Basic Research in Computer Science

BRICS RS-95-37 **Agerholm & Gordon: Experiments with ZF Set Theory in HOL and Isabelle**

Experiments with ZF Set Theory in HOL and Isabelle

Sten Agerholm
Mike Gordon

BRICS Report Series **RS-95-37**

ISSN 0909-0878 **July 1995**

See back inner page for a list of recent publications in the BRICS Report Series. Copies may be obtained by contacting:

**BRICS
Department of Computer Science
University of Aarhus
Ny Munkegade, building 540
DK - 8000 Aarhus C
Denmark**

**Telephone: +45 8942 3360
Telefax: +45 8942 3255
Internet: BRICS@brics.dk**

BRICS publications are in general accessible through WWW and anonymous FTP:

`http://www.brics.dk/`
`ftp ftp.brics.dk (cd pub/BRICS)`

Experiments with ZF Set Theory in HOL and Isabelle

Sten Agerholm and Mike Gordon

University of Cambridge Computer Laboratory
New Museums Site, Pembroke Street
Cambridge CB2 3QG, UK

Abstract. Most general purpose proof assistants support versions of typed higher order logic. Experience has shown that these logics are capable of representing most of the mathematical models needed in Computer Science. However, perhaps there exist applications where ZF-style set theory is more natural, or even necessary. Examples may include Scott's classical inverse-limit construction of a model of the untyped λ-calculus (D_∞) and the semantics of parts of the Z specification notation. This paper compares the representation and use of ZF set theory within both HOL and Isabelle. The main case study is the construction of D_∞. The advantages and disadvantages of higher-order set theory versus first-order set theory are explored experimentally. This study also provides a comparison of the proof infrastructure of HOL and Isabelle.

1 Introduction

Set theory is the standard foundation for mathematics and for formal notations like Z [30], VDM [14] and TLA+ [15]. However, most general purpose mechanised proof assistants support typed higher order logics (type theories). Examples include Alf [17], Coq [7], EHDM [19], HOL [12], IMPS [9], LAMBDA [10], LEGO [16], Nuprl [5], PVS [26] and Veritas [13]. For many applications type theory works well, but there are certain classical constructions, like the definition of the natural numbers as the set $\{\emptyset, \{\emptyset\}, \{\emptyset,\{\emptyset\}\}, \{\emptyset,\{\emptyset\},\{\emptyset,\{\emptyset\}\}\}, \cdots\}$, that are essentially untyped. Furthermore, the development of some branches of mathematics, e.g. abstract algebra, are problematical in type theory. In the long term it might turn out that such areas can be satisfactorily developed in a type-theoretic setting, but research is needed to establish this. For immediate practical applications it seems hard to justify not just taking ordinary (i.e. set-theoretic) mathematics 'off the shelf'.

Several proof assistants for set theory are available. In the formal methods community[1] the best known are probably Isabelle [23], a generic system that supports various kinds of type theory as well as ZF set theory, and EVES [28]. Both of these have considerable automation and are capable of proving difficult theorems. The Mizar system from Poland [27] is a low level proof checker based

[1] In other communities (e.g. artificial intelligence) there is related work on theorem proving for set theory (e.g. Ontic [18]).

on set theory that has been used to check enormous amounts of mathematics – there is even a journal devoted to it [8]. Mizar has only recently become well-known in the theorem proving community. Another active group focusses on metatheory and decision procedures for fragments of set theory [4]. Like Mizar, this work is not well known in the applied verification community.

Work with Isabelle provides particular insight into type theory versus set theory because it supports both. Users of Isabelle can choose between either ZF or various styles of type theory (including HOL-like higher order logic and Martin Löf type theory), depending on which seems most appropriate for the application in hand. However, the theories do not interface to each other, so one cannot move theorems back and forth between them (except by manually porting proofs). Anecdotal evidence from Isabelle users suggests that, for equivalent kinds of theorems, proof in higher order logic is usually easier and shorter than in set theory, but that certain general constructions (e.g. defining models of recursively defined datatypes) are easier to do in set theory. One reason why set-theoretic proofs can be more tedious is because they may involve the verification of set membership conditions; the corresponding conditions in higher order logic being handled automatically by type checking.[2]

It is hoped that the experiments reported here will clarify the strengths and weaknesses of type theory versus set theory as frameworks for mechanising the mathematics needed to support computer system verification. A long term goal is to develop a way of getting the best of both worlds in a single framework.

The rest of this paper is organised as follows: Section 2 presents a version of ZF set theory axiomatised in the HOL logic (this will be referred to as HOL-ST). This is then explored via a case study: the construction of D_∞, Scott's model of the λ-calculus (Section 3). In Section 4, Isabelle's set theory and proof infrastructure is discussed in relation to HOL-ST. The final section (Section 5) summarises our conclusions.

2 A ZF-like Set Theory in HOL

This section is condensed from a more leisurely and detailed exposition presented elsewhere [11].

2.1 Standard Axioms of Set Theory

HOL-ST is an axiomatic theory based on ZF, but formulated in higher order logic. The theory, called ST, introduces a new type V and a new constant $\in : V \times V \to \mathit{bool}$, together with higher-order versions of the ZF axioms.

Because they are formulated in higher order logic, the axioms described below are strictly stronger than ZF. Furthermore, although the axiom of choice is not

[2] Note, however, that explicit proof of membership conditions in set theory results in more being formally proved than with type checking, which is typically done by programs outside the logic; i.e. what is formally proved in the former case is just calculated in the latter.

stated explicitly, it is easily proved in HOL using the choice function (ε-operator). An axiom asserting the existence of unordered pairs is also not needed as this follows from Replacement (pairs are images of $\mathbb{P}\,\emptyset$). The axioms of ST in the HOL logic are as follows.

Extensionality : $\forall s\ t.\ (s = t) = (\forall x.\ x \in s = x \in t)$

Empty set : $\exists s.\ \forall x.\ \neg(x \in s)$

Union : $\forall s.\ \exists t.\ \forall x.\ x \in t = (\exists u.\ x \in u \wedge u \in s)$

Power sets : $\forall s.\ \exists t.\ \forall x.\ x \in t = x \subseteq s$

Separation : $\forall p\ s.\ \exists t.\ \forall x.\ x \in t = x \in s \wedge p\ x$

Replacement : $\forall f\ s.\ \exists t.\ \forall y.\ y \in t = \exists x.\ x \in s \wedge (y = f\ x)$

Foundation : $\forall s.\ \neg(s = \emptyset) \Rightarrow \exists x.\ x \in s \wedge (x \cap s = \emptyset)$

Infinity : $\exists s.\ \emptyset \in s \wedge \forall x.\ x \in s \Rightarrow (x \cup \{x\}) \in s$

These axioms use the auxiliary constants $\subseteq$, $\emptyset$, $\cup$, $\cap$ and singleton sets, which are easily defined prior to their use.

2.2 Auxiliary Set-theoretic Notions

Familiar set-theoretic notions can be defined from the axioms of ST using the definitional mechanisms of HOL. Some useful constants are listed below (see [11] for more explanation):

Empty set : $\forall x.\ \neg(x \in \emptyset)$

Subset relation : $s \subseteq t = \forall x.\ x \in s \Rightarrow x \in t$

Union : $x \in (s \cup t) = x \in s \vee x \in t$

Big union : $x \in \bigcup s\ =\ (\exists u.\ x \in u \wedge u \in s)$

Power set : $x \in \mathbb{P}\ s\ =\ x \subseteq s$

Set abstraction (**separation**) : $x \in \mathsf{Spec}\ s\ p\ =\ x \in s \wedge p\ x$

Simple abstraction notation : $\{x \in s \mid p[x]\}\ =\ \mathsf{Spec}\ s\ (\lambda x.\ p[x])$

Generalized abstraction notation :
$\{t[x_1,...,x_n] \in s \mid p[x_1,...,x_n]\} =$
$\text{Spec}\ s\ (\lambda x.\ \exists x_1 ... x_n.\ (x = t[x_1,...,x_n]) \wedge p[x_1,...,x_n])$

Image of a (logical) function :
$y \in \text{Image}\ f\ s \ = \ \exists x.\ x \in s \wedge (y = f\ x)$

Indexed union : $\bigcup_{x \in X} f(x) \ = \ \bigcup(\text{Image}\ f\ X)$

Infinite set :
$\emptyset \in \text{InfiniteSet} \wedge \forall x.\ x \in \text{InfiniteSet} \Rightarrow (x \cup \{x\}) \in \text{InfiniteSet}$

Finite sets : $\{x_1,\ x_2,\ \ldots\ ,\ x_n\} \ = \ \{x_1\} \cup (\{x_2\} \cup \ldots\ (\{x_n\} \cup \emptyset))$

Ordered pairs : $\langle x, y\rangle \ = \ \{\{x\},\ \{x, y\}\}$

Maplet notation : $x \mapsto y \ = \ \langle x, y\rangle$

Cartesian products :
$X \times Y \ = \ \{\langle x_1, x_2\rangle \in \mathbb{P}(\mathbb{P}(X \cup Y)) \mid x_1 \in X \wedge x_2 \in Y\}$

Relations : $X \leftrightarrow Y \ = \ \mathbb{P}(X \times Y)$

Partial functions :
$X \nrightarrow Y =$
$\{f \in X \leftrightarrow Y \mid \forall x\ y_1\ y_2.\ x \mapsto y_1 \in f \wedge x \mapsto y_2 \in f \Rightarrow (y_1 = y_2)\}$

Total functions :
$X \rightarrow Y \ = \ \{f \in X \nrightarrow Y \mid \forall x.\ x \in X \Rightarrow \exists y.\ y \in Y \wedge \langle x, y\rangle \in f\}$

Set function application : $f \diamond x \ = \ \varepsilon y.\ x \mapsto y \in f$

Set function abstraction :
$\lambda x \in X.\ f(x) \ = \ \{x \mapsto y \in X \times \text{Image}\ f\ X \mid y = f\ x\}$

Natural numbers :
$\text{num2Num}\ 0 \ = \ \emptyset$
$\text{num2Num}(\text{SUC}\ n) \ = \ \text{num2Num}\ n \cup \{\text{num2Num}\ n\}$

$\text{Num} \ = \ \{x \in \text{InfiniteSet} \mid \exists n.\ x = \text{num2Num}\ n\}$

$(\forall n.\ \text{Num2num}(\text{num2Num}\ n) \ = \ n) \wedge$
$(\forall x.\ x \in \text{Num} \ = \ (\text{num2Num}(\text{Num2num}\ x) = x))$

Note that functions of higher order logic are different from functions 'inside' set theory, which are just certain kinds of set of ordered pairs. The former will be called *logical functions* and the latter *set functions*. Logical functions have a type in HOL of the form $\alpha \to \beta$ and are denoted by terms of the form $\lambda x.\ t[x]$, whereas set functions have type V and are denoted by terms of the form $\lambda x \in X.\ s[x]$. The application of a logical function f to an argument x is denoted by $f\ x$, whereas the application of a set function s to x is denoted by $s \diamond x$.

2.3 Identity and Composition

The identity and composition functions can be defined in set theory. The identity function is defined by:

$$\mathsf{Id}\, X \ = \ \{\langle x, y\rangle \in X \times X \mid x = y\}.$$

Note that it would not be possible to define a set for Id without the set argument X since, to avoid paradoxes, sets must be built from existing ones.

The composition function could be defined in the same way, but we can exploit the presence of the (set) function arguments to avoid further arguments:

$$f \,\mathsf{O}\, g \ = \ \{\langle x, z\rangle \in \mathsf{domain}\, g \times \mathsf{range}\, f \mid \exists y.\ \langle x, y\rangle \in g \wedge \langle y, z\rangle \in f\}.$$

Here, domain and range are defined using Image by taking the first and second components of each pair in a set function. Note that the composition function is a logical function, it has type $V \to V \to V$.

2.4 Dependent Sum and Product

The dependent sum is a generalisation of Cartesian products $X \times Y$, the set of all pairs of elements in X and Y. In the dependent sum, the second component may depend on the first:

$$\sum_{x \in X} Y\ x \ = \ \bigcup_{x \in X} \bigcup_{y \in Y\ x} \{\langle x, y\rangle\}.$$

Here, $Y\ x$ can be any term of type V containing x or not. In the same way, the dependent product is a generalisation of $X \to Y$:

$$\prod_{x \in X} Y\ x \ = \ \{f \in \mathbb{P}(\sum_{x \in X} Y\ x) \mid \forall x \in X.\ \exists! y.\ \langle x, y\rangle \in f\}.$$

Clearly, both $\times$ and $\to$ are special cases of the dependent sum and the dependent product, respectively. In fact, the function abstraction introduced above yields an element of the dependent product:

$$\forall f X Y.\ (\forall x.\ x \in X \Rightarrow f\ x \in Y\ x) \Rightarrow (\lambda x \in X.\, f\ x) \in \prod_{x \in X} Y\ x.$$

The following application of HOL-ST will make essential use of the dependent product to overcome limitations in the type system of HOL's version of higher order logic.

3 Case Study: the Inverse Limit Construction

This section, which is condensed from a longer report [1], presents a case study of the application of HOL-ST to domain theory. This work was motivated by previous work on formalising domain theory in HOL [2] where it became clear that the inverse limit construction of solutions to recursive domain equations cannot be formalised in HOL easily. The type system of higher order logic is not rich enough to represent Scott's inverse limit construction directly, though other methods of solving recursive domain equations can be encoded [24]. However, the construction can be formalised in HOL-ST directly, because set theory supports general dependent products whereas higher order logic does not.

The inverse limit construction can be used to give solutions to any recursive domain (isomorphism) equation of the form

$$D \cong \mathcal{F}(D)$$

where $\mathcal{F}$ is an operator on domains, such as sum, product or (continuous) function space, or any combination of these. Thus, it might be possible to implement tools based on the present formalisation that support very general recursive datatype definitions in HOL-ST.

This will be illustrated with the construction of a domain D_∞ satisfying:

$$D_\infty \cong [D_\infty \to D_\infty],$$

where $[D \to E]$ denotes the domain of all continuous function from D to E (we shall always use $[\ \to\ ]$ for continuous functions and $\to$ for ordinary functions). Hence, D_∞ provides a model of the untyped λ-calculus.

The version of the inverse limit construction employed here is based on categorical methods using embedding-projection pairs, see e.g. [29, 25]. This was suggested by Plotkin as a generalisation of Scott's original inverse limit construction of a model of the λ-calculus in the late 60's. The formalisation is based on Paulson's accessible presentation in the book [20] but Plotkin's [25] was also used in part (in fact, Paulson based his presentation on this).

3.1 Basic Concepts of Domain Theory

Domain theory is the study of complete partial orders (cpos) and continuous functions between cpos. This section very briefly introduces the semantic definitions of central concepts of domain theory in HOL-ST.

A partial order is a pair consisting of a set and a binary relation such that the relation is reflexive, transitive and antisymmetric on all elements of the set:

$$\begin{aligned}
&\mathsf{po}\ D\ = \\
&\ \forall x \in \mathsf{set}\ D.\ \mathsf{rel}\ D\ x\ x\ \wedge \\
&\ \forall xyz \in \mathsf{set}\ D.\ \mathsf{rel}\ D\ x\ y \wedge \mathsf{rel}\ D\ y\ z \Rightarrow \mathsf{rel}\ D\ x\ z\ \wedge \\
&\ \forall xy \in \mathsf{set}\ D.\ \mathsf{rel}\ D\ x\ y \wedge \mathsf{rel}\ D\ y\ x \Rightarrow x = y.
\end{aligned}$$

The constants set and rel equal FST and SND respectively. The constant po has type $V \times (V \to V \to bool) \to bool$.

A complete[3] partial order is defined as a partial order in which all chains have least upper bounds (lubs):

$$\mathsf{cpo}\ D\ =\ \mathsf{po}\ D \wedge \forall X.\ \mathsf{chain}\ D\ X \Rightarrow \exists x.\ \mathsf{is\_lub}\ D\ X\ x,$$

where

$$\begin{array}{l}\mathsf{chain}\ D\ X\ =\ (\forall n.\ X\ n \in \mathsf{set}\ D) \wedge (\forall n.\ \mathsf{rel}\ D(X\ n)(X(n+1)))\\ \mathsf{is\_ub}\ D\ X\ x\ =\ x \in \mathsf{set}\ D \wedge \forall n.\ \mathsf{rel}\ D(X\ n)x\\ \mathsf{is\_lub}\ D\ X\ x\ =\ \mathsf{is\_ub}\ D\ X\ x \wedge \forall y.\ \mathsf{is\_ub}\ D\ X\ y \Rightarrow \mathsf{rel}\ D\ x\ y.\end{array}$$

Hence, a chain of elements of a partial order is a non-decreasing sequence of type $num \to V$.

The set of continuous functions is the subset of the set of monotonic functions that preserve lubs of chains:

$$\begin{array}{l}\mathsf{mono}(D, E)\ =\\ \quad \{f \in \mathsf{set}\ D \to \mathsf{set}\ E \mid \forall xy \in \mathsf{set}\ D.\ \mathsf{rel}\ D\ x\ y \Rightarrow \mathsf{rel}\ E(f \diamond x)(f \diamond y)\}\\ \mathsf{cont}(D, E)\ =\\ \quad \{f \in \mathsf{mono}(D, E) \mid\\ \qquad \forall X.\ \mathsf{chain}\ D\ X \Rightarrow f \diamond (\mathsf{lub}\ D\ X) = \mathsf{lub}\ E(\lambda n.\ f \diamond (X\ n))\}.\end{array}$$

Note that continuous (and monotonic) functions are set functions and therefore have type V.

The continuous function space construction on cpos is defined as the pair consisting of the set of continuous functions between two cpos and the pointwise ordering relation on functions:

$$\mathsf{cf}(D, E)\ =\ (\mathsf{cont}(D, E), \lambda fg.\ \forall x \in \mathsf{set}\ D.\ \mathsf{rel}\ E(f \diamond x)(g \diamond x)).$$

The construction always yields a cpo if its arguments are cpos.

3.2 The Inverse Limit Construction

Just as chains of elements of cpos have least upper bounds, there are "chains" of cpos that have "least upper bounds", called *inverse limits*. The ordering relation on elements is generalised to the notion of embedding morphisms between cpos. A certain constant Dinf, parametrised by a chain of cpos, can be proven once and for all to yield the inverse limit of the chain. In this section, we give a brief overview of a formalisation of the inverse limit construction.

[3] Our notion of completeness is sometimes called ω-completeness. Similarly, the chains are sometimes called ω-chains.

Embedding morphisms come in pairs with projections, forming the so-called embedding-projection pairs:

$$\begin{aligned}&\mathsf{projpair}(D,E)(e,p) \;=\\ &\quad e \in \mathsf{cont}(D,E) \wedge p \in \mathsf{cont}(E,D) \wedge\\ &\quad p \,\mathsf{O}\, e = \mathsf{Id}(\mathsf{set}\, D) \wedge \mathsf{rel}(\mathsf{cf}(E,E))(e \,\mathsf{O}\, p)(\mathsf{Id}(\mathsf{set}\, E)).\end{aligned}$$

The conditions make sure that the structure of E is richer than that of D (and can contain it). D is embedded into E by e (one-one) which in turn is projected onto D by p.

Embeddings uniquely determine projections (and vice versa). Hence, it is enough to consider embeddings

$$\mathsf{emb}(D,E)e \;=\; \exists p.\ \mathsf{projpair}(D,E)(e,p)$$

and define the associated projections, or *retracts* as they are often called, using the choice operator:

$$\mathsf{R}(D,E)e \;=\; \varepsilon p.\ \mathsf{projpair}(D,E)(e,p).$$

For readability, $\mathsf{emb}(D,E)e$ will sometimes be written using the standard mathematical notation $e : D \lhd E$. Similarly, $\mathsf{R}(D,E)e$ is sometimes written as e^R.

Embeddings are used to form chains of cpos in a similar way that the ordering on elements of cpos is used to form chains. A chain of cpos is a pair $(\mathbf{D}, \mathbf{e})$ consisting of a sequence of cpos D_n and a sequence of embeddings e_n where $e_n : D_n \lhd D_{n+1}$ for all $n : num$:

$$D_0 \overset{e_0}{\lhd} D_1 \overset{e_1}{\lhd} \cdots \overset{e_{n-1}}{\lhd} D_n \overset{e_n}{\lhd} \cdots.$$

The notion of (embedding-projection) chain of cpos is formalised as follows:

$$\mathsf{emb\_chain}\,\mathbf{D}\,\mathbf{e} \;=\; (\forall n.\ \mathsf{cpo}(\mathbf{D}\,n)) \wedge (\forall n.\ \mathsf{emb}(\mathbf{D}\,n, \mathbf{D}(\mathsf{SUC}\,n))(\mathbf{e}\,n)).$$

In HOL terms, we write $\mathbf{D}\,n$ and $\mathbf{e}\,n$ instead of D_n and e_n.

The inverse limit Dinf can now be defined as follows:

$$\begin{aligned}&\mathsf{Dinf}\,\mathbf{D}\,\mathbf{e} \;=\\ &\quad(\{x \in \prod_{n \in \mathrm{Num}} \mathsf{set}(\mathbf{D}(\mathsf{Num2num}\ n)) \;\mid\; \cdots\},\\ &\qquad\lambda xy.\, \forall n.\ \mathsf{rel}(\mathbf{D}\,n)(x\diamond(\mathsf{num2Num}\ n))(y\diamond(\mathsf{num2Num}\ n))),\end{aligned}$$

where the dependent product construction on sets is used. Informally, the underlying set of Dinf is defined as the subset of all infinite tuples x on which the n-th projection e_n^R maps the $(n+1)$-st index to the n-th index for all n: $e_n^R(x_{n+1}) = x_n$. The underlying relation is defined componentwise. The annoying $\mathsf{num2Num}$ conversions, which also appear in the omitted part, could be avoided by using the set of numbers Num instead of the type of numbers *num* to represent chains of cpos. However, the present choice makes proofs simpler. This issue is discussed further in Section 4.

The details of proving the property that Dinf yields the inverse limit of a given chain of cpos will not be given here (see [1]). Informally speaking, it must satisfy a certain commutivity condition, which says that there is a family of embeddings from the elements of the chain to Dinf such that the resulting diagrams commute, and it must be universal with this property in the sense that there is a unique embedding of Dinf into any other cpo with this property.

3.3 Construction of D_∞

In general, a recursive domain (isomorphism) equation can have the form

$$D \cong \mathcal{F}(D)$$

where $\mathcal{F}$ is a continuous covariant functor (a term of category theory). In particular, this means that $\mathcal{F}$ preserves inverse limits of chains of cpos and consists of both a construction on cpos and a construction on embeddings. Note that a domain equation is stated using only the cpo construction part of a functor.

The inverse limit construction provides solutions of any domain equation of the above form. We shall sketch very roughly how to obtain a solution of the equation

$$D \cong [D \to D],$$

which is stated using the standard notation for the continuous function space construction (defined as cf in HOL-ST). It can be proved once and for all that the continuous function space construction with an appropriate embedding is a continuous covariant functor.

First, a specific (but still parametrised) chain is constructed by iterating the continuous function space functor, starting at any cpo D with an embedding $e : D \lhd \mathsf{cf}(D, D)$. From this chain of cpos we obtain a partially concrete instantiation of Dinf, called Dinf_cf, which, due to the fact that the functor for the function space preserves inverse limits, yields a parametrised model of the untyped λ-calculus:

$$\begin{array}{l}\forall De.\\ \quad \mathsf{cpo}\ D \Rightarrow \mathsf{emb}(D, \mathsf{cf}(D, D))e \Rightarrow\\ \quad \mathsf{Dinf\_cf}\ D\ e \cong \mathsf{cf}(\mathsf{Dinf\_cf}\ D\ e, \mathsf{Dinf\_cf}\ D\ e).\end{array}$$

Choosing one concrete starting point (with for instance one element), and defining D_∞ to abbreviate the corresponding instantiation of Dinf_cf, a concrete nontrivial model of the untyped λ-calculus is obtained:

$$D_\infty \cong \mathsf{cf}(D_\infty, D_\infty).$$

(See [1] for further details.)

4 Set Theory in Isabelle

Isabelle/ZF supports set theory in first order logic [21, 22]; it is an extension of a first order logic instantiation of the generic theorem prover Isabelle [23] with axioms of Zermelo-Fraenkel set theory. In contrast, HOL-ST supports set theory in higher order logic. Formalising the inverse limit construction in Isabelle/ZF revealed various differences between the two systems. This section summarises the conclusions of the comparison presented in [3].

4.1 First-order versus Higher-order Set Theory

The formalisation presented above exploited higher order logic as much as possible. Hence, cpos were HOL pairs, ordering relations were HOL functions and chains were HOL functions from the HOL type of natural numbers *num* to *V*. Alternatively, we could have chosen to do more work in set theory. For instance, the natural number argument of chains could have been in the set Num and cpos could have been represented by (non-reflexive) relations of set theory.

It makes a difference whether sets of ST or types of HOL are used. Using the latter, set membership conditions are avoided and furthermore, type checking is done automatically in ML. Using sets, type checking, i.e. ensuring terms are in the right sets, is done by theorem proving, and it is done late.

Obviously, exploiting the additional power of set theory will require leaving higher order logic and paying a price. It is an interesting and difficult question which parts of the formalisation should be done in set theory and which should be done in higher order logic. As noted in Section 3.2, the definition of the inverse limit constructor Dinf could have been simplified if chains of cpos had been represented using the set of natural numbers instead of the HOL type. However, experiments showed that it was worth paying the inconvenience of the translation functions at this stage since many theorems and proofs became quite horrible later, with the set approach. This in turn is related to the choice of representing ordinary chains in higher order logic; had they been represented using set theory numbers it would have been more natural to stick to this.

In Isabelle/ZF, there is much less choice. Most of the development must be done in set theory since the first order logic is so weak; for instance, it does not provide natural numbers, pairs, or functions (the individuals of the logic are sets). Therefore, chains must be represented as set functions and cpos must be represented as set theory pairs where both the set and the relation components are sets. In principle, the function type of Isabelle's polymorphic higher-order meta logic could be used to represent chains, but the definition of cpos must quantify over chains and in first order logic it is only possible to quantify over individuals (sets), not meta level functions. The meta logic is meant for expressing and reasoning in logic instantiations of Isabelle, the so-called object logics, not for formalising concepts in object logics.

The consequence of doing more work in set theory is that terms and proofs become more complicated, since set membership conditions appear more often. For instance, to prove that a term is a chain it must be shown that it is a set

function, i.e. that it is a relation on the right sets that defines a function (though, usually the λ-abstraction is used and then it is only necessary to type check the body of this, due to a pre-proved theorem). Similarly, constructions on cpos have more complicated definitions and proofs: before applying the relation of a cpo to its arguments, these must be shown to be in the right sets.

4.2 Proof Support

Whilst there are benefits from the more powerful higher order logic of HOL-ST, Isabelle/ZF has the advantage of providing better proof support for set theory. Though proofs in principle are longer in terms of number of proof steps (of applying theorems), due to the additional type conditions, they are in fact much shorter in terms of lines, and easier to write. Note that the disadvantage of Isabelle/ZF mentioned above is logic dependent and could be avoided by using Isabelle/HOL, which supports a HOL-like higher order logic, rather than Isabelle/FOL as a basis for ZF set theory (though this would introduce translation functions as in HOL-ST). The better proof support of Isabelle/ZF is not logic dependent and thus would be preserved if we moved to Isabelle/HOL.

Isabelle provides an elegant proof infrastructure. Meta logic theorems can express both object logic theorems, inference rules and tactics. It does not implement a separate inference rule and tactic for each operation as in HOL; instead, the same theorem is applied either in a forward or backward fashion. Hence, there are very few tools for forward and backward proof. In fact, the main way to prove theorems is by the principle of resolution, which supports both styles.

The notion of resolution in Isabelle supports 'real' backward proofs better than they are supported in HOL. In Isabelle/ZF, one almost always works from the conclusion of a goal backward towards the assumptions, due to the design of Isabelle resolution tactics. In HOL-ST, one often ends up doing a lot of sometimes ugly assumption hacking working forward from the assumptions towards the conclusion, due to the design of HOL resolution tactics. More natural backward strategies like conditional rewriting and a matching modus ponens style strategy like `MATCH_MP_TAC`, which may be viewed as a simplified form of Isabelle resolution, are not well supported in HOL. These strategies are particularly useful in HOL-ST, compared to HOL, due the fact that many theorems have set membership assumptions.

The backward proofs of the formalization were usually more than 50% shorter in Isabelle/ZF (we used the default subgoal package of HOL). Its subgoal module provides a kind of flat structure on proof states where all subgoals can be accessed directly by an index specified by tactics; proving subgoal i results in all subgoals with a higher index to be shifted such that the indexing contains no holes. This makes it possible to access all subgoals, possibly at the same time, and to prove several subgoals by just repeating a tactic supplied with the right list of theorems—no matter where they would appear in a HOL proof tree.

Another reason why proofs are shorter is that usually the instantiation of both existentially and universally quantified variables is handled automatically in Isabelle. It provides a notion of unknown variables which can be instantiated

in proofs. This means that witnesses and proper instantiations are constructed behind the scenes, possibly in stages. Unification is essential for allowing this. Existential quantifiers are often introduced by backward strategies, for instance, when employing the transitivity of a cpo relation or the facts that function composition preserves the function set, continuity or embeddings: such theorems have free variables in the antecedents that do not appear in the consequent.

HOL-ST proofs could probably be simplified if there was a way of handling existential quantifiers. At the moment, witnesses must be provided on the spot and manually (some user-contributed tools for solving existential goals automatically are available, but we have not tried them with HOL-ST). Furthermore, it is often necessary to instantiate universally quantified variables manually.

5 Conclusions

The case study on formalising a model of the λ-calculus via the inverse limit construction shows that combining set theory and higher order logic in the same theorem prover provides a useful system for doing mathematics. The simplicity and convenience of higher order logic can be exploited as well as the expressive power of set theory. Rather than working in set theory only, it was shown that set and type theoretic reasoning can be mixed to advantage by exploiting set theory only when it is necessary and working in higher order logic the rest of the time. The logic of Isabelle's set theory, being first order, is weaker and therefore larger parts of the formalisation had to be done in set theory.

One of the main disadvantages of set theory is the presence of explicit type (set membership) conditions. This means that type checking is done late by theorem proving whereas in higher order logic type checking is done early in ML. Furthermore, type checking is automatic in HOL but cannot be fully automated in set theory. Thus, the comparison with Isabelle/ZF showed that proofs required more theorems since certain concepts were formalised in logic in HOL which had to be represented in set theory in Isabelle/ZF (since its first order logic is weaker). On the other hand, the comparison also showed that Isabelle/ZF supports proofs in set theory much better than HOL-ST, which generally speaking lacks ways of handling conditional theorems conveniently, and also does not provide any support for unknown variables for quantifier reasoning like in Isabelle/ZF. Since the better proof support in Isabelle/ZF is not logic dependent, extending Isabelle/HOL with ZF set theory might be a way of combining the benefits of HOL-ST and Isabelle/ZF. However, this would introduce the disadvantage of translation functions between types and sets as in HOL-ST.

In summary: it is not yet clear to us whether set theory in higher order logic is right, or just more support for set theory in first order logic is needed. Points requiring further consideration include the following.

1. ST in HOL yields a need for ugly translation functions (num2Num etc). Can these be hidden by suitable parsing and printing functions together with specialised theorem proving support?

2. ST in FOL yields a lot of set membership conditions. Can these be better automated, perhaps with theorem proving tools that mimic 'type checking'?
3. The construction of D_∞ might be possible in a type theory with dependent types. Are there natural examples that really require set theory? Embedding Z semantics is a candidate.

Acknowledgements

The research described here is partly supported by EPSRC grant `GR/G23654`, partly by an HCMP fellowship under the EuroForm network, and partly by BRICS[4]. We are grateful to Larry Paulson, who read a preliminary draft of this paper and suggested some improvements. Francisco Corella, whose PhD thesis [6] contains many insights into mechanizing set theory (including the use of simple type theory as the underlying logic) has had a significant influence on the second author's thinking.

References

1. S. Agerholm. Formalising a model of the λ-calculus in HOL-ST. Technical Report 354, University of Cambridge Computer Laboratory, November 1994.
2. S. Agerholm. *A HOL Basis for Reasoning about Functional Programs*. PhD thesis, BRICS, Department of Computer Science, University of Aarhus, December 1994. Available as Technical Report RS-94-44.
3. S. Agerholm. A comparison of HOL-ST and Isabelle/ZF. Technical Report 369, University of Cambridge Computer Laboratory, 1995.
4. D. Cantone, A. Ferro, and E. Omodeo, editors. *Computable Set Theory*, volume 1. Clarendon Press, Oxford, 1989.
5. R. L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, 1986.
6. F. Corella. Mechanizing set theory. Technical Report 232, University of Cambridge Computer Laboratory, 1991.
7. G. Dowek, A. Felty, H. Herbelin, G. Huet, C. Murthy, C. Parent, C. Paulin-Mohring, and B. Werner. The Coq proof assistant user's guide - version 5.8. Technical Report 154, INRIA-Rocquencourt, 1993.
8. Roman Matuszewski (ed). *Formalized Mathematics*. Université Catholique de Louvain, 1990 –. Subscription is $10 per issue or $50 per year (including postage). Subscriptions and orders should be addressed to: Fondation Philippe le Hodey, MIZAR, Av.F.Roosevelt 35, 1050 Brussels, Belgium (fax: +32 (2) 640.89.68).
9. W. M. Farmer, J. D. Guttman, and F. Javier Thayer. IMPS: An interactive mathematical proof system. *Journal of Automated Reasoning*, 11(2):213–248, 1993.
10. S. Finn and M. P. Fourman. *L2 – The LAMBDA Logic*. Abstract Hardware Limited, September 1993. In LAMBDA 4.3 Reference Manuals.
11. M. J. C. Gordon. Merging HOL with set theory: preliminary experiments. Technical Report 353, University of Cambridge Computer Laboratory, 1994.

[4] Basic Research in Computer Science, Centre of the Danish National Research Foundation.

12. M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: A Theorem-proving Environment for Higher-Order Logic*. Cambridge University Press, 1993.
13. F. K. Hanna, N. Daeche, and M. Longley. Veritas+: a specification language based on type theory. In M. Leeser and G. Brown, editors, *Hardware specification, verification and synthesis: mathematical aspects*, volume 408 of *Lecture Notes in Computer Science*, pages 358–379. Springer-Verlag, 1989.
14. C. B. Jones. *Systematic Software Development using VDM*. Prentice Hall International, 1990.
15. L. Lamport. TLA+. Available on the World Wide Web at the URL: `http://www.research.digital.com/SRC/tla/tla.html`.
16. Z. Luo and R. Pollack. LEGO proof development system: User's manual. Technical Report ECS-LFCS-92-211, University of Edinburgh, LFCS, Computer Science Department, University of Edinburgh, The King's Buildings, Edinburgh, EH9 3JZ, May 1992.
17. L. Magnusson and B. Nordström. The ALF proof editor and its proof engine. In *Types for Proofs and Programs: International Workshop TYPES '93*, number 806 in Lecture Notes in Computer Science, pages 213–237. Springer-Verlag, 1994.
18. D. A. McAllester. *ONTIC: A Knowledge Representation System for Mathematics*. MIT Press, 1989.
19. P. M. Melliar-Smith and John Rushby. The enhanced HDM system for specification and verification. In *Proc. Verkshop III*, volume 10 of *ACM Software Engineering Notes*, pages 41–43. Springer-Verlag, 1985.
20. L. C. Paulson. *Logic and Computation: Interactive Proof with Cambridge LCF*. Cambridge Tracts in Theoretical Computing 2, Cambridge University Press, 1987.
21. L. C. Paulson. Set theory for verification: I. From foundations to functions. *Journal of Automated Reasoning*, 11(3):353–389, 1993.
22. L. C. Paulson. Set theory for verification: II. Induction and Recursion. Technical Report 312, University of Cambridge Computer Laboratory, 1993.
23. L. C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.
24. K. D. Petersen. Graph model of lambda in higher order logic. In J. J. Joyce and C. H. Seger, editors, *Proceedings of the 6th International Workshop on Higher Order Logic Theorem Proving and its Applications*, volume 780 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.
25. G. Plotkin. *Domains*. Course notes, Department of Computer Science, University of Edinburgh, 1983.
26. PVS World Wide Web page. `http://www.csl.sri.com/pvs/overview.html`.
27. Piotr Rudnicki. *An Overview of the MIZAR Project*. Unpublished; but available by anonymous FTP from `menaik.cs.ualberta.ca` in the directory `pub/Mizar/Mizar_Over.tar.Z`, 1992.
28. M. Saaltink. Z and EVES. Technical Report TR-91-5449-02, Odyssey Research Associates, 265 Carling Avenue, Suite 506, Ottawa, Ontario K1S 2E1, Canada, October 1991.
29. M. Smyth and G. D. Plotkin. The category-theoretic solution of recursive domain equations. *SIAM Journal of Computing*, 11, 1982.
30. J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall International Series in Computer Science, 2nd edition, 1992.

This article was processed using the LaTeX macro package with LLNCS style

Recent Publications in the BRICS Report Series

RS-95-37 **Sten Agerholm and Mike Gordon.** ***Experiments with ZF Set Theory in HOL and Isabelle*****. July 1995. 14 pp. To appear in** ***Proceedings of the 8th International Workshop on Higher Order Logic Theorem Proving and its Applications,*** **LNCS, 1995.**

RS-95-36 **Sten Agerholm.** ***Non-primitive Recursive Function Definitions*****. July 1995. 15 pp. To appear in** ***Proceedings of the 8th International Workshop on Higher Order Logic Theorem Proving and its Applications,*** **LNCS, 1995.**

RS-95-35 **Mayer Goldberg.** ***Constructing Fixed-Point Combinators Using Application Survival*****. June 1995. 14 pp.**

RS-95-34 **Jens Palsberg.** ***Type Inference with Selftype*****. June 1995. 22 pp.**

RS-95-33 **Jens Palsberg, Mitchell Wand, and Patrick O'Keefe.** ***Type Inference with Non-structural Subtyping*****. June 1995. 22 pp.**

RS-95-32 **Jens Palsberg.** ***Efficient Inference of Object Types*****. June 1995. 32 pp. To appear in** ***Information and Computation*****. Preliminary version appears in** ***Ninth Annual IEEE Symposium on Logic in Computer Science,*** **LICS '94 Proceedings, pages 186–195.**

RS-95-31 **Jens Palsberg and Peter Ørbæk.** ***Trust in the λ-calculus*****. June 1995. 32 pp. To appear in** ***Static Analysis: 2nd International Symposium,*** **SAS '95 Proceedings, 1995.**

RS-95-30 **Franck van Breugel.** ***From Branching to Linear Metric Domains (and back)*****. June 1995. 30 pp. Abstract appeared in Engberg, Larsen, and Mosses, editors,** ***6th Nordic Workshop on Programming Theory,*** **NWPT '6 Proceedings, 1994, pages 444-447.**

RS-95-29 **Nils Klarlund.** ***An $n \log n$ Algorithm for Online BDD Refinement.*** **May 1995. 20 pp.**

RS-95-28 **Luca Aceto and Jan Friso Groote.** ***A Complete Equational Axiomatization for MPA with String Iteration*****. May 1995. 39 pp.**