_bin\\
&\qquad (\boldsymbol{\lambda} h : bin\_item.\ \boldsymbol{\lambda}\_, prev : Binding.\\
&\qquad\qquad if\ (member\ sig\_item\_ident\_eq\ h.\mathbf{1}\ sig)\ prev\ (cons\ h\ prev))\\
&: Signature \longrightarrow Binding \longrightarrow Binding
\end{aligned}
$$

Definition* 62 *Apply.

$$\begin{aligned}
&Apply \stackrel{def}{=} \mathbf{\Pi}\, z_1, z_2 \,|\, Ztype.\, \mathbf{\Pi}\, eq : (Typ\ z_1) \to (Typ\ z_2) \to bool.\\
&\qquad \boldsymbol{\lambda} r : Typ\ (Rel\ z_1\ z_2).\, \boldsymbol{\lambda} x : Typ\ z_1.\\
&\qquad\quad list_iter\ (in\mathbf{1}\ error)\\
&\qquad\qquad\quad (\boldsymbol{\lambda} h : Typ\ (prod_ty\ z_1\ z_2).\, \boldsymbol{\lambda} prev : Error + (Typ\ z_2).\\
&\qquad\qquad\qquad if\ (eq\ x\ h.\mathbf{1})\ (in\mathbf{2}\ h.\mathbf{2})\ prev)\\
&\qquad\qquad\quad r\\
&: \mathbf{\Pi}\, z, z' \,|\, Ztype.\, \mathbf{\Pi}\, eq : (Typ\ z) \to (Typ\ z) \to bool.\\
&\quad (Typ\ (Rel\ z\ z')) \to (Typ\ z) \to (Error + (Typ\ z'))
\end{aligned}$$

Definition* 63 *mk_sum_bin_item, msbi.

$$\begin{aligned}
&mk_sum_bin_item \stackrel{def}{=} \boldsymbol{\lambda} i : Ident.\, case\ (\boldsymbol{\lambda}_ : Error.\, in\mathbf{1}\ error)\\
&\qquad (\boldsymbol{\lambda} n : small_item.\, in\mathbf{2}\ ((i, n.\mathbf{1}), n.\mathbf{2}))\\
&: (Error + small_item) \to (Error + small_item)\\
&msbi \stackrel{def}{=} mk_sum_bin_item
\end{aligned}$$

C.2 Wrapper functions

In this section we explain how to define the wrapper functions referred to in Section 3.5.6. We first define a more general version of the type *small_item*:

Definition* 64 *pre_small_item.

$$\begin{aligned}
&pre_small_item \stackrel{def}{=} \boldsymbol{\lambda} F : Ztype \to Ztype.\, (\mathbf{\Sigma}\, z : Ztype.\, Typ\ (F\ z))\\
&: (Ztype \to Ztype) \to Type
\end{aligned}$$

Next, we define wrappers specific to each of the various constructors for *Ztype*. As examples, we display the wrappers for *nat_ty* and for *finset_ty z*.

Definition 65 pre_nat_wrap.

$$
\begin{aligned}
& \mathit{pre\_nat\_wrap} \stackrel{\mathit{def}}{=} \boldsymbol{\lambda} F : \mathit{Ztype} \rightarrow \mathit{Ztype}.\ \boldsymbol{\lambda}\, \mathit{out} \mid \mathit{Type}. \\
& \quad \boldsymbol{\lambda} f : (\mathit{Typ}\ \ (F\ \mathit{nat\_ty})) \rightarrow (\mathit{Error} + \mathit{out}). \\
& \quad\quad \mathit{sigma\_rec} \\
& \quad\quad\quad (\mathit{Ztype\_dep\_enum} \\
& \quad\quad\quad\quad (\boldsymbol{\lambda} z : \mathit{Ztype}.\ (\mathit{Typ}\ \ (F\ z)) \rightarrow (\mathit{Error} + \mathit{out})) \\
& \quad\quad\quad\quad (\boldsymbol{\lambda} n : \mathit{Typ}\ \ (F\ \mathit{nat\_ty}).\ f\ n) \\
& \quad\quad\quad\quad (\boldsymbol{\lambda}\_ : \mathit{Typ}\ \ (F\ \mathit{bool\_ty}).\ \mathit{in}\mathbf{1}\ \mathit{error}) \\
& \quad\quad\quad\quad (\boldsymbol{\lambda} g : \mathit{GivenType}.\ \boldsymbol{\lambda}\_ : \mathit{Typ}\ \ (F\ \ (\mathit{given\_ty}\ g)).\ \mathit{in}\mathbf{1}\ \mathit{error}) \\
& \quad\quad\quad\quad (\boldsymbol{\lambda} z : \mathit{Ztype}.\ \boldsymbol{\lambda}\_ : \mathit{Typ}\ \ (F\ \ (\mathit{finset\_ty}\ z)).\ \mathit{in}\mathbf{1}\ \mathit{error}) \\
& \quad\quad\quad\quad (\boldsymbol{\lambda} z, z_1 : \mathit{Ztype}.\ \boldsymbol{\lambda}\_ : \mathit{Typ}\ \ (F\ \ (\mathit{prod\_ty}\ z\ z_1)).\ \mathit{in}\mathbf{1}\ \mathit{error})) \\
& :\ \boldsymbol{\Pi} F : \mathit{Ztype} \rightarrow \mathit{Ztype}.\ \boldsymbol{\Pi}\, \mathit{out} \mid \mathit{Type}. \\
& \quad ((\mathit{Typ}\ \ (F\ \mathit{nat\_ty})) \rightarrow (\mathit{Error} + \mathit{out})) \rightarrow ((\mathit{pre\_small\_item}\ F) \rightarrow (\mathit{Error} + \mathit{out}))
\end{aligned}
$$

Definition 66 pre_finset_wrap.

$$
\begin{aligned}
& \mathit{pre\_finset\_wrap} \stackrel{def}{=} \\
& \boldsymbol{\lambda} z : \mathit{Ztype}. \\
& \boldsymbol{\lambda} \mathit{z\_wrap} : \mathbf{\Pi}\, F : \mathit{Ztype} \rightarrow \mathit{Ztype}.\ \mathbf{\Pi}\, \mathit{out} : \mathit{Type}. \\
& \qquad ((\mathit{Typ}\ (F\ z)) \rightarrow (\mathit{Error} + \mathit{out})) \rightarrow ((\mathit{pre\_small\_item}\ F) \rightarrow (\mathit{Error} + \mathit{out})). \\
& \boldsymbol{\lambda} F : \mathit{Ztype} \rightarrow \mathit{Ztype}.\ \boldsymbol{\lambda}\, \mathit{out} \mid \mathit{Type}. \\
& \quad \boldsymbol{\lambda} f : (\mathit{Typ}\ (F\ (\mathit{finset\_ty}\ z))) \rightarrow (\mathit{Error} + \mathit{out}). \\
& \qquad \mathit{sigma\_rec} \\
& \qquad\quad (\mathit{Ztype\_dep\_enum} \\
& \qquad\qquad (\boldsymbol{\lambda} z : \mathit{Ztype}.\ (\mathit{Typ}\ (F\ z)) \rightarrow (\mathit{Error} + \mathit{out})) \\
& \qquad\qquad (\boldsymbol{\lambda} n : \mathit{Typ}\ (F\ \mathit{nat\_ty}).\ \mathit{in}1\ \mathit{error}) \\
& \qquad\qquad (\boldsymbol{\lambda}\_ : \mathit{Typ}\ (F\ \mathit{bool\_ty}).\ \mathit{in}1\ \mathit{error}) \\
& \qquad\qquad (\boldsymbol{\lambda} g : \mathit{GivenType}.\ \boldsymbol{\lambda}\_ : \mathit{Typ}\ (F\ (\mathit{given\_ty}\ g)).\ \mathit{in}1\ \mathit{error}) \\
& \qquad\qquad (\boldsymbol{\lambda} z_1 : \mathit{Ztype}.\ \boldsymbol{\lambda}\_ : \mathit{Typ}\ (F\ (\mathit{finset\_ty}\ z)). \\
& \qquad\qquad\qquad \mathit{z\_wrap}\ (\boldsymbol{\lambda} x : \mathit{Ztype}.\ F\ (\mathit{finset\_ty}\ x))\ f\ (z_1, x)) \\
& \qquad\qquad (\boldsymbol{\lambda} z_1, z_2 : \mathit{Ztype}.\ \boldsymbol{\lambda}\_ : \mathit{Typ}\ (F\ (\mathit{prod\_ty}\ z_1\ z_2)).\ \mathit{in}1\ \mathit{error})) \\
& :\ \mathbf{\Pi}\, z : \mathit{Ztype}. \\
& \quad \mathbf{\Pi}\, \mathit{z\_wrap} : \mathbf{\Pi}\, F : \mathit{Ztype} \rightarrow \mathit{Ztype}.\ \mathbf{\Pi}\, \mathit{out} : \mathit{Type}. \\
& \qquad ((\mathit{Typ}\ (F\ z)) \rightarrow (\mathit{Error} + \mathit{out})) \rightarrow ((\mathit{pre\_small\_item}\ F) \rightarrow (\mathit{Error} + \mathit{out})). \\
& \quad \mathbf{\Pi}\, F : \mathit{Ztype} \rightarrow \mathit{Ztype}.\ \mathbf{\Pi}\, \mathit{out} \mid \mathit{Type}. \\
& \qquad ((\mathit{Typ}\ (F\ \mathit{nat\_ty})) \rightarrow (\mathit{Error} + \mathit{out})) \rightarrow ((\mathit{pre\_small\_item}\ F) \rightarrow (\mathit{Error} + \mathit{out}))
\end{aligned}
$$

The other wrappers, *pre_bool_wrap*, *pre_given_wrap*, *pre_fun_wrap*, and *pre_prod_wrap* are defined in a similar manner.

We then define a general wrapper, which works for all *Ztypes*, which uses the elimination rule on *Ztypes* to select the correct specific wrapper to be applied in a given case.

Definition 67 pre_wrap.

$$
\begin{aligned}
& \mathit{pre\_wrap} \stackrel{\text{def}}{=} \mathit{Ztype\_elim} \\
& \qquad (\boldsymbol{\lambda} z : \mathit{Ztype}.\ \boldsymbol{\lambda} F : \mathit{Ztype} \rightarrow \mathit{Ztype}.\ \boldsymbol{\lambda}\ \mathit{out} \mid \mathit{Type}. \\
& \qquad\quad ((\mathit{Typ}\ \ (F\ z)) \rightarrow (\mathit{Error} + \mathit{out})) \rightarrow (\mathit{pre\_small\_item}\ F) \rightarrow (\mathit{Error} + \mathit{out})) \\
& \qquad \mathit{pre\_nat\_wrap} \\
& \qquad \mathit{pre\_bool\_wrap} \\
& \qquad \mathit{pre\_given\_wrap} \\
& \qquad \mathit{pre\_finset\_wrap} \\
& \qquad \mathit{pre\_prod\_wrap} \\
& :\ \boldsymbol{\Pi}\, z : \mathit{Ztype}.\ \boldsymbol{\Pi}\, F : \mathit{Ztype} \rightarrow \mathit{Ztype}.\ \boldsymbol{\Pi}\ \mathit{out} \mid \mathit{Type}. \\
& \qquad ((\mathit{Typ}\ \ (F\ z)) \rightarrow (\mathit{Error} + \mathit{out})) \rightarrow ((\mathit{pre\_small\_item}\ F) \rightarrow (\mathit{Error} + \mathit{out}))
\end{aligned}
$$

The general wrapper, *pre_wrap*, is then used to define various wrapper functions for specific situations. For example, the following is the wrapper which is commonly used for unary functions on *Ztype*s.

Definition 68 wrap.

$$
\begin{aligned}
& \mathit{wrap} \stackrel{\text{def}}{=} \boldsymbol{\lambda} z, z_1 : \mathit{Ztype}.\ \boldsymbol{\lambda} f : (\mathit{Typ}\ \ z) \rightarrow (\mathit{Typ}\ \ z_1). \\
& \qquad \mathit{case}\ \ (\boldsymbol{\lambda}\_ : \mathit{Error}.\ \mathit{in}\mathbf{1}\ \ \mathit{error}) \\
& \qquad\quad (\mathit{sigma\_rec} \\
& \qquad\qquad (\boldsymbol{\lambda} z_2 : \mathit{Ztype}.\ \boldsymbol{\lambda} x : \mathit{Typ}\ \ z_2. \\
& \qquad\qquad\quad \mathit{pre\_wrap} \\
& \qquad\qquad\qquad z\ \ (\boldsymbol{\lambda} y : \mathit{Ztype}. y)\ \ (\boldsymbol{\lambda} y : \mathit{Typ}\ \ z.\ \mathit{in}\mathbf{2}\ \ (z_1, (f\ \ x)))\ \ (z_2, x))) \\
& :\ \boldsymbol{\Pi}\, z, z_1 : \mathit{Ztype}.\ ((\mathit{Typ}\ \ z) \rightarrow (\mathit{Typ}\ \ z_1)) \rightarrow \\
& \qquad ((\mathit{Error} + \mathit{small\_item}) \rightarrow (\mathit{Error} + \mathit{small\_item}))
\end{aligned}
$$

The next function is a more general wrapper for unary functions, Here, only the domain of the wrapped function is required to be a *Ztype*.

Definition 69 wrap_1.

$$
\begin{aligned}
& \mathit{wrap}_1 \overset{\mathit{def}}{=} \boldsymbol{\lambda} z : \mathit{Ztype}.\ \boldsymbol{\lambda}\, \mathit{out} \mid \mathit{Type}.\ \boldsymbol{\lambda} f : (\mathit{Typ}\ z) \rightarrow \mathit{out}. \\
& \qquad \mathit{case}\ (\boldsymbol{\lambda}\_ : \mathit{Error}.\ \mathit{in}\mathbf{1}\ \mathit{error}) \\
& \qquad\qquad (\mathit{sigma\_rec} \\
& \qquad\qquad\quad (\boldsymbol{\lambda} z_1 : \mathit{Ztype}.\ \boldsymbol{\lambda} x : \mathit{Typ}\ z_1. \\
& \qquad\qquad\qquad \mathit{pre\_wrap}\ z\ (\boldsymbol{\lambda} y : \mathit{Ztype}. y)\ (\boldsymbol{\lambda} y : \mathit{Typ}\ z.\ \mathit{in}\mathbf{2}\ (f\ x))\ (z_1, x))) \\
& : \boldsymbol{\Pi}\, z : \mathit{Ztype}.\ \boldsymbol{\Pi}\, \mathit{out} \mid \mathit{Type}.\ ((\mathit{Typ}\ z) \rightarrow \mathit{out}) \rightarrow \\
& \qquad ((\mathit{Error} + \mathit{small\_item}) \rightarrow (\mathit{Error} + \mathit{out}))
\end{aligned}
$$

Next, we define a wrapper for *partial*, unary functions on *Ztype*s. Failure of the wrapped function, when applied to an argument of the correct *Ztype*, is identified with the case where the wrapped function is applied to an argument of the wrong *Ztype*.

Definition 70 wrap₂.

$$
\begin{aligned}
& \mathit{wrap}_2 \overset{\mathit{def}}{=} \boldsymbol{\lambda} z, z_1 : \mathit{Ztype}.\ \boldsymbol{\lambda} f : (\mathit{Typ}\ z) \rightarrow (\mathit{Error} + (\mathit{Typ}\ z_1)). \\
& \quad \mathit{case}\ (\boldsymbol{\lambda}\_ : \mathit{Error}.\ \mathit{in}\mathbf{1}\ \mathit{error}) \\
& \qquad (\mathit{sigma\_rec}\ (\boldsymbol{\lambda} z_2 : \mathit{Ztype}.\ \boldsymbol{\lambda} x : \mathit{Typ}\ z_2. \\
& \qquad\qquad\qquad \mathit{pre\_wrap}\ z \\
& \qquad\qquad\qquad\qquad (\boldsymbol{\lambda} y : \mathit{Ztype}. y) \\
& \qquad\qquad\qquad\qquad (\boldsymbol{\lambda} y : \mathit{Typ}\ z.\ \mathit{case}\ (\boldsymbol{\lambda}\_ : \mathit{Error}.\ \mathit{in}\mathbf{1}\ \mathit{error}) \\
& \qquad\qquad\qquad\qquad\qquad (\boldsymbol{\lambda} v : \mathit{Typ}\ z_1.\ \mathit{in}\mathbf{2}\ (z\mathbf{1}, v)) \\
& \qquad\qquad\qquad\qquad\qquad (f\ y)) \\
& \qquad\qquad\qquad\qquad (z_2, x))) \\
& : \boldsymbol{\Pi}\, z, z_1 : \mathit{Ztype}.\ ((\mathit{Typ}\ z) \rightarrow (\mathit{Error} + (\mathit{Typ}\ z_1))) \rightarrow \\
& \qquad ((\mathit{Error} + \mathit{small\_item}) \rightarrow (\mathit{Error} + \mathit{small\_item}))
\end{aligned}
$$

Next we define a wrapper for predicates over *Ztype*s (i.e. functions of type $(\mathit{Typ}\ z) \rightarrow \mathit{Prop}$, for some *Ztype*, z). This wrapper returns the value *absurd* if the wrapped predicate is applied to an argument of the wrong *Ztype*.

Definition 71 Pwrap.

$$
\begin{aligned}
&Pwrap \stackrel{def}{=} \boldsymbol{\lambda} z : Ztype.\ \boldsymbol{\lambda} P : (Typ\ z) \rightarrow Prop.\ \boldsymbol{\lambda} x : Error + small\_item. \\
&\quad case\ (\boldsymbol{\lambda}\_ : Error.\ absurd \\
&\qquad (\boldsymbol{\lambda} p : Prop.\ p) \\
&\qquad (wrap_1\ z\ P\ x)) \\
&: \boldsymbol{\Pi}\, z : Ztype.\ ((Typ\ z) \rightarrow Prop) \rightarrow ((Error + small\_item) \rightarrow Prop)
\end{aligned}
$$

Finally, we define wrappers for two-argument and three-argument functions on *Ztype*s.

Definition 72 double_wrap.

$$
\begin{aligned}
&double\_wrap \stackrel{def}{=} \boldsymbol{\lambda} z, z_1, z_2 : Ztype. \\
&\quad \boldsymbol{\lambda} f : (Typ\ z) \rightarrow (Typ\ z_1) \rightarrow (Typ\ z_2). \\
&\quad \boldsymbol{\lambda} x, y : Error + small\_item. \\
&\qquad case\ (\boldsymbol{\lambda}\_ : Error.\ in1\ error) \\
&\qquad\quad (\boldsymbol{\lambda} g : (Typ\ z_1) \rightarrow (Typ\ z_2).\ wrap\ z_1\ z_2\ g\ y) \\
&\qquad\quad (wrap_1\ z\ f\ x) \\
&: \boldsymbol{\Pi}\, z, z_1, z_2 : Ztype.\ ((Typ\ z) \rightarrow (Typ\ z_1) \rightarrow (Typ\ z_2)) \rightarrow \\
&\quad ((Error + small\_item) \rightarrow (Error + small\_item) \rightarrow (Error + small\_item))
\end{aligned}
$$

Definition 73 triple_wrap.

$$
\begin{aligned}
&triple\_wrap \stackrel{def}{=} \boldsymbol{\lambda} z, z_1, z_2, z_3 : Ztype. \\
&\quad \boldsymbol{\lambda} f : (Typ\ z) \rightarrow (Typ\ z_1) \rightarrow (Typ\ z_2) \rightarrow (Typ\ z_3). \\
&\quad \boldsymbol{\lambda} w, x, y : Error + small\_item. \\
&\qquad case\ (\boldsymbol{\lambda}\_ : Error.\ in1\ error) \\
&\qquad\quad (\boldsymbol{\lambda} g : (Typ\ z_1) \rightarrow (Typ\ z_2) \rightarrow (Typ\ z_3).\ double\_wrap\ z_1\ z_2\ z_3\ g\ x\ y) \\
&\qquad\quad (wrap_1\ z\ f\ x) \\
&: \boldsymbol{\Pi}\, z, z_1, z_2, z_3 : Ztype.\ ((Typ\ z) \rightarrow (Typ\ z_1) \rightarrow (Typ\ z_2) \rightarrow (Typ\ z_3)) \rightarrow \\
&\quad ((Error + small\_item) \rightarrow (Error + small\_item) \rightarrow \\
&\qquad (Error + small\_item) \rightarrow (Error + small\_item))
\end{aligned}
$$

C.2.1 Lemmas about wrapper functions

We show that when a wrapped version of a function is applied to an argument of the correct *Ztype*, the value returned is indeed that computed by the unwrapped version of the function. We prove similar results for all the wrappers.

Lemma 46. *wrap_lemma*

$$\begin{aligned}&\forall\, z, z_1 : Ztype.\ \forall f : (Typ\ z) \rightarrow (Typ\ z_1).\ \forall\, v : Typ\ z.\\ &\quad wrap\ z\ z_1\ f\ (in2\ (z, v)) = in2\ (z_1, (f\ v))\end{aligned}$$

Proof. Omitted.

C.2.2 Wrapped versions of functions

Definition 74 IS_TRUE.

$$IS\_TRUE \stackrel{def}{=} Pwrap\ bool\_ty\ is\_true$$
$$: (Error + small\_item) \rightarrow Prop$$

Definition 75 IS_FALSE.

$$IS\_FALSE \stackrel{def}{=} Pwrap\ bool\_ty\ is\_false$$
$$: (Error + small\_item) \rightarrow Prop$$

Definition 76 ALREADY_KNOWN.

$$ALREADY\_KNOWN \stackrel{def}{=} in2\ (given\_ty\ Report\_ty, already\_known)$$
$$: Error + small\_item$$

Definition 77 LT.

$$LT \stackrel{def}{=} double\_wrap\ nat\_ty\ nat\_ty\ bool\_ty\ lt$$
$$: (Error + small\_item) \rightarrow (Error + small\_item) \rightarrow (Error + small\_item)$$

Definition* 78 *PLUS.

$$\mathit{PLUS} \stackrel{\mathit{def}}{=} \mathit{double\_wrap}\ \mathit{nat\_ty}\ \mathit{nat\_ty}\ \mathit{nat\_ty}\ \mathit{plus}$$
$$:\ (\mathit{Error} + \mathit{small\_item}) \rightarrow (\mathit{Error} + \mathit{small\_item}) \rightarrow (\mathit{Error} + \mathit{small\_item})$$

Definition* 79 *NULL.

$$\mathit{NULL} \stackrel{\mathit{def}}{=} \boldsymbol{\lambda}\, z \,|\, \mathit{Ztype}\boldsymbol{.}\ \mathit{in}\mathbf{2}\ \ (\mathit{finset\_ty}\ z, \mathit{Null} \mid z)$$
$$:\ \boldsymbol{\Pi}\, z \,|\, \mathit{Ztype}\boldsymbol{.}\ \mathit{Error} + \mathit{small\_item}$$

Definition* 80 *SINGLE.

$$\mathit{SINGLE} \stackrel{\mathit{def}}{=} \boldsymbol{\lambda}\, z \,|\, \mathit{Ztype}\boldsymbol{.}\ \mathit{wrap}\ z\ \ (\mathit{finset\_ty}\ z)\ \ (\mathit{Single} \mid z)$$
$$:\ \mathit{Ztype} \rightarrow (\mathit{Error} + \mathit{small\_item}) \rightarrow (\mathit{Error} + \mathit{small\_item})$$

Definition* 81 *EQUAL.

$$\begin{aligned} &\mathit{EQUAL} \stackrel{\mathit{def}}{=} \boldsymbol{\lambda}\, z \,|\, \mathit{Ztype}\boldsymbol{.} \\ &\qquad \mathit{double\_wrap}\ z\ z\ \mathit{bool\_ty}\ \ (\mathit{Equal}\ z) \\ &:\ \boldsymbol{\Pi}\, \_ \,|\, \mathit{Ztype}\boldsymbol{.}\ (\mathit{Error} + \mathit{small\_item}) \rightarrow (\mathit{Error} + \mathit{small\_item}) \rightarrow (\mathit{Error} + \mathit{small\_item}) \end{aligned}$$

Definition* 82 *PAIR.

$$\begin{aligned} &\mathit{PAIR} \stackrel{\mathit{def}}{=} \boldsymbol{\lambda}\, z, z' \,|\, \mathit{Ztype}\boldsymbol{.} \\ &\qquad \mathit{double\_wrap}\ z\ z'\ \ (\mathit{prod\_ty}\ z\ z') \\ &\qquad\qquad (\boldsymbol{\lambda} x : \mathit{Typ}\ z\boldsymbol{.}\ \boldsymbol{\lambda} y : \mathit{Typ}\ z'\boldsymbol{.}\ (x, y)) \\ &:\ \boldsymbol{\Pi}\, z, z' \,|\, \mathit{Ztype}\boldsymbol{.}\ (\mathit{Error} + \mathit{small\_item}) \rightarrow (\mathit{Error} + \mathit{small\_item}) \rightarrow (\mathit{Error} + \mathit{small\_item}) \end{aligned}$$

Definition* 83 *UNION.

$$\begin{aligned} &\mathit{UNION} \stackrel{\mathit{def}}{=} \boldsymbol{\lambda}\, z \,|\, \mathit{Ztype}\boldsymbol{.}\ \boldsymbol{\lambda}\, \mathit{eq} \,|\, (\mathit{Typ}\ z) \rightarrow (\mathit{Typ}\ z) \rightarrow \mathit{bool}\boldsymbol{.} \\ &\qquad \mathit{double\_wrap}\ \ (\mathit{finset\_ty}\ z)\ \ (\mathit{finset\_ty}\ z)\ \ (\mathit{finset\_ty}\ z)\ \ (\mathit{Union}\ \mathit{eq}) \\ &:\ \boldsymbol{\Pi}\, z \,|\, \mathit{Ztype}\boldsymbol{.}\ ((\mathit{Typ}\ z) \rightarrow (\mathit{Typ}\ z) \rightarrow \mathit{bool}) \rightarrow \\ &\qquad (\mathit{Error} + \mathit{small\_item}) \rightarrow (\mathit{Error} + \mathit{small\_item}) \rightarrow (\mathit{Error} + \mathit{small\_item}) \end{aligned}$$

Definition 84 IN.

$$IN \stackrel{def}{=} \boldsymbol{\lambda}\, z \,|\, Ztype.\, \boldsymbol{\lambda}\, eq \,|\, (Typ\ z) \longrightarrow (Typ\ z) \longrightarrow bool.$$
$$double\_wrap\ z\ (finset\_ty\ z)\ bool\_ty\ (In\ eq)$$
$$: \boldsymbol{\Pi}\, z \,|\, Ztype.\, ((Typ\ z) \longrightarrow (Typ\ z) \longrightarrow bool) \longrightarrow$$
$$(Error + small\_item) \longrightarrow (Error + small\_item) \longrightarrow (Error + small\_item)$$

Definition 85 FUN_SINGLE.

$$FUN\_SINGLE \stackrel{def}{=} \boldsymbol{\lambda}\, z_1, z_2 \,|\, Ztype.$$
$$double\_wrap\ z_1\ z_2\ (fun\_ty\ z_1\ z_2)\ (FunSingle \mid z_1 \mid z \mid \mathbf{2})$$
$$: Ztype \longrightarrow Ztype \longrightarrow$$
$$(Error + small\_item) \longrightarrow (Error + small\_item) \longrightarrow (Error + small\_item)$$

Definition 86 FUN_UNION.

$$FUN\_UNION \stackrel{def}{=} \boldsymbol{\lambda}\, z_1, z_2 \,|\, Ztype.$$
$$\boldsymbol{\lambda}\, eq \,|\, (Typ\ z_1) \longrightarrow (Typ\ z_1) \longrightarrow bool.\, \boldsymbol{\lambda}\, eq \,|\, (Typ\ z_2) \longrightarrow (Typ\ z_2) \longrightarrow bool.$$
$$double\_wrap\ (fun\_ty\ z_1\ z_2)\ (fun\_ty\ z_1\ z_2)\ (fun\_ty\ z_1\ z_2)\ (FunUnion\ eq_1\ eq_2)$$
$$: \boldsymbol{\Pi}\, z_1, z_2 \,|\, Ztype.\, ((Typ\ z_1) \longrightarrow (Typ\ z_1) \longrightarrow bool) \longrightarrow ((Typ\ z_2) \longrightarrow (Typ\ z_2) \longrightarrow bool)$$
$$(Error + small\_item) \longrightarrow (Error + small\_item) \longrightarrow (Error + small\_item)$$

Definition 87 DOM.

$$DOM \stackrel{def}{=} \boldsymbol{\lambda}\, z_1, z_2 \,|\, Ztype.\, wrap\ (fun\_ty\ z_1\ z_2)(finset\_ty\ z_1)\ (Dom \mid z_1 \mid z_2)$$
$$: Ztype \longrightarrow Ztype \longrightarrow$$
$$(Error + small\_item) \longrightarrow (Error + small\_item) \longrightarrow (Error + small\_item)$$

Definition 88 FOR.

$$FOR \stackrel{def}{=} \boldsymbol{\lambda}\, t \,|\, Type.\, \boldsymbol{\lambda} x : t.\, \boldsymbol{\lambda} f : nat \longrightarrow t \longrightarrow t.\, wrap_1\ nat\_ty\ (nat\_rec\ x\, f)$$
$$: \boldsymbol{\Pi}\, t \,|\, Type.\, t \longrightarrow (nat \longrightarrow t \longrightarrow t) \longrightarrow (Error + small\_item) \longrightarrow (Error + t)$$

Definition 89 NIL_BIN.

$$NIL\_BIN \stackrel{def}{=} in\mathbf{2}\ nil\_bin$$
$$: Error + Binding$$

Definition* 90 *PAIR_BIN.

$$\begin{aligned}
&\mathit{PAIR_BIN} \stackrel{\mathit{def}}{=} \\
&\quad \mathit{case}\ (\boldsymbol{\lambda}_ : \mathit{Error}.\ \boldsymbol{\lambda}_ : \mathit{Error} + \mathit{Binding}.\ \mathit{in}\mathbf{1}\ \mathit{error}) \\
&\qquad (\boldsymbol{\lambda} b_1 : \mathit{Binding}.\ \mathit{case}\ (\boldsymbol{\lambda}_ : \mathit{Error}.\ \mathit{in}\mathbf{1}\ \mathit{error}) \\
&\qquad\qquad (\boldsymbol{\lambda} b_2 : \mathit{Binding}.\ \mathit{in}\mathbf{2}\ (b_1, b_2))) \\
&: \boldsymbol{\Pi}\, t \,|\, \mathit{Type}.\ (\mathit{Error} + \mathit{Binding}) \rightarrow (\mathit{Error} + \mathit{Binding}) \rightarrow (\mathit{Error} + \mathit{Binding})
\end{aligned}$$

Definition* 91 *CONS_BIN.

$$\begin{aligned}
&\mathit{CONS_BIN} \stackrel{\mathit{def}}{=} \\
&\quad \mathit{case}\ (\boldsymbol{\lambda}_ : \mathit{Error}.\ \boldsymbol{\lambda}_ : \mathit{Error} + \mathit{Binding}.\ \mathit{in}\mathbf{1}\ \mathit{error}) \\
&\qquad (\boldsymbol{\lambda} x : \mathit{bin_item}.\ \mathit{case}\ (\boldsymbol{\lambda}_ : \mathit{Error}.\ \mathit{in}\mathbf{1}\ \mathit{error}) \\
&\qquad\qquad (\boldsymbol{\lambda} b : \mathit{Binding}.\ \mathit{in}\mathbf{2}\ (\mathit{cons}\ x\ l))) \\
&: \boldsymbol{\Pi}\, t \,|\, \mathit{Type}.\ (\mathit{Error} + \mathit{bin_item}) \rightarrow (\mathit{Error} + \mathit{Binding}) \rightarrow (\mathit{Error} + \mathit{Binding})
\end{aligned}$$

Definition* 92 *IF.

$$\begin{aligned}
&\mathit{IF} \stackrel{\mathit{def}}{=} \boldsymbol{\lambda}\, t \,|\, \mathit{Type}. \\
&\quad \mathit{case}\ (\boldsymbol{\lambda}_ : \mathit{Error}.\ \boldsymbol{\lambda}_, _ : \mathit{Error} + t.\ \mathit{in}\mathbf{1}\ \mathit{error}) \\
&\qquad (\boldsymbol{\lambda} b : \mathit{bool}.\ \boldsymbol{\lambda} x, y : \mathit{Error} + t.\ \mathit{if}\ b\ x\ y) \\
&: \boldsymbol{\Pi}\, t \,|\, \mathit{Type}.\ (\mathit{Error} + \mathit{bool}) \rightarrow (\mathit{Error} + t) \rightarrow (\mathit{Error} + t) \rightarrow (\mathit{Error} + t)
\end{aligned}$$

Definition* 93 *APPLY.

$$\begin{aligned}
&\mathit{APPLY} \stackrel{\mathit{def}}{=} \\
&\quad \boldsymbol{\lambda}\, z, z' \,|\, \mathit{Ztype}.\ \boldsymbol{\lambda} eq : (\mathit{Typ}\ z) \rightarrow (\mathit{Typ}\ z) \rightarrow \mathit{bool}.\ \boldsymbol{\lambda} x, y : \mathit{Error} + \mathit{small_item}. \\
&\qquad \mathit{case}\ (\boldsymbol{\lambda}_ : \mathit{Error}.\ \mathit{in}\mathbf{1}\ \mathit{error}) \\
&\qquad\quad (\boldsymbol{\lambda} f : (\mathit{Typ}\ z) \rightarrow (\mathit{Error} + \mathit{small_item}).\ \mathit{wrap}_2\ z\ z'\ f\ y) \\
&\qquad\quad (\mathit{wrap}_1\ (\mathit{fun_ty}\ z\ z')\ (\mathit{Apply}\ |\ z\ |\ z'\ eq)\ x) \\
&: \boldsymbol{\Pi}\, z_1, z_2 \,|\, \mathit{Ztype}.\ \boldsymbol{\Pi}\, eq \,|\, (\mathit{Typ}\ z_1) \rightarrow (\mathit{Typ}\ z_1) \rightarrow \mathit{bool}. \\
&\quad (\mathit{Error} + \mathit{small_item}) \rightarrow (\mathit{Error} + \mathit{small_item}) \rightarrow (\mathit{Error} + \mathit{small_item})
\end{aligned}$$

Appendix D

LEGO library functions and lemmas

This appendix lists all the definitions and lemmas taken from the LEGO library which are mentioned in this thesis. The LEGO library was collated from the private libraries of the members of the LEGO club at the LFCS so it is impossible to pinpoint the origins of these definitions.

$$\mathit{trueProp} \stackrel{\mathit{def}}{=} \forall P\!:\!\mathit{Prop}.\, P \Rightarrow P$$
$$: \mathit{Prop}$$

$$\mathit{bool\_rec} \stackrel{\mathit{def}}{=} \text{[omitted]} : \mathbf{\Pi}\, T \,|\, \mathit{Type}.\, T \rightarrow T \rightarrow (\mathit{bool} \rightarrow T)$$

$$\mathit{is\_true} \stackrel{\mathit{def}}{=} \boldsymbol{\lambda} b\!:\!\mathit{bool}.\, b = \mathit{true}$$
$$: \mathit{bool} \rightarrow \mathit{Prop}$$

is_false is defined analogously to *is_true*.

$$\mathit{if} \stackrel{\mathit{def}}{=} \boldsymbol{\lambda}\, t \,|\, \mathit{Type}.\, \boldsymbol{\lambda} b\!:\!\mathit{bool}.\, \boldsymbol{\lambda} x, y\!:\!t.\, \mathit{bool\_rec}\; x\; y\; b$$
$$: \mathbf{\Pi}\, t \,|\, \mathit{Type}.\, \mathit{bool} \rightarrow t \rightarrow t \rightarrow t$$

$$\mathit{andalso} \stackrel{\mathit{def}}{=} \boldsymbol{\lambda} a, b\!:\!\mathit{bool}.\, \mathit{if}\;\; a\;\, b\;\, \mathit{false}$$
$$: \mathit{bool} \rightarrow \mathit{bool} \rightarrow \mathit{bool}$$

$$\mathit{orelse} \stackrel{\mathit{def}}{=} \boldsymbol{\lambda} a, b\!:\!\mathit{bool}.\, \mathit{if}\;\; a\;\, \mathit{true}\;\, b$$
$$: \mathit{bool} \rightarrow \mathit{bool} \rightarrow \mathit{bool}$$

$$inv \stackrel{def}{=} \lambda b{:}bool.\ if\ b\ false\ true$$
$$: bool \to bool$$

$$nat\_iter \stackrel{def}{=} \text{[omitted]} : \Pi\, t \,|\, Type.\ t \to (t \to t) \to (nat \to t)$$

$$nat\_rec \stackrel{def}{=} \text{[omitted]} : \Pi\, t \,|\, Type.\ t \to (nat \to t \to t) \to (nat \to t)$$

$$lt \stackrel{def}{=} nat\_iter\ (nat\_iter\ false\ (\lambda\_{:}bool.\ true))$$
$$(\lambda f{:}nat \to bool.\ nat\_rec\ false\ (\lambda x{:}nat.\ \lambda\_{:}bool. f\ x))$$
$$: nat \to nat \to bool$$

$$plus \stackrel{def}{=} \lambda m, b{:}nat.\ nat\_iter\ n\ suc\ m$$
$$: nat \to nat \to nat$$

$$list\_iter \stackrel{def}{=} \text{[omitted]} : \Pi\, s, t \,|\, Type.\ t \to (s \to t \to t) \to ((list\ s) \to t)$$

$$list\_rec \stackrel{def}{=} \text{[omitted]} : \Pi\, s, t \,|\, Type.\ t \to (s \to (list\ s) \to t \to t) \to ((list\ s) \to t)$$

$$member \stackrel{def}{=} \text{[omitted]} : \Pi\, t \,|\, Type.\ (t \to t \to bool) \to t \to (list\ t) \to bool$$

$$case \stackrel{def}{=} \text{[omitted]} : \Pi\, s, t, u \,|\, Type.\ (s \to u) \to (t \to u) \to ((s + t) \to u)$$

$$is\_in\mathbf{1} \stackrel{def}{=} \lambda\, s, t \,|\, Type.\ \lambda x{:}s + t.\ \exists y{:}z.\ in\mathbf{1}\ y = x$$
$$: \Pi\, s, t \,|\, Prop.\ (s + t) \to Prop$$

*is_in***2** is defined analogously to *is_in***1**.

$$in\mathbf{1}\_not\_in\mathbf{2} : \forall\, s, t{:}Type.\ \forall\, x{:}s.\ \neg(is\_in\mathbf{2}\ (in\mathbf{1}\ x))$$

$$sigma\_rec \stackrel{def}{=} \text{[omitted]}$$
$$: \Pi\, S \,|\, Type.\ \Pi\, F \,|\, S \to Type.\ \Pi\, T \,|\, Type.$$
$$(\Pi\, x{:}S.\ (F\ x) \to T) \to ((\Sigma\, y{:}A. F\ y) \to T)$$

D.1 The logic of LEGO

The LEGO library provides an encoding an intuitionistic, higher-order logic into the type theory UTT. For completeness, we show the definitions that make up this encoding.

$\forall$ is encoded as $\mathbf{\Pi}$.

$\Rightarrow$ is encoded as$\rightarrow$.

$$\exists \stackrel{def}{=} \boldsymbol{\lambda}\, T \,|\, \mathit{Type}.\, \mathbf{\Pi}\, P : T \rightarrow \mathit{Prop}.\ \forall\, Q : \mathit{Prop}.\ (\forall\, t : T.\ (P\ t) \Rightarrow Q) \Rightarrow Q$$
$$: \mathbf{\Pi}\, T \,|\, \mathit{Type}.\ (T \rightarrow \mathit{Prop}) \rightarrow \mathit{Prop}$$

$$\mathit{absurd} \stackrel{def}{=} \forall\, P : \mathit{Prop}.\, P$$
$$: \mathit{Prop}$$

$$\wedge \stackrel{def}{=} \boldsymbol{\lambda} P, Q : \mathit{Prop}.\ \forall\, R : \mathit{Prop}.\ (P \Rightarrow Q \Rightarrow R) \Rightarrow R$$
$$: \mathit{Prop} \rightarrow \mathit{Prop} \rightarrow \mathit{Prop}$$

$$\vee \stackrel{def}{=} \boldsymbol{\lambda} P, Q : \mathit{Prop}.\ \forall\, R : \mathit{Prop}.\ (P \Rightarrow R) \Rightarrow (Q \Rightarrow R) \Rightarrow R$$
$$: \mathit{Prop} \rightarrow \mathit{Prop} \rightarrow \mathit{Prop}$$

$$\neg \stackrel{def}{=} \boldsymbol{\lambda} P : \mathit{Prop}.\, P \Rightarrow \mathit{absurd}$$
$$: \mathit{Prop} \rightarrow \mathit{Prop}$$

Appendix E

Correctness of the Representation

In this appendix we discuss the relationship between our representation of Z and the semantics for Z proposed by Spivey [Spi88].

We have said that Z′ is a sublanguage of Z: we now make this statement precise by showing how to embed the syntax of Z′ into (the sublanguage of) Z that is treated in Spivey's semantics. It is necessary for us to extend Z with definitions of finite sets, natural numbers, and booleans. With the exception of the latter, these are all part of the library described in the ZRM.

We then state what it means for our semantics for Z′ to be *sound* with respect to Spivey's semantics for Z. We conjecture that this is indeed the case, and discuss what is involved in proving this result.

E.1 Translating Z′ to Z

The translation rules (figures E–1 and E–2) are straightforward. The Z′ phrase classes PRELUDE, MAINSPEC and SPEC are all translated to the Z phrase class SPEC. The Z′ phrase classes TYPE and TERM are both translated to the Z phrase class TERM. All the other terms in Z′ remain the same when translated, except for the `include` schema expression which must be treated specially.

Specification $[\,] : \mathrm{SPEC}_{Z'} \rightarrow \mathrm{SPEC}_Z$

[prelude `in` mainspec] = [prelude] `in` [mainspec]

Prelude $[\,] : \mathrm{prelude}_{Z'} \rightarrow \mathrm{SPEC}_Z$

[`given` $\mathrm{ident}_1, \ldots, \mathrm{ident}_n$] = `given` $\mathrm{ident}_1, \ldots, \mathrm{ident}_n$

Main specification $[\,] : \mathrm{mainspec}_{Z'} \rightarrow \mathrm{SPEC}_Z$

[`let` schema `end`] = `let` [schema] `end`

[`let` word = sexp] = `let` [word] = [sexp]

[mainspec `in` mainspec] = [mainspec] `in` [mainspec]

Schema $[\,] : \mathrm{SCHEMA}_{Z'} \rightarrow \mathrm{SCHEMA}_Z$

[decl | pred] = [decl] | [pred]

Declaration $[\,] : \mathrm{DECL}_{Z'} \rightarrow \mathrm{DECL}_Z$

[ident : type] = ident : [type]

[decl ; decl] = [decl] ; [decl]

Type $[\,] : \mathrm{TYPE}_{Z'} \rightarrow \mathrm{TERM}_Z$

$[\mathbb{N}] = \mathbb{N}$

$[\mathbb{B}] = \mathbb{B}$

[ident] = ident

$[\mathbb{F}\,\mathrm{type}] = \mathbb{F}\,[\mathrm{type}]$

$[\mathrm{type}_1 \times \mathrm{type}_2] = [\mathrm{type}_1] \times [\mathrm{type}_2]$

Term $[\,] : \mathrm{TERM}_{Z'} \rightarrow \mathrm{TERM}_Z$

[ident] = ident

[$\emptyset$ `[`type`]`] = $\emptyset$ `[`[type]`]`

$[\{\mathrm{term}_1, \ldots, \mathrm{term}_n\}] = \{[\mathrm{term}_1], \ldots, [\mathrm{term}_n]\}$

$[(\mathrm{term}_1, \mathrm{term}_2)] = ([\mathrm{term}_1], [\mathrm{term}_2])$

$[\mathrm{term}_1(\mathrm{term}_2)] = [\mathrm{term}_1]([\mathrm{term}_2])$

Figure E–1: Translating Z' to Z

Schema expression $[\,] : \mathrm{SEXP}_{Z'} \longrightarrow \mathrm{SEXP}_Z$

$[\texttt{schema}\ \mathrm{schema}\ \texttt{end}] = \texttt{schema}\ [\mathrm{schema}]\ \texttt{end}$

$[\mathrm{sdes}] = \mathrm{sdes}$

$[\neg\ \mathrm{sexp}] = \neg\ [\mathrm{sexp}]$

$[\mathrm{sexp}_1 \wedge \mathrm{sexp}_2] = [\mathrm{sexp}_1] \wedge [\mathrm{sexp}_2]$

$[\mathrm{sexp}_1 \vee \mathrm{sexp}_2] = [\mathrm{sexp}_1] \vee [\mathrm{sexp}_2]$

$[\mathrm{sexp}_1 \Rightarrow \mathrm{sexp}_2] = [\mathrm{sexp}_1] \Rightarrow [\mathrm{sexp}_2]$

$[\mathrm{sexp} \setminus (\mathrm{ident}_1, \ldots, \mathrm{ident}_n)] = [\mathrm{sexp}] \setminus (\mathrm{ident}_1, \ldots, \mathrm{ident}_n)$

$[\exists\ \mathrm{schema} \bullet \mathrm{sexp}] = \exists\ [\mathrm{schema}] \bullet [\mathrm{sexp}]$

$[\forall\ \mathrm{schema} \bullet \mathrm{sexp}] = \forall\ [\mathrm{schema}] \bullet [\mathrm{sexp}]$

$[\texttt{include}\ \mathrm{sdes}\ \mathrm{decl}\ \mathrm{pred}] = \texttt{schema}\ \mathrm{sdes} : [\mathrm{decl}] \mid [\mathrm{pred}]\ \texttt{end}$

Predicate $[\,] : \mathrm{PRED}_{Z'} \longrightarrow \mathrm{PRED}_Z$

$[\mathrm{term}_1 = \mathrm{term}_2] = [\mathrm{term}_1] = [\mathrm{term}_2]$

$[\mathrm{term}_1 \in \mathrm{term}_2] = [\mathrm{term}_1] \in [\mathrm{term}_2]$

$[\texttt{true}] = \texttt{true}$

$[\texttt{false}] = \texttt{false}$

$[\neg\ \mathrm{pred}] = \neg\ [\mathrm{pred}]$

$[\mathrm{pred}_1 \wedge \mathrm{pred}_2] = [\mathrm{pred}_1] \wedge [\mathrm{pred}_2]$

$[\mathrm{pred}_1 \vee \mathrm{pred}_2] = [\mathrm{pred}_1] \vee [\mathrm{pred}_2]$

$[\mathrm{pred}_1 \Rightarrow \mathrm{pred}_2] = [\mathrm{pred}_1] \Rightarrow [\mathrm{pred}_2]$

$[\exists\ \mathrm{ident} : \mathrm{type} \bullet \mathrm{pred}] = \exists\ \mathrm{ident} : [\mathrm{type}] \bullet [\mathrm{pred}]$

$[\forall\ \mathrm{ident} : \mathrm{type} \bullet \mathrm{pred}] = \forall\ \mathrm{ident} : [\mathrm{type}] \bullet [\mathrm{pred}]$

IDENT, WORD, DECOR and SDES are identical in Z and Z′ .

Figure E–2: Translating Z′ to Z

E.2 Soundness property

The following extract from [Spi88] explains how to give meaning to sequents of the form SEXP $\vdash$ PRED. (We have added the subscript "Spi" to Spivey's semantic brackets to distinguish them from our own.)

> Given an environment ρ obtained by evaluating a Z specification, we can say whether such a sequent is *valid* or *invalid*: the sequent
>
> $se \vdash p$
>
> is valid if and only if, in the environment ρ enriched with *se*, every model of the global variety satisfies p. Formally, if
>
> $$\rho_1 = enrich(\rho, sexp\ \rho\ \mathbf{1}\ [\![se]\!]_{\text{Spi}}),$$
>
> then the sequent is valid in ρ if and only if
>
> $$pred\ \rho_1\ \mathbf{1}\ [\![p]\!]_{\text{Spi}} = \rho_1.global.models.$$

The objects and operations of Spivey's semantics that are used here (varieties, environments, *enrich*, *sexp*, *pred*, *etc*) are described in Section 3.3 and defined fully in [Spi88].

In our semantics we also give a meaning to sequents. Given a UTT environment E, obtained by translating a Z specification, we say that a sequent

$se \vdash p$

is provable if and only if we can prove in UTT, using only the axioms of E, that all restricting models of the *Schema*, S, obtained by evaluating se in E satisfy the predicate obtained by evaluating p over the signature $S.\mathbf{1}$. Formally this is stated as follows:

$$Has\_prop\ [\![se, E]\!]\ [\![p, [\![se, E]\!].\mathbf{1}]\!]$$

Soundness statement We believe that our semantics for Z' is sound with respect to Spivey's semantics for Z, in the following sense:

> For any Z′ MAINSPEC mainspec, SEXP sexp, and PRED pred, if the sequent sexp ⊢ pred is provable in the UTT environment obtained by translating mainspec, then the sequent [sexp] ⊢ [pred] is valid in the environment obtained by evaluating [mainspec] according to Spivey's semantics.

We conjecture that the soundness statement above is true. We do not attempt to give a complete formal proof of this because of the magnitude of the task; this is an unfortunate consequence of our choice of a relatively shallow embedding for Z. Instead, we shall sketch how this result might be proved, stating what lemmas are required, proving some of these and giving intuitive arguments why the others should be true.

E.2.1 A restricted class of semantic objects

We assume that we have a fixed set of given types.

We begin by defining restricted portions of the semantic objects Global, *Predicate*, and *Schema*, which we shall call GoodGlobal, GoodPred, GoodPred-Body, GoodSchema, *etc*. The definitions are shown in figure E–3.

These restricted semantic objects have the property that they can be mapped back to phrase classes in Z′ . We shall define two such mappings, which we shall name, simply, f and g.

For those elements of GoodPredBody which contain no *lookup*s we can define an inverse to the semantic operation $[\![\ ]\!]_{\text{top}}$ on PRED. Similarly, the inverse of $[\![\ ]\!]_{\text{top}}$ on DECL gives us a function from Vars to DECL. Hence, given any GoodGlobal G we can produce a SCHEMA, s, consisting of the DECL derived from the Vars component of G, and the PRED obtained as described above from the conjunction of all the GoodPreds in the image of the GoodAxioms component of G (provided that none of these contain any *lookup*s). We then use Spivey's semantic operations to obtain from this SCHEMA the following variety:

$(\mathit{spec}\ \boldsymbol{\rho}\ [\![\texttt{let [s] end}]\!]_{\text{Spi}}).\mathit{global}$

GoodGlobal = Vars $\times$ GoodAxioms

GoodAxioms = UTTIdent ⤀ GoodPredBody

GoodSchemaDict = UTTIdent ⤀ GoodSchema

GoodSchema ::= "(" *Signature* "," GoodPred ")"

GoodPred ::= "$\boldsymbol{\lambda} b : Binding$**.** " GoodPredBody

GoodPredBody	::=	"*IS_TRUE* (*EQUAL*" *Ztype* GoodTerm GoodTerm ")"
	|	"*IN* (*EQUAL*" *Ztype* ")" GoodTerm GoodTerm
	|	"*trueProp*"
	|	"*absurd*"
	|	"~" GoodPredBody
	|	GoodPredBody "$\wedge$" GoodPredBody
	|	GoodPredBody "$\vee$" GoodPredBody
	|	GoodPredBody "$\Rightarrow$" GoodPredBody
	|	"$\exists$ " IDENT " : *Typ*" *Ztype* "**.** " GoodPredBody
	|	"$\forall$ " IDENT " : *Typ*" *Ztype* "**.** " GoodPredBody
GoodTerm	::=	"*in***2** (" *Ztype* ", " IDENT ")"
	|	"(*lookup* (" IDENT ", " *Ztype* ") *b*)"
	|	"*NULL* " *Ztype*
	|	"*ADD_ONE* (*Equal* " *Ztype* ")" GoodTerm GoodTerm
	|	"*PAIR*" GoodTerm GoodTerm
	|	"*APPLY* (*Equal*" *Ztype* ")" GoodTerm GoodTerm

Figure E–3: Definitions of "Good" semantic objects

Here $\boldsymbol{\rho}$ is the environment produced by enriching the empty environment with the assumed given types. The above procedure gives us a partial function f from GoodGlobals to varieties.

Given any GoodPred P, we can form a *Signature*, *sig* by making a list of all the *sig_items* to which *lookup* is applied in P. We can then compute a PRED, pred, such that $[\![\text{pred}, \mathit{sig}]\!]$ is identical to P. Hence, we have a mapping from GoodPred to PRED. We can also define an inverse to the semantic function $[\![\ ]\!]$ on DECL. We use these two mappings to define a function g from GoodSchema to SCHEMA.

E.2.2 Proof Strategy

Assume we have a set of given types, a MAINSPEC, mainspec, a SEXP, sexp and a PRED, pred. Assume that in the Global E.$\mathbf{1}$, where

$$E = [\![\text{mainspec}, \emptyset]\!]$$

we can prove

$$\mathit{Has\_prop}\ [\![\text{sexp}, E]\!]\ [\![\text{pred}, [\![\text{sexp}, E]\!].\mathbf{1}]\!]$$

We claim (Statement 1) that there is a GoodSchema S which can be shown to be logically equivalent to $[\![\text{sexp}, E]\!]$, and which has the same signature. We can therefore prove

$$\mathit{Has\_prop}\ S\ [\![\text{pred}, S.\mathbf{1}]\!]$$

Now we consider the specification obtained by extending mainspec with the schema g(S), treated as an axiomatic description. Consider the Global, G, obtained by evaluating this specification:

$$\text{G} = ([\![\text{mainspec}\ \texttt{in let}\ \text{H}\texttt{:}\ \text{g(S)}\ \texttt{end}, \emptyset]\!]).\mathbf{1}$$

(Here, H is a label chosen as described in the discussion of Statement 2.) Facts 3 and 4 tell us that G is a GoodGlobal.

We claim (Statement 2) that we can use the axioms of G to prove $[\![\text{pred}]\!]_{\text{top}}$. We then claim (Statement 3) that this implies [pred] is valid in the variety f (G). Finally, we claim that (Statement 4) this variety is the same as that derived by using Spivey's semantics to evaluate [sexp] in the environment obtained by evaluating [mainspec].

Hence, Spivey's semantics validates the sequent

$$[\text{sexp}] \vdash [\text{pred}]$$

E.2.3 Lemmas needed to prove soundness

Fact 1 For all TERMs term, $[\![\text{term}]\!]_{\text{top}}$ is a GoodTerm.

Proof. By induction on the structure of TERM.

Fact 2 For all PREDs pred, $[\![\text{pred}]\!]_{\text{top}}$ is a GoodPredBody.

Proof. By induction on the structure of PRED, using Fact 1 as needed.

Fact 3 For all MAINSPECs mainspec, the Global $[\![\mathit{mainspec}, \emptyset]\!]$ is always a Good-Global.

Proof. By induction on the structure of MAINSPEC, using Facts 1 and 2 as needed.

Fact 4 For all GoodGlobals, g, and all GoodSchemas, S, and all labels, H the Global

$$G \bigcup [\![\mathit{let}\ \ H\!:\!(g\ S)\ \ \mathit{end}]\!]$$

is always a GoodGlobal.

Proof. By induction on the structure of GoodSchema.

Fact 5 For all TERMs term and all *Signature*s *sig*, $[\![\text{term}, \mathit{sig}]\!]$ is a GoodTerm.

Proof. By induction on the structure of TERM.

Fact 6 For all PREDs pred and all *Signatures sig* $[\![\text{pred}, sig]\!]_{\text{body}}$ is a GoodPred-Body.

Proof. By induction on the structure of PRED, using Fact 5 as needed.

Fact 7 For all PREDs pred and all *Signatures sig* $[\![\text{pred}, sig]\!]$ is a GoodPred.

Proof. This follows directly from Fact 6.

Fact 8 For all SCHEMAs schema, $[\![\text{schema}]\!]$ is a GoodSchema.

Proof. This follows directly from Fact 7.

Statement 1

Translating a SEXP gives a *Schema* which is not necessarily a GoodSchema. However, from this *Schema*, we can systematically derive a GoodSchema which has the same signature and is logically equivalent.

Why should this be true?

The translation rules for SEXP always produce GoodSchemas, except for the following cases: hiding, existential and universal quantification, and primed schemas. In all of these cases it is possible to give rules for systematically constructing a GoodSchema and to prove in UTT that that GoodSchema is logically equivalent to the one obtained by our translation. We shall explain how this is done for the case of hiding by giving an example and informally describing the general transformation.

Consider the following annotated schema expression:

$\texttt{schema x:T; y:T | x} =_{\mathrm{T}} \texttt{y end} \setminus \texttt{(x:T)}$

By our translation rules, this will be represented by the *Schema*

$$
\begin{array}{l}
\mathit{Hide}\ [(\llbracket x \rrbracket, \mathit{given\_ty}\ T)] \\
\quad ([(\llbracket x \rrbracket, \mathit{given_ty}\ T), (\llbracket y \rrbracket, \mathit{given\_ty}\ T)], \\
\quad \boldsymbol{\lambda} b : \mathit{Binding}.\ \mathit{IS\_TRUE}\ (\mathit{EQUAL}\ (\mathit{given\_ty}\ T) \\
\qquad\qquad\qquad (\mathit{lookup}\ (\llbracket x \rrbracket, \mathit{given\_ty}\ T)\ b) \\
\qquad\qquad\qquad (\mathit{lookup}\ (\llbracket y \rrbracket, \mathit{given\_ty}\ T)\ b)))
\end{array}
$$

Expanding the definition of *Hide*, we see that this is the same as:

$$
\begin{array}{l}
([(\llbracket y \rrbracket, \mathit{given\_ty}\ T)], \\
\boldsymbol{\lambda} b : \mathit{Binding}.\ \exists\, b' : \mathit{Binding}. \\
\quad (\mathit{is\_true}\ (\mathit{matches}\ [(\llbracket x \rrbracket, \mathit{given\_ty}\ T)]\ b'))\ \wedge \\
\quad (\mathit{IS\_TRUE}\ (\mathit{EQUAL}\ (\mathit{given\_ty}\ T) \\
\qquad\qquad\qquad (\mathit{lookup}\ (\llbracket x \rrbracket, \mathit{given\_ty}\ T)\ (\mathit{join\_bin}\ b\ b')) \\
\qquad\qquad\qquad (\mathit{lookup}\ (\llbracket y \rrbracket, \mathit{given\_ty}\ T)\ (\mathit{join\_bin}\ b\ b')))))
\end{array}
$$

We obtain a GoodSchema by doing the following: for each *sig_item* $(\llbracket i \rrbracket, z)$ in the hidden signature, we insert an existential quantifier of the form $\exists\, i : \mathit{Typ}\ z.$ into the predicate. The sequence of existential quantifiers so obtained replaces the single quantifier $\exists\, b' : \mathit{Binding}.$ Every occurrence of $(\mathit{lookup}\ (\llbracket i \rrbracket, z)\ (\mathit{join\_bin}\ b\ b'))$ in the body of the predicate is replaced by the wrapped form of the variable bound in the existential quantifier, i.e., $(\mathit{in}\mathbf{2}\ (\mathit{given\_ty}\ z, i))$. All other *lookup*s of a *sig_item* in $(\mathit{join\_bin}\ b\ b')$ are replaced by *lookup*s of the same *sig_item* in b. For our example, this transformation has the following result:

$$
\begin{array}{l}
([(\llbracket y \rrbracket, \mathit{given\_ty}\ T)], \\
\boldsymbol{\lambda} b : \mathit{Binding}.\ \exists\, x : \mathit{Typ}\ (\mathit{given\_ty}\ T). \\
\quad \mathit{IS\_TRUE}\ (\mathit{EQUAL}\ (\mathit{given\_ty}\ T) \\
\qquad\qquad\qquad (\mathit{in}\mathbf{2}\ (\mathit{given\_ty}\ T, x)) \\
\qquad\qquad\qquad (\mathit{lookup}\ (\llbracket y \rrbracket, \mathit{given\_ty}\ T)\ b))))
\end{array}
$$

In general, the transformation described above will always produce a GoodSchema which can be shown in UTT to be logically equivalent to the *Schema* produced by *Hide*, (provided that *Hide* is itself being applied to a GoodSchema) This can be proved by induction on the list of *sig_item*s to be hidden. The proof relies on the restricted structure of GoodPred — the only references to the *Binding* argument b in a GoodPred are via *lookup*.

Similar transformations can be defined for the operations *Exists* and *All*. For the *Prime* operation, the transformation is as follows: Suppose we have a a GoodSchema S. The *Prime* operation gives us the following:

$$(\mathit{prime\_sig}\ S.\mathbf{1}, \lambda b{:}\mathit{Binding}.\ S.\mathbf{2}\ (\mathit{post\_bin}\ b))$$

For every *sig_item* s in $S.\mathbf{1}$, we replace all occurrences of $(\mathit{lookup}\ s\ (\mathit{post\_bin}\ b))$ in the above by $(\mathit{lookup}\ (\mathit{prime\_item}\ s)\ b)$. We can prove that this gives us a logically equivalent GoodSchema.

To prove Statement 1 formally we need to prove a more general statement in order to deal with the case of schema designators, which are evaluated by looking them up in the ambient UTTenv.

We first generalise the definition of "good" semantic objects so that it encompasses the whole of a UTTenv: a GoodUTTenv is one composed of a GoodGlobal together with a SchemaDict whose image consists only of GoodSchemas.

Next, we generalise the definition of "logical equivalence" so that it applies to UTTenvs: two UTTenvs E and E′ are logically equivalent if their Globals are identical, their SchemaDicts have the same domain, and, given any UTTident, i within this domain, the two *Schema*s E.SchemaDict(i) and E′.SchemaDict(i) have the same signature and are logically equivalent.

We must then prove the following more general statement.

> Translating a MAINSPEC gives a UTTenv, E, which is not necessarily a GoodUTTenv. However, given this UTTenv, we can systematically derive a GoodUTTenv, E′ which is logically equivalent to E.

This proof would be done by induction on the structure of MAINSPEC.

Next we will have to show, by induction on the structure of SEXP, that Statement 1 is true, provided that the evaluation of the SEXP takes place within a GoodUTTenv.

To complete the proof of statement 1, we will need to show that if a SEXP, sexp, is translated within two logically equivalent UTTenvs, the resulting *Schema*s are logically equivalent.

Statement 2

This states that given a GoodGlobal G, a GoodSchema S, and a PRED pred, if we can show, using only the axioms of G and the rules of higher-order intuitionistic logic, that

$$\mathit{Has\_prop}\ S\ [\![\text{pred}, S.1]\!]$$

then there is a label H such that, in the GoodGlobal G $\bigcup$ $[\![$`let` H : $(g\ S)$ `end`$]\!]$ we can prove $[\![\text{pred}]\!]_{\text{top}}$.

Why should this be true? Examine what happens when we prove

$$\mathit{Has\_prop}\ S\ [\![\text{pred}, S.1]\!]$$

Let us suppose that the signature of S is $[([\![x_1]\!], t_1), \ldots, ([\![x_n]\!], t_n)]$. After expanding the definition of *Has_prop* and and then using the `Intros` tactic, we obtain a proof context of the following form. (The names for the introduced *small_item*s must be chosen carefully in order to be syntactically identical to the

*Ident*s in the signature of S.

$$
\begin{array}{l}
[b : Binding] \\
[H : restricts\_to\_model\ S\ b\ :] \\
[x_1, \ldots, x_n : small\_item] \\
[H_1 : Eq\ x_1\ (lookup\ (\llbracket x_1 \rrbracket, t_1)\ b)] \\
\vdots \\
[H_n : Eq\ x_n\ (lookup\ (\llbracket x_n \rrbracket, t_n)\ b)] \\
[HH : S.\mathbf{2}\ b]
\end{array}
$$

$$? : \llbracket \text{pred}, S.\mathbf{1} \rrbracket_{\text{body}}$$

We then rewrite the goal and the hypothesis *HH* with the equalities $H_1 \ldots H_n$. The resulting hypothesis and goal will contain no *lookup*s. The salient part of the proof context for the rest of the proof consists only of the declarations of $x_1 \ldots x_n$ and the rewritten version of the *HH*. This part of the proof context extends the original GoodGlobal, G, giving a new GoodGlobal, which is the same as $G \bigcup \llbracket \texttt{let}\ HH\ :\ (g\ S)\ \texttt{end} \rrbracket$. (The label is chosen to be the same as the name of the hypothesis HH.) The goal is the same as $\llbracket \text{pred} \rrbracket_{\text{top}}$. We must then show that the original goal can be proved if and only if the current goal can be proved in the current context – this is true because of the particular UTT rules that we have used to manipulate the proof up to this point.

The paragraph above conceals several non-trivial proof obligations. For example, showing that the proof can always be completed using only a reduced portion of the proof context would require complicated reasoning about the rules of UTT and the behaviour of the LEGO system.

Statement 3

Given a GoodGlobal G and a Z′ PRED pred, if we can prove the statement $\llbracket \text{pred} \rrbracket_{\text{top}}$ using only the axioms in G and the rules of intuitionistic higher-order logic, then [pred] is valid in the variety f (G).

Why should this be true? Intuitively, this is true because the proof rules for intuitionistic logic are sound with respect to the classical model theory used in Spivey's semantics. A complete proof might be done by induction on the structure of the proofs of $[\![\text{pred}]\!]_{\text{top}}$.

Statement 4

Given a MAINSPEC, mainspec, a SEXP, sexp, and a label, H, let S be a GoodSchema, obtained by the methods described in the proof of Statement 1, that is logically equivalent to $[\![sexp, [\![\text{mainspec}, \emptyset]\!]]\!]$. Then the variety

$$V = \mathrm{f}\ (([\![\text{mainspec}\ \texttt{in let}\ \mathrm{H} :\ (g\ S)\ \texttt{end}, \emptyset]\!]).\mathbf{2})$$

is the same as that derived by using Spivey's semantics to evaluate [sexp] in the environment obtained by evaluating [mainspec]. In the language of Spivey's semantics, if $\boldsymbol{\rho}$ is the environment obtained by evaluating [mainspec] then we have that

$$V = (enrich(\boldsymbol{\rho}, sexp\ \boldsymbol{\rho}\ \mathbf{1}\ [\![[\text{sexp}]]\!]_{\text{Spi}})).global$$

Why should this be true? We believe that the proof of this statement should be routine, since the major differences between Spivey's semantics and our own have been ironed out in Statement 1.

To prove this statement formally, we would first have to prove a more general statement (as was necessary for Statement 1) relates environments and GoodUTTenvs. First we must define a mapping h from GoodUTTenvs to environments. The general statement is then the following:

> Given a main specification MAINSPEC let E be a GoodUTTenv, obtained by the methods described in the proof of Statement 1, that is logically equivalent to $[\![\text{MAINSPEC}, \emptyset]\!]$. Then the environment h(E) is the same as that obtained by evaluating MAINSPEC using Spivey's semantics.

The proof of this would be done by induction on the structure of MAINSPEC.

Bibliography

[1] [Acz82] Peter Aczel. The type theoretic interpretation of constructive set theory: Choice Principles. In *The LEJ Brouwer Centenary Symposium*, A.S. Troelstra and D. van Dalen, editors. North Holland, 1982.

[Abr84a] J-R. Abrial. Programming as a Mathematical Exercise. In *Mathematical Logic and Programming Languages*, C.A.R. Hoare and J.C. Shepherdson, editors. Prentice-Hall International, 1984.

[Abr84b] J-R. Abrial. The Mathematical Construction of a Program and its Application to the Construction of Mathematics. In *Science of Computer Programming*, 4, 1984.

[AHM87] Arnon Avron, Furio Honsell and Ian A. Mason. Using Typed Lambda Calculus to Implement Formal Systems on a Machine. Technical Report ECS-LFCS-87-31, LFCS, The University of Edinburgh, 1987.

[Alt93] Thorsten Altenkirch. *Construction, Inductive Types and Strong Normalization*. PhD thesis, The University of Edinburgh, 1993.

[ASM79] J-R. Abrial, S.A. Schuman and B. Meyer. Specification Language Z. In *On the Construction of Programs*, R.M. McKeag and A.M. Macnaghten, editors. Cambridge University Press, 1979.

[BCJ84] H. Barringer and J.H. Cheng and C.B. Jones. A Logic Covering Undefinedness in Program Proofs. In *Acta Informatica*, 21, 251-69.

[BG81] R.M. Burstall and J.A. Goguen. An informal introduction to specification using CLEAR. In *The Correctness Problem in Computer Science*, R.S. Boyer and J.S. Moore, editors, Academic Press, 1981.

[BG+92] Richard Boulton and Andrew Gordon *et al*. Experience with embedding hardware description languages in HOL. In V. Stavridou, T.F. Melham, and R.T. Boute, editors, *Theorem Provers in Circuit Design: Theory, Practice and Experience*, pages 129-156. IFIP TC10/WG 10.2, North Holland, June 1992.

[BG94] Jonathan Bowen and Mike Gordon. Z and HOL, 1994. submitted to the '94 Z User Meeting.

[BM91] Rod Burstall and James McKinna. Deliverables: an approach to program development in the Calculus of Constructions. . Technical Report ECS-LFCS-91-133, LFCS, The University of Edinburgh, 1991.

[CH88] T. Coquand and G. Huet. The calculus of constructions. In *Information and Computation, vol 76*, 1988.

[CO+95] Judy Crow, Sam Owre *et al* A tutorial introduction to PVS. *Workshop on Industrial-Strength Formal Specification Techniques*, Florida, 1995.

[Con86] Robert L. Constable, *et al*. *Implementing Mathematics with the NuPrl Proof Development System*. Prentice-Hall, 1986.

[CH89] Robert L. Constable, *et al*. *NuPrl as a General Logic* Technical Report TR89-1021, Computer Science, Cornell University.

[Coq95] Coq: Formal Specifications and Program Validation. `http://zenon.inria.fr:8003/Equipes/COQ-eng.html`

[DFH+93] Dowek, Felty, *et al*. The Coq proof assistant user's guide, version 5.8. Technical report, INRIA-Rocquencourt, 1993.

[Geo91] C. George. The RAISE Specification Language: A Tutorial. In *Proceedings of VDM '91*, Springer-Verlag LNCS 551, 1991.

[GGH93] Stephen J. Garland, John V. Guttag, and James J. Horning. An Overview of Larch. In *Functional Programming, Concurrency, Simulation and Automated Reasoning*, Peter E. Lauer, editor. Springer-Verlag LNCS 693, 1993.

[Gog94] Healfdene Goguen. The Metatheory of UTT. In, *Proceedings of the BRA workshop on Types and Proofs*, 1994.

[Gog95] Healfdene Goguen. *A Typed Operational Semantics for Type Theory*. PhD thesis, The University of Edinburgh, 1995.

[Gor88] M.J.C. Gordon HOL: A Proof Generating System for Higher-Order Logic. In *VLSI Specification, Verification and Synthesis*, edited by G. Birtwhistle and P.A. Subrahmanyam, Kluwer, 1988.

[Har91] W. Harwood. Proof rules for Balzac. Technical Report WTH/P7/001, Imperial Software Technology, Cambridge, 1991.

[Hay93] Ian Hayes, editor. *Specification Case Studies*. Prentice Hall, second edition, 1993.

[HHP87] Robert Harper, Furio Honsell and Gordon Plotkin. A Framework for Defining Logics. Technical Report *ECS-LFCS-87-23*, LFCS, The University of Edinburgh, 1987.

[HJ88] C.A.R. Hoare and He, Jifeng. Data Refinement in a Categorical Setting. Technical Report, Programming Research Group, Oxford University Computing Laboratory, 1988.

[HOL95] The HOL Theorem Prover. `http://www.comlab.ox.ac.uk/archive/formal-methods/hol.html`

[HST89] Robert Harper, Donald Sannella and Andrzej Tarlecki Structure and Representation in LF. Technical Report ECS-LFCS-89-75, LFCS, The University of Edinburgh, 1989.

[JHS86] He Jifeng, C.A.R. Hoare, and J.W. Sanders. Data Refinement Refined. Lecture Notes in Computer Science 213, 1986.

[HJN93] I.J. Hayes, C.B. Jones and J.E. Nicholls. Understanding the differences between VDM and Z. Technical Report UMCS-93-8-1, University of Manchester, 1993.

[Hof92] Martin Hofmann. Formal Development of Functional Programs in Type Theory — A Case Study. Technical Report ECS-LFCS-92-228, LFCS, The University of Edinburgh, 199.

[How80] W.A. Howard. The formulæ-as-types notion of construction. In J.Hindley and J.Seldin, editors, *To H.B. Curry: Essays on Combinatory Logic*. Academic Press, 1980.

[JJ+91] C.B. Jones, K.D. Jones, *et al*. *"Mural": A Formal Development Support System* Springer-Verlag, 1991.

[JM94] Claire Jones and Savi Maharaj. The LEGO library. Available electronically at `http://www.dcs.ed.ac.uk/packages/lego/`

[Jon86] Cliff B. Jones. *Systematic Software Development Using VDM*. Prentice-Hall International, 1986.

[Jon92] R.B. Jones. Methods and Tools for the Verification of Critical Properties. In R. Shaw, editor, *Proceedings of the 5th BCS-FACS workshop on refinement*. Springer-Verlag, 1992.

[KST94] Stefan Kahrs, Donald Sannella and Andrzej Tarlecki. The Definition of Extended ML. Technical Report ECS-LFCS-94-300, LFCS, The University of Edinburgh, 1994.

[Lar94] Larch Home Page `http://larch-www.lcs.mit.edu:8001/larch/`

[LEG95] LEGO Home Page `http://www.dcs.ed.ac.uk/packages/lego/`

[LP92] Zhaohui Luo and Robert Pollack. LEGO Proof Development System: user's manual, Technical report ECS-LFCS-92-211, LFCS, The University of Edinburgh, 1992.

[Luo90] Zhaohui Luo. *An Extended Calculus of Constructions*. PhD thesis, The University of Edinburgh, 1990.

[Luo91] Zhaohui Luo. Program Specification and Data Refinement in Type Theory. Proc. of the Fourth Inter. Joint Conf. on the Theory and Practice of Software Development (TAPSOFT), 1991.

[Luo92] Zhaohui Luo. A Unifying Theory of Dependent Types: the Schematic Approach. Technical Report ECS-LFCS-92-202, LFCS, The University of Edinburgh, 1992.

[Luo94] Zhaohui Luo. *Computation and Reasoning: A Type Theory for Computer Science*. Oxford University Press, 1994.

[Mah90] Savitri Maharaj. Implementing Z in LEGO. Master's thesis, University of Edinburgh, 1990.

[Mah94] Savi Maharaj. Encoding Z-style schemas in UTT. In *Types for Proofs and Programs*, Lecture Notes in Computer Science, 806. Springer-Verlag, 1994.

[MG94] Savi Maharaj and Elsa Gunter. Studying the ML module system in HOL. In *Higher Order Logic Theorem Proving and Its Applications*, Lecture Notes in Computer Science, 859. Springer-Verlag, 1994.

[Mar93] Andrew Martin. Encoding W: A logic for Z in 2OBJ. In *FME '93: Industrial-Strength Formal Methods*, Lecture Notes in Computer Science. Springer-Verlag, 1993.

[McK92] James Hugh McKinna. *Deliverables: A Categorical Approach to Program Development in Type Theory*. PhD thesis, The University of Edinburgh, 1992.

[Mil89] Robin Milner. *Communication and Concurrency*, Prentice-Hall International, 1989.

[Mor90] Carroll Morgan. *Programming from specifications*, Prentice-Hall International, 1990.

[MT94] D.B. MacQueen and M. Tofte. A Semantics for Higher-Order Functors. In *European Symposium on Programming* Springer-Verlag, 1994.

[Nes94] Monica Nesi. Value-Passing CCS in HOL. In Proceedings of the 6*th International Workshop on Higher Order Logic Theorem Proving and its Applications*, Springer-Verlag LNCS 780, 1994.

[Nup95] NuPrl 4 Automated Reasoning System Browser. `http://www.cs.cornell.edu/Info/Projects/NuPrl/nuprl.html`

[Pol95] Robert Pollack *The Theory of LEGO*. PhD thesis, The University of Edinburgh, 1995.

[Pro2] ProofPower server. Send email to ProofPower-server@win.icl.co.uk.

[Rai95] RAISE—Rigorous Approach to Industrial Software Engineering. `http://dream.dai.ed.ac.uk/raise/`

[Saa91] M. Saaltink. Z and EVES. Technical Report TR-91-5449-02, Odyssey Research Associates, 265 Carling Avenue, Suite 506, Ottawa, Ontario E1S 2E1, Canada, 1991.

[San89] Donald Sannella. Formal program development in Extended ML for the working programmer. In *Proceedings of the 3rd BCS/FACS Workshop on Refinement*, Hursley Park, 1990.

[ST89] Donald Sannella and Andrzej Tarlecki. Toward formal development of ML programs: foundations and methodology. In *Proc. Colloq. on Current Issues in Programming Languages*, Joint Conference on Theory and Practice of Software Development (TAPSOFT), Barcelona, Springer LNCS, 1989.

[Sch95] Thomas Schreiber. An axiomatic approach to imperative programs in type theory. Technical Report (forthcoming), LFCS, The University of Edinburgh, 1995.

[Spi88] J.M. Spivey. *Understanding Z: A Specification Language and its formal semantics*. Cambridge Tracts in Theoretical Computer Science 3, 1988.

[Spi92] J.M. Spivey. *The Z Notation: A Reference Manual*, second edition. Prentice-Hall International, 1992.

[Sym93] Syme, D. (1994) Reasoning with the Formal Definition of Standard ML in HOL. In: *Higher Order Logic Theorem Proving and its Applications, 1993*. Lecture Notes in Computer Science 780, Springer-Verlag.

[VG94] VanInwegen, M. and Gunter, E. (1994) HOL-ML. In: *Higher Order Logic Theorem Proving and its Applications, 1993*. Lecture Notes in Computer Science 780, Springer-Verlag.

Index