Merging HOL with Set Theory

preliminary experiments

Mike Gordon

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

Email: `Mike.Gordon@cl.cam.ac.uk`
WWW: `http://www.cl.cam.ac.uk/users/mjcg/`

Abstract

Set theory is the standard foundation for mathematics, but the majority of general purpose mechanised proof assistants support versions of type theory (higher order logic). Examples include Alf, Automath, Coq, EHDM, HOL, IMPS, LAMBDA, LEGO, Nuprl, PVS and Veritas. For many applications type theory works well and provides, for specification, the benefits of type-checking that are well-known in programming. However, there are areas where types get in the way or seem unmotivated. Furthermore, most people with a scientific or engineering background already know set theory, whereas type theory may appear inaccessable and so be an obstacle to the uptake of proof assistants based on it. This paper describes some experiments (using HOL) in combining set theory and type theory; the aim is to get the best of both worlds in a single system. Three approaches have been tried, all based on an axiomatically specified type V of ZF-like sets: (i) HOL is used without any additions besides V; (ii) an embedding of the HOL logic into V is provided; (iii) HOL axiomatic theories are automatically translated into set-theoretic definitional theories. These approaches are illustrated with two examples: the construction of lists and a simple lemma in group theory.

Contents

1 Introduction

Set theory is the standard foundation for mathematics. Formal notations like Z [33], VDM [19] and TLA+ [20] are based on sets. However, most general purpose mechanised proof assistants support typed higher order logics (i.e. type theories). Examples include Alf [22], Coq [8], EHDM [26], HOL [13], IMPS [11], LAMBDA [12], LEGO [21], Nuprl [6], PVS [30] and Veritas [16]. The reasons for this are, I think, varied:

- Functions are a pervasive concept in computer science and so taking them as primitive, as is done by (most forms of) type theory, is natural. In set theory, functions are a derived notion (sets of ordered pairs) and it is tedious to have to derive basic laws like β-conversion.
- Types are an accepted and effective method of structuring data and type-checking is a powerful technique for finding errors. For example, types can be used to index terms and formulae for efficient retrieval. Type theories come with a type discipline built-in, set theory doesn't.
- General laws become simpler when typed. For example, $x + 0 = 0$ is an equation if x has type $\mathbb{N}$, but in untyped set theory the corresponding fact would be the implication $x \in \mathbb{N} \Rightarrow x + 0 = 0$. Determining if an equation can be used to rewrite a term involves only matching. The applicability of a conditional equation requires theorem proving to determine whether the condition holds.
- Much research in logic for computer science has a type-theoretic flavour. Type theories are thus a hot topic and are perhaps considered more exciting and modern than set theory.

For many applications type theory works well, but there are areas where it becomes cumbersome. Certain classical constructions, like the definition of the natural numbers as the set `{∅, {∅}, {∅,{∅}}, {∅,{∅},{∅,{∅}}}, ··· }`, 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'.

Another problem with type theory is its lack of a standard formulation. There are lots of different type theories, based on a wide variety of philosophical conceptions. The proof tools listed above are based on many different theories.

- Automath is based on de Bruijn's own very general logic (which anticipated many more recent developments).

- Alf, Coq and LEGO support different versions of the Calculus of Constructions.
- EHDM, PVS and Veritas each support different classical higher order logics with dependent types.
- HOL and LAMBDA support similar polymorphic versions of the simple theory of types.
- IMPS supports simple type theory with non-denoting terms and a theory interpretation mechanism.
- Nuprl supports Martin Löf type theory (a constructive logic with a very elaborate type system).

Becoming fluent in using a logical system can require a large investment of time, which one does not want to make only to find out later that one has backed the wrong theory!

In contrast to the situation with type theory, there is much less variation among set theories. The well known formulations are, for practical purposes, pretty much equivalent. They are all defined by axioms in predicate calculus. The only variations are (i) whether proper classes are in the object or meta language and (ii) how many large cardinals are postulated to exist. The vast majority of mathematicians are happy with ZFC (Zermelo-Fraenkel set theory with the Axiom of Choice). The various type theories differ not only in the ontological assumptions made but also in the kind of language used to express these assumptions; they even differ in their underlying philosophical conception of mathematical truth (e.g. intuitionistic/constructive versus classical).

Several proof assistants for set theory exist. In the formal methods community [1] the best known are probably Isabelle [29], a generic system that supports various kinds of type theory as well as ZF set theory, and EVES [32]. Both of these have considerable automation and are capable of proving difficult theorems. The Mizar system from Poland [31] is a low level proof checker based on set theory that has been used to check enormous amounts of mathematics – there is even a journal devoted to it [9]. Mizar has only recently become well-known in the theorem proving community. It employs some kind of type system on top of its set theory and so is likely to contain ideas relevant to the topic of this paper. However, I have not yet learned enough about Mizar to make further comments. 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

---

[1] In other communities (e.g. artificial intelligence) there is related work on theorem proving for set theory (e.g. Ontic [23]).

various styles of type theory (including HOL-like higher order logic and Martin Löf type theory), dependending 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] Isabelle users liken set theory to machine code and type theory to a high-level language.

Type theories can be classified into those whose semantics is given in terms of set theory and those whose semantics is essentially non set-theoretic. The former includes the logics supported by EHDM, HOL, IMPS, LAMBDA, PVS and (possibly) Veritas; the latter includes the logics of Alf, Coq, LEGO and Nuprl. There is a straightforward way of combining type theories that have a set-theoretic semantics with raw set theory: regard the type theory as a layer of defined notations on top of set theory. If this view is taken, then type checking reduces to being a special case of ordinary theorem proving (namely, the proving of set membership conditions). It is much less clear how to combine non set-theoretic type theories with ordinary set theory.

This paper reports on some experiments in combining the HOL logic (a form of simple type theory with a set-theoretic semantics) with a ZFC-like set theory. The goal is to provide an extension of HOL that enables set-theoretic reasoning to be harmoniously blended with the kind of type-theoretic reasoning that is familiar to HOL users. The challenge of this work is not theoretical, but lies in engineering a smooth connection between the existing HOL logic and set theory. The presence of set theory should not interfere with standard HOL-style reasoning (or invalidate existing proofs); rather it should provide the possibility of cleanly combining new set-theoretic methods of definition and proof with familiar type-theoretic reasoning. To avoid starting from scratch, an existing version of HOL (namely HOL88.2.02) has been used for the experiments. If they are deemed successful, then a foundation will have been laid for a successor system in which set theory is the core formal system and HOL is just an application (i.e. a library) mounted on top of it. Other type theories (e.g. PVS and Veritas, which have subtypes and dependent types) could then be implemented as libraries. Treating HOL as an application built on top of another logic (the metalogic) is reminiscent of Isabelle. The difference is that Isabelle's metalogic is quite weak but set theory, which plays the role of the metalogic here, is very

---

[2] Note, however, that explicit proof of membership conditions in set theory results in more being formally proved than with typechecking, 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.

strong. However, Isabelle's ZF is a strong candidate for being the platform on which to build the final system.

In Section 2 a theory of sets in HOL is described. The standard axioms (extensionality, union, power sets, replacement, foundation and infinity) are postulated about a new constant $\in : V \times V \to bool$, where V is a new HOL type, whose members are to be interpreted as sets. Further axioms such as the Axiom of Choice and the existence of 'universes' are discussed in Section 2.2.

In Section 3 some standard notions are developed on top of the axioms. These provide the usual repertoire of set-theoretic concepts (subsets, pairs, products, power sets, functions, etc). A set comprehension notation is also introduced, using HOL's built-in support for set abstraction syntax (a derived rule for its elimination has been programmed).

In Section 4 lists are constructed inside V, to illustrate how monomorphic HOL types can be defined in terms of sets. This does not require any additions to the HOL system beyond the declaration of V and the various set-theoretic axioms.

In Section 5 the systematic translation of HOL types and terms into V is discussed.

In Section 6, some transfer principles between the HOL logic and V are described. These allows theorems to be systematically moved between type theory and set theory, enabling proofs of set membership to be converted into typechecking. Operations on sets can be converted into HOL type operators, and so polymorphic types can be defined. The transfer principles are illustrated (i) by showing how they can be used to define polymorphic lists by a set-theoretic construction and (ii) via a simple group-theoretic example.

In Section 7, a partially-implemented mechanism for translating 'abstract' HOL theories to definitional set-theoretic theories is outlined. This translation converts types and type operators to sets and functions on sets, respectively; and axioms of the 'abstract' theory are translated to assumptions of theorems in set theory. This provides a facility offering some of the power of both the 'theory interpretations' of IMPS [10] and 'abstract theories' in HOL [34, 15]. It is briefly illustrated using the group theory example.

The experimental system obtained by adding the set theory embodied in the type V to HOL will be called HOL-ST.

2 A ZF-like set theory in HOL

Some of the details in this section are taken from Johnstone's book [18] and Paulson's paper [27].

Because the axioms are expressed in higher order logic they are probably[3] equiv-

---

[3] I do not know enough set theory to be more precise.

alent to ZFC plus the existence of at least one large cardinal.

2.1 Standard axioms of set theory

This section contains the axioms of a ZF-like set theory expressed in the HOL logic. These axioms are postulated about a new type V and a new constant $\in\ :\ V\times V\ \rightarrow\ bool$. The axioms make the type V a model of ZF and thus imply that the universe $\mathcal{U}$ used by Pitts in the book *Introduction to HOL* [13] to provide a semantics for HOL must be quite rich (e.g. contain an inaccessible cardinal). The remarks at the end of Section 15.1 of *Introduction to HOL* assume all theories are purely definitional, and so do not apply here, (since the theory introducing V is not definitional).

Because they are formulated in higher order logic, the ZF-like axioms described below are strictly stronger than ZF (see Section 2.1.6) and thus the resulting set theory will be called ST to avoid confusion. The axiom of choice is not stated explicitly, but is implied by existence of the choice function (ε-operator) in HOL (see Section 2.2). The axioms of ST in the HOL logic are first listed and then each explained in a separate subsection.

`Extensionality`	`∀s t. (s = t) = (∀x. x ∈ s = x ∈ t)`
`Empty set`	`∃s. ∀x. ¬(x ∈ s)`
`Union`	`∀s. ∃t. ∀x. x ∈ t = (∃u. x ∈ u ∧ u ∈ s)`
`Power sets`	`∀s. ∃t. ∀x. x ∈ t = x ⊆ s`
`Separation`	`∀p s. ∃t. ∀x. x ∈ t = x ∈ s ∧ p x`
`Replacement`	`∀f s. ∃t. ∀y. y ∈ t = ∃x. x ∈ s ∧ (y = f x)`
`Foundation`	`∀s. ¬(s = ∅) ⇒ ∃x. x ∈ s ∧ (x ∩ s = ∅)`
`Infinity`	`∃s. ∅ ∈ s ∧ ∀x. x ∈ s ⇒ (x ∪ {x}) ∈ s`

2.1.1 Axiom of extensionality

Two sets are equal if and only if they have the same members.

Extensionality

```
∀s t. (s = t) = (∀x. x ∈ s = x ∈ t)
```

2.1.2 Axiom of empty set

A set containing no elements, the empty set, is postulated to exist.

Empty set

$$\exists s.\ \forall x.\ \neg(x \in s)$$

This axiom legitimates the introduction of a constant $\emptyset$ with the property:

$$\forall x.\ \neg(x \in \emptyset)$$

2.1.3 Axiom of union

For a set of sets `s` there exists a set `t` that is the union of all the members of `s`, i.e. `x` is a member of `t` iff `x` is a member of some set, `u` say, in `s`.

Union

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

This axiom legitimises the introduction of a constant $\bigcup$ such that:

$$\forall s\ x.\ x \in \bigcup s = (\exists u.\ x \in u \wedge u \in s)$$

2.1.4 Axiom of power sets

For any set `s` there exists a set `t` whose members are the subsets of `s`. First the subset relation is defined by:

$$s \subseteq t = \forall x.\ x \in s \Rightarrow x \in t$$

The axiom is then:

Power sets

$$\forall s.\ \exists t.\ \forall x.\ x \in t = x \subseteq s$$

This axiom legitimates the introduction of a constant $\mathbb{P}$ with the property:

$$\forall s\ x.\ x \in \mathbb{P}\ s = x \subseteq s$$

2.1.5 Axiom of separation

If p is a property and s a set, then the axiom of separation says that there exists a subset t of s consisting of those members x that satisfy p.

Separation

$$\forall p\ s.\ \exists t.\ \forall x.\ x \in t = x \in s \land p\ x$$

This axiom legitimates the introduction of a constant Spec satisfying:

$$\forall s\ p\ x.\ x \in \mathsf{Spec}\ s\ p = x \in s \land p\ x$$

Thus $\mathsf{Spec}\ s\ p$ denotes the subset of s satisfying the predicate p. The notation $\{x \in s \mid p[x]\}$, where $p[x]$ is a formula containing a free variable x, denotes $\mathsf{Spec}\ s\ (\lambda x.\ p[x])$ i.e. the subset of s consisting of those x that satisfy $p[x]$.

More generally $\{t[x_1,\ldots,x_n] \in s \mid p[x_1,\ldots,x_n]\}$ denotes:

$$\{x \in s \mid \exists x_1 \ldots x_n.\ (x = t[x_1,\ldots,x_n]) \land p[x_1,\ldots,x_n]\}$$

i.e. $\mathsf{Spec}\ s\ (\lambda x.\ \exists x_1 \ldots x_n.\ (x = t[x_1,\ldots,x_n]) \land p[x_1,\ldots,x_n])$

2.1.6 Axiom of replacement

The idea underlying the axiom of replacement is that if f is a function and s is a set, then the image of s under f should also be a set.

Replacement

$$\forall f\ s.\ \exists t.\ \forall y.\ y \in t = \exists x.\ x \in s \land (y = f\ x)$$

Note that the universal quantification is over total functions. If the quantification is extended to include partial functions (which can be done in HOL by representing partial functions as single-valued relations) then the axiom of empty set and the axiom of separation follow, and so need not be postulated separately. For example, the empty set can be shown to exist by specialising f to the everywhere-undefined function.

The universally quantified f in the axiom of replacement can be specialised to terms containing higher-order variables. This possibility makes the set theory presented here strictly stronger than ZFC, since such specialisations go beyond the possible instances of the first-order formulations of replacement. An elegant and concise discussion of this point can be found in Paulson's paper on ZF in Isabelle [27].

The axiom of replacement legitimates the introduction of a constant Image with the property:

```
∀f s y. y ∈ Image f s = ∃x. x ∈ s ∧ (y = f x)
```

2.1.7 Axiom of foundation

The elements of type V should be thought of as built in 'stages', starting from the empty set and then using the various axioms. Every set `s` is constructed 'after' its members are constructed, and so every non-empty set must contain a member, `x` say, consisting of sets constructed 'earlier' than `s`, and so `x` and `s` must be disjoint. To formalise this, the notion of set intersection must first be defined. The intersection of `s` and `t` is the subset of `s` consisting of elements that are in `t`, hence by separation:

```
∃f. ∀s t x. x ∈ f s t = x ∈ s ∧ x ∈ t
```

which legitimises the introduction of a constant ∩ satisfying:

```
∀s t x. x ∈ (s ∩ t) = x ∈ s ∧ x ∈ t
```

The axiom of foundation is then:

Foundation

```
∀s. ¬(s = ∅) ⇒ ∃x. x ∈ s ∧ (x ∩ s = ∅)
```

2.1.8 Axiom of infinity

The preceding axioms do not entail the existence of an infinite set. The axiom of infinity postulates the existence of a set containing:

```
∅,  {∅},  {∅,{∅}},  {∅,{∅},{∅,{∅}}}, ...
```

To formalise this, it is convenient to first define (i) singleton sets and (ii) the union of two sets. If `s` is any set, then the axiom of replacement ensures that the image of $\mathbb{P}\emptyset$ under the constant function λx.`s` is a set. It is clearly the singleton set containing `s`. This establishes:

```
∀s. ∃t. ∀x. x ∈ t = (x = s)
```

which legitimates the introduction of a constant Singleton satisfying:

$$\forall s\ x.\ x \in \mathsf{Singleton}\ s = (x = s)$$

the notation $\{s\}$ is equivalent to Singleton s.

To define the union of two sets it is sufficient to first establish:[4]

$$\exists f.\ \forall s\ t\ x.\ x \in f\ s\ t = x \in s \lor x \in t$$

and then to use this to legitimise the introduction of a constant $\cup$ satisfying:

$$\forall s\ t\ x.\ x \in (s \cup t) = x \in s \lor x \in t$$

The axiom of infinity can now be stated as:

Infinity

$$\exists s.\ \emptyset \in s \land \forall x.\ x \in s \Rightarrow (x \cup \{x\}) \in s$$

This axiom legitimates the introduction of a constant InfiniteSet satisfying:

$$\emptyset \in \mathsf{InfiniteSet} \land \forall x.\ x \in \mathsf{InfiniteSet} \Rightarrow (x \cup \{x\}) \in \mathsf{InfiniteSet}$$

2.2 Other axioms

The eight axioms just described are standard, though (as already discussed in Section 2.1.6) their statement in higher order logic makes the resulting set theory more powerful than first order ZF.

Another standard axiom is the axiom of choice. This comes in various forms, the strongest of which is Global Choice:

Global Choice

$$\exists f{:}V\to V.\ \forall s.\ \neg(s = \emptyset) \Rightarrow f\ s \in s$$

However, the version of higher order logic in the HOL system contains a choice function ε as a primitive constant. The notation $\varepsilon x.\ x \in s$ denotes a member of s if one exists, otherwise it denotes an arbitrary value. The presence of this notation entails:

---

[4] The proof of this is slightly tricky. The one done in HOL involves first defining pairs.

```
∀s. ¬(s = ∅) ⇒ (εx. x ∈ s) ∈ s
```

Global Choice follows directly from this.

Working mathematicians sometimes assume an axiom postulating that each set is contained in a 'universe'. For example, Cohn in his textbook *Universal Algebra* [5] says (I have slightly edited the following quote):

> A set U is said to be *universal* or a *universe*, if it satisfies the following conditions:
>
> (i) If $X \in U$, then $X \subseteq U$.
> (ii) If $X \in U$, then $\mathbb{P}(X) \in U$.
> (iii) If $X, Y \in U$, then $\{X, Y\} \in U$.
> (iv) If $F = (F_i)_{i \in I}$, where $F_i \in U$ and $I \in U$, then $\bigcup F \in U$.

Cohn then goes on to postulate the following axiom of infinity:

> Every set is a member of some universe.

Similar axioms are common in category theory books, where universes are often called 'Grothendieck universes'. Sometimes the axiom above is assumed; sometimes, following Mac Lane, it is just assumed that at least one universe exists. Typical discussions can be found in McLarty's book *Elementary Categories, Elementary Toposes* [25] and Borceux's volume entitled *Basic Category Theory* in the *Encyclopedia of Mathematics and its Applications* [2].

Universe existence axioms are introduced, in part, as an alternative to proper classes. In systems like HOL and Isabelle, classes correspond to unary predicates in the metalogic [27, Section 3.1]. The availability of such predicates weakens the motivation for these axioms, but maybe they can sometimes provide a simpler alternative to the cumulative hierarchy used to construct bounds to fixed points in Paulson's elegant construction of datatypes [28]?

From the perspective of a logician, the uncertainty of the exact consistency strength of ST and its non-standard nature may be upsetting. However, from an 'engineering' perspective it can be argued that one wants the most powerful tools available subject, of course, to their being logically sound. If formulae are easier to write and proofs are easier to perform using, for example, a higher order axiom of replacement and/or a universe existence axiom, then they should be assumed. This is in the spirit of the 'working mathematician approach' mentioned above. I can see little merit in sticking to first order ZFC unless practical benefits result. On the other hand, consideration of parsimony (Occam's Razor) requires that one should not introduce exotic mechanisms for their own sake. I think that more research is needed to establish the best version of set theory for practical applications.

3 Defined notions

The usual set-theoretic notions can be developed in a standard way. The details are only sketched here.

3.1 Bounded quantifiers

The notation $\forall\, x \in X.\ t[x]$ abbreviates $\forall\, x.\ x \in X \Rightarrow t[x]$ and the notation $\exists\, x \in X.\ t[x]$ abbreviates $\exists\, x.\ x \in X \wedge t[x]$.

3.2 Finite sets

The empty set $\emptyset$, singletons $\{\mathtt{x}\}$ and binary unions $s_1 \cup s_2$ were introduced in the preceding section. The notation for finite sets is defined by:

$$\{\mathtt{x}_1,\ \mathtt{x}_2,\ \dots\ ,\ \mathtt{x}_n\} \ = \ \{\mathtt{x}_1\} \cup (\{\mathtt{x}_2\} \cup \dots (\{\mathtt{x}_n\} \cup \emptyset))$$

3.3 Ordered pairs and cartesian products

$$\langle \mathtt{x},\mathtt{y}\rangle = \{\{\mathtt{x}\},\ \{\mathtt{x},\mathtt{y}\}\}$$

$\langle \mathtt{x}_1,\mathtt{x}_2,\dots,\mathtt{x}_{n-1},\mathtt{x}_n\rangle$ abbreviates $\langle \mathtt{x}_1,\langle \mathtt{x}_2,\dots,\langle \mathtt{x}_{n-1},\mathtt{x}_n\rangle \dots\rangle\rangle$. The functions Fst and Snd extract the first and second components of pairs. The characteristic property of pairs is:

$$\forall \mathtt{x1\ x2\ y1\ y2}.\ (\langle \mathtt{x1},\mathtt{y1}\rangle = \langle \mathtt{x2},\mathtt{y2}\rangle) = (\mathtt{x1} = \mathtt{x2}) \wedge (\mathtt{y1} = \mathtt{y2})$$

The cartesian product $\times$ is defined by:

$$\mathtt{X} \times \mathtt{Y} \ = \ \{\langle \mathtt{x1},\mathtt{x2}\rangle \in \mathbb{P}(\mathbb{P}(\mathtt{X} \cup \mathtt{Y})) \mid \mathtt{x1} \in \mathtt{X} \wedge \mathtt{x2} \in \mathtt{Y}\}$$

In connection with relations and functions (see Section 3.4), the notation $\mathtt{x} \mapsto \mathtt{y}$ will sometimes be used for the pair $\langle \mathtt{x},\mathtt{y}\rangle$.

3.4 Relations and functions

A relation between `X` and `Y` is a subset of the product of `X` and `Y`. The set of all relations between `X` and `Y` is denoted by $\mathtt{X} \leftrightarrow \mathtt{Y}$ and defined by:

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

Functions are single-valued relations. The set of all (partial or total) functions from `X` to `Y` is denoted by $\mathtt{X} \nrightarrow \mathtt{Y}$ and defined by:

```
X ⇸ Y =
 {f ∈ X ↔ Y |
  ∀x y1 y2. x ↦ y1 ∈ f ∧ x ↦ y2 ∈ f ⇒ (y1 = y2)}
```

The set of total functions from `X` to `Y` is denoted by `X → Y` and defined by:

```
X → Y =
 {f ∈ X ⇸ Y | ∀x. x ∈ X ⇒ ∃y. y ∈ Y ∧ ⟨x,y⟩ ∈ f}
```

Functions inside set theory (i.e. members of `X → Y`) need to be distinguished from functions in the underlying logic. If disambiguation is needed, the former will be called *set functions* and the latter *logical functions*. Set functions are terms of type V, whereas logical functions have types of the form $\sigma_1 \to \sigma_2$.

The application of a set function `f` to argument `x` is denoted either by `f x`, just like the application of a logical function, or by `f ◇ x` if the set-theoretic meaning is to be emphasized. However, no ambiguity results from the former notation because set functions and logical functions can never have the same type. The application of a set function is defined by:

```
f ◇ x  =  εy. x ↦ y ∈ f
```

where ε is the Hilbert ε-operator (see Section 2.2).

3.5 Abstraction notation for set functions

The notation `Fn x ∈ X.` t[x] denotes the set function `GraphFn X (λx.` t[x]`)`, where `GraphFn` is defined by:

```
GraphFn X f = {x ↦ y ∈ X × Image f X | y = f x}
```

Thus `Fn x ∈ X.` t[x] is equivalent to:

```
{x ↦ y ∈ X × Image(λx.t[x])X | y = t[x]}
```

The set function version of β-conversion is:

```
y ∈ X ⊢ (Fn x ∈ X. t[x]) ◇ y = t[y]
```

which is implemented as a HOL conversion. Note that `Fn x ∈ X.` t[x] has type V, whereas `λx.` t[x] has type $V \to V$.

3.6 Truthvalues (Booleans)

Define `False = ∅`, `True = {∅}`, `Bool = {True,False}` and

```
bool2Bool b  =  (b => True | False)
```

It then follows that:

```
True ∈ Bool,   False ∈ Bool,

∀b. (bool2Bool b = True) = b,   ∀b. (bool2Bool b = False) = ¬b,

∀b1 b2. (bool2Bool b1 = bool2Bool b2) = (b1 = b2)
```

3.7 Natural numbers

The natural numbers are represented inside V by the set:

```
{∅, {∅}, {∅,{∅}}, {∅,{∅},{∅,{∅}}}, ... }
```

To formalise this, define the logical function num2Num (of type $num \rightarrow V$) by primitive recursion (Suc is the successor function):

```
num2Num 0 = ∅
num2Num(Suc n) = num2Num n ∪ {num2Num n}
```

The set Num of natural numbers is then defined by:

```
Num = {x ∈ InfiniteSet | ∃n. x = num2Num n}
```

It is straightforward to define a function Num2num such that:

```
(∀n. Num2num(num2Num n) = n) ∧
(∀x. x ∈ Num = (num2Num(Num2num x) = x))
```

3.8 Unions of countable sequences

A countable sequence of sets is a function `s` of type $num \rightarrow V$. The union of the sequence `s` is:

```
s 0 ∪ s 1 ∪ s 2 ∪ ···
```

This set is just the union ($\bigcup$) of the image of Num under `s ∘ Num2num` (the function obtained by composing Num2num with `s`) i.e. `UnionSeq s`, where:

$$\mathsf{UnionSeq}\ \mathtt{s}\ \mathtt{=}\ \bigcup(\mathsf{Image}(\mathtt{s}\ \circ\ \mathsf{Num2num})\mathsf{Num})$$

The notation $\bigcup_{\mathtt{n}} t[\mathtt{n}]$ abbreviates $\mathsf{UnionSeq}(\lambda\mathtt{n}.\,t[\mathtt{n}])$.

4 Lists: a set-theoretic construction in HOL

The type V can be used to perform set-theoretic constructions that would be hard or impossible in pure HOL. A non-trivial example, due to Sten Agerholm, is the inverse-limit construction of Scott's model of the λ-calculus D_∞ [1]. Only a trivial example will be presented here: the construction of lists. However, this does illustrate some of the extra possibilities that the type V provides.

4.1 Lists in HOL and ST

In the current HOL system, lists of elements of type σ are represented as a subtype of the type $(num \to \sigma)\times num$, the idea being that a pair (f, n) represents the list $[f\ \ 0; f\ \ 1;\ \ \ldots\ ; f(\mathtt{n}-1)]$.[5]

A more direct approach uses $\langle x_1, \langle x_2, \ldots, \langle x_n, \mathsf{True}\rangle \ldots\rangle\rangle$ to represent the list $[x_1;\ \ldots ; x_n]$ (the empty list is represented by True). However, this is not 'well-typed' since tuples with different lengths have different types This approach thus cannot be used to define lists in the HOL logic. However, the construction can easily be performed inside ST. Define by primitive recursion:

$$\begin{array}{l}(\mathsf{FiniteList}\ \mathtt{X}\ \ 0\ =\ \{\mathsf{True}\})\ \wedge \\ (\mathsf{FiniteList}\ \mathtt{X}\ \ (\mathsf{Suc}\ \mathtt{n})\ =\ \mathsf{FiniteList}\ \mathtt{X}\ \mathtt{n}\ \cup\ (\mathtt{X}\ \times\ \mathsf{FiniteList}\ \mathtt{X}\ \mathtt{n}))\end{array}$$

Thus FiniteList `X n` is the set of lists of members of `X` whose length is less than or equal to `n`. The set List `X` of all finite lists of members of `X` is thus defined by:

$$\mathsf{List}\ \mathtt{X}\ =\ \bigcup_{\mathtt{n}}\ \mathsf{FiniteList}\ \mathtt{X}\ \mathtt{n}$$

A routine proof by mathematical induction establishes:

$$\forall\mathtt{X}.\ \mathsf{List}\ \mathtt{X}\ \subseteq\ \{\mathsf{True}\}\ \cup\ (\mathtt{X}\ \times\ \mathsf{List}\ \mathtt{X})$$

and simple set-theoretic manipulations establish

---

[5] To ensure that the pairs (f_1, x_1) and (f_2, x_2) are equal if and only if the corresponding lists are equal, it is required that pairs (f, n) representing lists have the property that $f\ \ m$ equals some canonical value when m is greater than or equal to the length n of the list. The subtype consisting of such pairs (f, n) is used to define lists.

```
∀X. {True} ∪ (X × List X) ⊆ List X
```

hence:

```
∀X. List X = {True} ∪ (X × List X)
```

The following structural induction for lists also follows easily by mathematical induction.

```
∀P X.
 P True ∧ (∀l ∈ List X. P l ⇒ ∀x ∈ X. P⟨x,l⟩)
 ⇒
 ∀l ∈ List X. P l
```

4.2 Lifting ST lists into HOL: the monomorphic case

The function List can be used to define list types in higher order logic. This illustrates one way of using ST: as a source of representations for type definitions.

It is not clear how to use List to define lists of elements of arbitrary type (this is discussed in more detail later), but monomorphic lists of elements of a type isomorphic to a set are straightforward. As an illustrative example, lists of numbers will be defined here.

The first step is to define a new type *numlist* isomorphic to the subtype of V consisting of the members of the set List Num. This subtype is characterized by the predicate `λl. l ∈ List Num`. The result of this definition is the following definitional axiom that characterizes the type *numlist*.

```
∃rep. TypeDefinition(λl. l ∈ (List Num))rep
```

where TypeDefinition is the built-in HOL constant defined by:

```
TypeDefinition P rep =
 (∀x' x''. (rep x' = rep x'') ⇒ (x' = x'')) ∧
 (∀x. P x = (∃x'. x = rep x'))
```

From the definitional axiom characterizing *numlist* it is routine to define a bijection List2numlist from the set List Num to the type *numlist* and its inverse numlist2List such that:

```
(∀a. List2numlist(numlist2List a) = a) ∧
(∀r. r ∈ (List Num) = (numlist2List(List2numlist r) = r))
```

The various list processing operators can be defined by lifting the corresponding operations from V to *numlist*.

```
Nil = List2numlist True

Cons x l = List2numlist⟨num2Num x, numlist2List l⟩

Hd l = Num2num(Fst(numlist2List l))

Tl l = List2numlist(Snd(numlist2List l))
```

The required properties of these functions can then be proved by rewriting away their definitions, followed by straightforward set theory.

```
∀x l. Hd(Cons x l) = x

∀x l. Tl(Cons x l) = l

∀x1 x2 l1 l2.
  (Cons x1 l1 = Cons x2 l2) = (x1 = x2) ∧ (l1 = l2)

∀x l. ¬(Cons x l = Nil)

∀P. P Nil ∧ (∀l. P l ⇒ ∀x. P(Cons x l)) ⇒ ∀l. P l
```

I cannot see any way in pure HOL of lifting the function List into a HOL type operator (the set argument of List would have to correspond to a type variable in HOL). However, if transfer principles between higher order logic and ST are added then this can be done (see Section 6 below).

5 A pure embedding of HOL in ST

As has been already illustrated, it is possible to translate types and terms of the HOL logic into V. For example, the types *bool* and *num* can be translated to the sets Bool and Num, respectively, and then the functions bool2Bool and num2Num can be used to translate terms of type *bool* and *num*.

This translation can be systematised: a logical function `rep` $: \sigma \to V$ represents a HOL type σ by a set `s`$:V$ iff `rep` is one-to-one and maps σ onto `s`. The members of `s` are those values `x` of type V such that `x ∈ s`. Thus, `rep` represents σ by `s` iff:

```
(∀x' x''. (rep x' = rep x'') ⇒ (x' = x'')) ∧
(∀x. x ∈ s = (∃x'. x = rep x'))
```

HOL already contains a constant TypeDefinition defined by:

```
TypeDefinition P rep =
 (∀x' x''. (rep x' = rep x'') ⇒ (x' = x'')) ∧
 (∀x. P x = (∃x'. x = rep x'))
```

The condition that `rep`, of type $\sigma \to V$, represents σ by `s` is thus defined by `SetType s rep`, where:

```
SetType s rep = TypeDefinition (λx:V. x ∈ s) rep
```

If `rep` represents σ by `s`, then any term $t : \sigma$ is represented by `rep` t, which has the property that `rep` $t \in$ `s`.

In this systematised setting, the representation of the truthvalues and natural numbers is expressed by `SetType Bool bool2Bool` and `SetType Num num2Num`.

To represent product types, define:

```
rep1 ⋇ rep2 = λ(x1,x2). ⟨rep1 x1, rep2 x2⟩
```

If $\texttt{rep}_1$ represents σ_1 and $\texttt{rep}_2$ represents σ_2, then $\texttt{rep}_1 \divideontimes \texttt{rep}_2$ represents $\sigma_1 \times \sigma_2$. This follows from the easily proved theorem:

```
∀s1 s2 rep1 rep2.
  SetType s1 rep1 ∧ SetType s2 rep2 ⇒
  SetType(s1 × s2)(rep1 ⋇ rep2)
```

To represent function types, first observe that if `rep` represents σ by `s`, then `s` is determined by `rep`. Define:

```
RepSet rep = εs. ∀y. y ∈ s = ∃x. y = rep x
```

then it follows that:

```
∀s rep. SetType s rep ⇒ (RepSet rep = s)
```

Now define:

```
rep1 ↠ rep2 =
 λf.
  {x' ↦ y' ∈ RepSet rep1 × RepSet rep2 |
    ∃x y. (x' = rep1 x) ∧ (y' = rep2 y) ∧ (f x = y)}
```

If $\texttt{rep}_1$ represents σ_1 and $\texttt{rep}_2$ represents σ_2, then the set $\texttt{rep}_1 \twoheadrightarrow \texttt{rep}_2$ represents $\sigma_1 \to \sigma_2$. This is verified by the theorem:

```
∀s1 s2 rep1 rep2.
  SetType s1 rep1 ∧ SetType s2 rep2 ⇒
  SetType(s1 → s2)(rep1 ↠ rep2)
```

⋇ and ↠ enable any type constructed out of representable types using × and → to be represented. For example, the function

```
(num2Num ⋇ num2Num) ↠ bool2Bool
```

represents the type *num*×*num* → *bool* by the set Num × Num → Bool.

Types are defined in HOL by declaring them to be in one-to-one correspondence with non-empty subtypes of existing types. Thus if a type has already been represented as a set then any type (i.e. 0-ary type operator) defined by a subset of it will also be represented as a set. In HOL, type operators can also be defined, but I cannot see any uniform automatic way in pure HOL of generating representations of polymorphic type operators from their definitions (see Section 6 for an 'impure' approach). However, just as representations of the type operators → and × were constructed manually, so representations of the other built-in type operators could be manually defined. For example, the constant List could be used to represent HOL lists via a suitable representation function (details omitted).

In general, each n-ary operator *op* (e.g. ×) needs to be associated with an n-argument function, Op say (e.g. ×), and a representation function, OP say (e.g. ⋇), such that:

```
∀s1 ⋯ sn rep1 ⋯ repn.
  SetType s1 rep1 ∧ ⋯ ∧ SetType sn repn
  ⇒
  SetType(Op s1 ... sn)(OP rep1 ... repn)
```

There are ten type operators built-in to the HOL logic. Four of these have already been considered (arity shown in brackets): *bool* (0), *num* (0), → (2) and × (2). The remaining ones are: *ind* (0), *one* (0), + (2), *list* (1), *tree* (0) and *ltree* (1). It should be possible to represent these in ST (though *ind* will need to be represented axiomatically) and thus by applying appropriate representation functions, to convert any HOL theorem into an ST theorem.

Such an embedding in ST could be supported by syntactic conventions: if σ is a type then denote the corresponding set by $\downarrow\sigma$, and if t is a term of type σ then let $\downarrow t$ denote the corresponding set. Thus $\downarrow t : V$ and $\vdash\ \downarrow t \in \downarrow\sigma$. For example, with such a convention, the HOL theorem:

```
∀m n.
 (0 + m = m) ∧
 (m + 0 = m) ∧
 ((Suc m) + n = Suc(m + n)) ∧
 (m + (Suc n) = Suc(m + n))
```

becomes:

```
∀m n ∈ ↓num.
  ((↓+ ◇ ↓0) ◇ m = m) ∧
  ((↓+ ◇ m) ◇ ↓0 = m) ∧
  ((↓+ ◇ (↓Suc ◇ m)) ◇ n = ↓Suc ◇ ((↓+ ◇ m) ◇ n)) ∧
  ((↓+ ◇ m) ◇ (↓Suc ◇ n) = ↓Suc ◇ ((↓+ ◇ m) ◇ n))
```

where:

```
↓num = Num
↓0 = num2Num 0
↓Suc = (num2Num -↠ num2Num)Suc
↓+ = (num2Num -↠ (num2Num -↠ num2Num))+
```

6 Transfer principles between HOL and ST

To try to increase the synergy between HOL and ST, 'hard-wired' (and highly experimental) transfer principles have been implemented. The underlying intuition is to regard higher order logic as a typed layer on top of set theory, so that each HOL theorem denotes a corresponding set-theoretic fact. This set-theoretic semantics is essentially just the Pitts semantics given in *Introduction to HOL* [13, Chapter 15] and the transfer principles amount to a shallow embedding [3] corresponding to this semantics.

The transfer principles have a number of components:

(i) A principle `HOL2ST` for transferring theorems from pure HOL into ST.

(ii) A principle `ST2HOL` for transferring theorems from pure ST into HOL.

(iii) A principle `CTYPE` for obtaining a type membership statement in ST from the type of a constant in HOL.

(iv) A definitional principle for defining new HOL types and type operators in terms of sets.

(v) A definitional principle for 'lifting' an ST constant into HOL.

Before these can be described and illustrated the method of associating HOL types and terms with ST sets will be specified.

6.1 Associating HOL types with sets

HOL types will be associated with members of V and type operators with curried functions of type $V \to V \to \cdots \to V$.

More specifically, each n-ary type operator op is automatically associated with a constant $|op|$. If op is 0-ary (i.e. is a type constant), then $|op|$ has type V and intuitively denotes the set corresponding to op. If op is n-ary with $n > 0$, then $|op|$ is a curried function taking n arguments of type V and returning a result of type V, this function constructs a set corresponding to $(\sigma_1,\ldots,\sigma_n)op$ from the sets corresponding to the types $\sigma_1, \ldots, \sigma_n$.

For example, $|\mathrm{num}|$ is a constant of type V, $|\mathrm{list}|$ is a constant of type $V \to V$ and $|\times|$ is a constant of type $V \to (V \to V)$.

The set $|\mathsf{Bool}|$ will be axiomatically asserted equal to Bool and the functions $|\to|$ and $|\times|$ to the already discussed functions $\to$ and $\times$ (the identification of $|\times|$ with $\times$ is discussed further in Section 6.2.1). However, other HOL types will normally not correspond to their explicitly constructed ST counterparts. For example, the set $|\mathrm{num}|$ is not the same as Num (though, of course, it is isomorphic to it).

The set associated with a type σ will be denoted by $[\![\sigma]\!]$. This notation is chosen to suggest the semantic nature of the association. Type variables range over types and are thus associated with variables that range over sets. A type variable α will be associated with a variable of type V with the same name.

The set $[\![\sigma]\!]$ is defined inductively by:

$$\begin{array}{lcll}
[\![op_0]\!] & = & |op_0| & (op_0 \text{ has arity } 0) \\
[\![\alpha]\!] & = & \alpha & (\text{type variables}) \\
[\![(\sigma_1,\ldots,\sigma_n)op_n]\!] & = & |op_n|\ [\![\sigma_1]\!]\ \cdots\ [\![\sigma_n]\!] & (op_n \text{ has arity } n)
\end{array}$$

6.2 Associating HOL constants with sets

In HOL-ST, each HOL constant c is automatically associated with a constant $|\mathsf{c}|$. If c is monomorphic (has a type containing no type variables), then $|\mathsf{c}|$ will have type V. If the type of c contains n distinct type variables, then $|\mathsf{c}|$ will be a (curried) function taking n arguments of type V and returning a result of type V. The notation $\sigma[\alpha_1,\ldots,\alpha_n]$ indicates that type σ contains type variables $\alpha_1, \ldots, \alpha_n$. The instance of this type in which type variable α_i is instantiated to type σ_i (for $1 \leq i \leq n$) is denoted by $\sigma[\sigma_1,\ldots,\sigma_n]$. Thus, if the generic type of a constant c is $\sigma[\alpha_1,\ldots,\alpha_n]$, then all occurrences of c will have a type of the form $\sigma[\sigma_1,\ldots,\sigma_n]$.

For example, $|+|$ is a constant of type V which denotes the set function corresponding to addition (a member of `|num|`$\to$`(|num|`$\to$`|num|))` and $|\mathsf{I}|$, where the HOL constant I is the polymorphic identity function with type $\alpha \to \alpha$, is a constant of type $V \to V$ which denotes the function that maps any set `X` to the identity set function on `X`, i.e. λ`X. Fn x` $\in$ `X. x`.

6.2.1 Properties of associated sets

HOL's constants include the truthvalues T and F, the Boolean logical operators $\neg$, $\wedge$, $\vee$, $\Rightarrow$, the quantifiers $\forall$ and $\exists$, and equality $=$.

T, F, $\neg$, $\wedge$, $\vee$ and $\Rightarrow$ are monomorphic, so the type of the associated constants is V. The HOL types of $\forall$, $\exists$ and $=$ contain one type variable, so the type of the associated constants is $V{\rightarrow}V$. Thus:

$|\mathsf{T}| : V$
$|\mathsf{F}| : V$
$|\neg| : V$
$|\wedge| : V$
$|\vee| : V$
$|\Rightarrow| : V$
$|\forall| : V{\rightarrow}V$
$|\exists| : V{\rightarrow}V$
$|{=}| : V{\rightarrow}V$

These logical constants are related to their set-theoretic counterparts by the following laws:

$$|\mathsf{T}| = \mathsf{True}$$

$$|\mathsf{F}| = \mathsf{False}$$

$$\forall \mathtt{x}.\ \mathtt{x} \in \mathsf{Bool} \Rightarrow (|\neg| \diamond \mathtt{x} = \mathsf{bool2Bool}(\neg(\mathtt{x} = |\mathsf{T}|)))$$

$$\forall \mathtt{x}.\ \mathtt{x} \in \mathsf{Bool} \Rightarrow (\neg(\mathtt{x} = |\mathsf{T}|) = (|\neg| \diamond \mathtt{x} = |\mathsf{T}|))$$

$$\begin{array}{l} \forall \mathtt{x}\ \mathtt{y}.\ \mathtt{x} \in \mathsf{Bool} \wedge \mathtt{y} \in \mathsf{Bool} \Rightarrow \\ \qquad ((|\wedge| \diamond \mathtt{x}) \diamond \mathtt{y} = \mathsf{bool2Bool}((\mathtt{x} = |\mathsf{T}|) \wedge (\mathtt{y} = |\mathsf{T}|))) \end{array}$$

$$\begin{array}{l} \forall \mathtt{x}\ \mathtt{y}.\ \mathtt{x} \in \mathsf{Bool} \wedge \mathtt{y} \in \mathsf{Bool} \Rightarrow \\ \qquad ((\mathtt{x} = |\mathsf{T}|) \wedge (\mathtt{y} = |\mathsf{T}|) = ((|\wedge| \diamond \mathtt{x}) \diamond \mathtt{y} = |\mathsf{T}|)) \end{array}$$

$$\begin{array}{l} \forall \mathtt{x}\ \mathtt{y}.\ \mathtt{x} \in \mathsf{Bool} \wedge \mathtt{y} \in \mathsf{Bool} \Rightarrow \\ \qquad ((|\vee| \diamond \mathtt{x}) \diamond \mathtt{y} = \mathsf{bool2Bool}((\mathtt{x} = |\mathsf{T}|) \vee (\mathtt{y} = |\mathsf{T}|))) \end{array}$$

$$\begin{array}{l} \forall \mathtt{x}\ \mathtt{y}.\ \mathtt{x} \in \mathsf{Bool} \wedge \mathtt{y} \in \mathsf{Bool} \Rightarrow \\ \qquad ((\mathtt{x} = |\mathsf{T}|) \vee (\mathtt{y} = |\mathsf{T}|) = ((|\vee| \diamond \mathtt{x}) \diamond \mathtt{y} = |\mathsf{T}|)) \end{array}$$

$$\begin{array}{l} \forall \mathtt{x}\ \mathtt{y}.\ \mathtt{x} \in \mathsf{Bool} \wedge \mathtt{y} \in \mathsf{Bool} \Rightarrow \\ \qquad ((|\Rightarrow| \diamond \mathtt{x}) \diamond \mathtt{y} = \mathtt{bool2Bool}((\mathtt{x} = |\mathsf{T}|) \Rightarrow (\mathtt{y} = |\mathsf{T}|))) \end{array}$$

```
∀x y. x ∈ Bool ∧ y ∈ Bool ⇒
       ((x = |T|) ⇒ (y = |T|) = ((|⇒|◇x)◇y = |T|))

∀f X. f ∈ (X → Bool) ⇒
       ((|∀| X)◇f = bool2Bool(∀x. x ∈ X ⇒ (f◇x = |T|)))

∀f X. f ∈ (X → Bool) ⇒
       ((∀x. x ∈ X ⇒ (f◇x = |T|)) = ((|∀| X)◇f = |T|))

∀f X. f ∈ (X → Bool) ⇒
       ((|∃| X)◇f = bool2Bool(∃x. x ∈ X ∧ (f◇x = |T|)))

∀f X. f ∈ (X → Bool) ⇒
       ((∃x. x ∈ X ∧ (f◇x = |T|)) = ((|∃| X)◇f = |T|))

∀X x y. x ∈ X ∧ y ∈ X ⇒
         (((|=| X)◇x)◇y = bool2Bool(x = y))

∀X x y. x ∈ X ∧ y ∈ X ⇒
        ((x = y) = (((|=| X)◇x)◇y = |T|))
```

Some of these laws follow from `HOL2ST` and the definitions of the logical operators in HOL, others need to be postulated as axioms of HOL-ST.

There is a difficulty with identifying |×| and ×, because cartesian products of types are defined in HOL, and transferring this definition into ST might result in a clash with ST's 'native' notion of pairing (see Section 3.3). This (and related) points need further work. More generally, the exact set of axioms needed to support the transfer principles is still under consideration. The current prototype HOL-ST system rests on rather insecure foundations: it has as axioms a number of formulae that are in fact derivable and, worse, may well be inconsistent!

6.3 Associating HOL and ST terms

Each HOL term t is associated with a term $[\![t]\!]$ of type V as follows:

$$\begin{array}{lll}
[\![x:\sigma]\!] & = \ x : V & \text{(variables)} \\
[\![\mathsf{c}:\sigma[\sigma_1,\ldots,\sigma_n]\,]\!] & = \ |\mathsf{c}|\ [\![\sigma_1]\!]\ \cdots\ [\![\sigma_n]\!] & \text{(constants)} \\
[\![\lambda\, x:\sigma.\ t]\!] & = \ \mathtt{Fn}\ x\ \in\ [\![\sigma]\!].\ [\![t]\!] & \text{(abstractions)} \\
[\![t_1\ t_2]\!] & = \ [\![t_1]\!] \diamond [\![t_2]\!] & \text{(applications)}
\end{array}$$

6.4 The transfer principle `HOL2ST`

The intuitive idea of the transfer principle `HOL2ST` is that it is a new primitive inference rule that converts a conventional HOL theorem $\vdash t$ to the ST theorem $\vdash [\![t]\!] = |\mathsf{T}|$. However, this idea needs to be refined a bit.

First, consider the ST counterpart of $\vdash$ `m + 0 = m`. A first guess might be:

$$\vdash ((|{=}|\ |num|)\diamond((|{+}|\diamond \mathtt{m})\diamond|0|))\diamond \mathtt{m} = |\mathsf{T}|$$

Unfortunately, this is not necessarily true if the variable `m` is not restricted to be a member of the set $|num|$. The correct ST theorem is:

$$\mathtt{m} \in |num| \vdash ((|{=}|\ |\mathrm{num}|)\diamond((|{+}|\diamond \mathtt{m})\diamond|0|))\diamond \mathtt{m} = |\mathsf{T}|$$

Another problem arises with polymorphic theorems. Consider:

$$\vdash \exists \mathtt{x}{:}\alpha.\ \mathsf{T}$$

which is true because types in the HOL logic are non-empty. It would be inconsistent to derive from this a theorem:

$$(|\exists|\ \alpha)\diamond(\mathtt{Fn\ x} \in \alpha.\ |\mathsf{T}|) = |\mathsf{T}|$$

(which is equivalent to $\exists \mathtt{x} \in \alpha.\ \mathsf{T}$, i.e. $\exists \mathtt{x}.\ \mathtt{x} \in \alpha$), because α could be instantiated to the empty set. The result of applying `HOL2ST` must thus have a non-emptiness assumption:

$$\mathsf{Inhab}\ \alpha \vdash (|\exists|\ \alpha)\diamond(\mathtt{Fn\ x} \in \alpha.\ |\mathsf{T}|) = |\mathsf{T}|$$

where Inhab `X` ("`X` is inhabited") is defined by:

$$\vdash \forall \mathtt{X}.\ \mathsf{Inhab}\ \mathtt{X} = \exists \mathtt{x}.\ \mathtt{x} \in \mathtt{X}$$

Because the transfer principle `HOL2ST` may introduce membership and non-emptiness assumptions it will, for simplicity, be restricted to theorems without assumptions (of course, HOL theorems with assumptions can be transferred to ST by first fully discharging them).

To avoid foundational problems (e.g. inconsistency), `HOL2ST` is not applicable to terms that involve the type V or types that have been defined in terms of it.

6.5 Transferring HOL types to ST

All HOL constants have a type. The type transfer principle `CTYPE` generates a membership property in ST for the counterpart of each HOL constant. `CTYPE` is a conversion that takes a transferred constant (i.e. a constant of the form |c|) and returns its 'ST-type'. For example:

```
CTYPE "|0|"     =  ⊢ |0| ∈ |num|

CTYPE "|Suc|"   =  ⊢ |Suc| ∈ |num| → |num|

CTYPE "|+|"     =  ⊢ |+| ∈ |num| → (|num| → |num|)

CTYPE "|FST|"   = Inhab α, Inhab β ⊢ |FST| α β ∈ (α × β) → α
```

6.6 Simplifying transferred theorems

The transfer principle `HOL2ST` results in theorems that are completely encoded inside ST. In particular, the various logical operators are converted to their set-theoretic counterparts, e.g. the HOL theorem `⊢ ∀m n. m + n = n + m` becomes the ST theorem:

```
⊢ (|∀| |num|)◇
    (Fn m ∈ |num|.
      (|∀| |num|)◇
      (Fn n ∈ |num|.
        ((|=| |num|)◇((|+|◇m)◇n))◇((|+|◇n)◇m))) =
    |T|
```

It is usually much more convenient to have a theorem in which the logical operators are not encoded inside ST. The derived rule `ST_SIMP` deduces the following partially encoded theorem from the fully encoded one shown above:

```
⊢ ∀m n ∈ |num|. (|+|◇m)◇n = (|+|◇n)◇m
```

`ST_SIMP` uses various laws, including those listed in Section 6.2.1. For example:

```
⊢ ∀x y. x ∈ Bool ∧ y ∈ Bool ⇒
        ((|∧|◇x)◇y = bool2Bool((x = |T|) ∧ (y = |T|)))

⊢ ∀f X. f ∈ (X → Bool) ⇒
        ((|∀| X)◇f = bool2Bool(∀x. x ∈ X ⇒ (f◇x = |T|)))

⊢ ∀X x y. x ∈ X ∧ y ∈ X ⇒ (((|=| X)◇x)◇y = bool2Bool(x = y))
```

```
⊢ ∀b. (bool2Bool b = |T|) = b
```

In applying such laws it is often necessary to deduce set membership properties of the form $x \in X$. Membership properties for constants follow from the type transfer principle `CTYPE`, but for compound terms they need to be inferred using the derived laws:

```
⊢ f ∈ X → Y  ∧  x ∈ X  ⇒  f◇x ∈ Y

⊢ (∀x. x ∈ X ⇒ f x ∈ Y)  ⇒  GraphFn X f ∈ X → Y
```

The derived rule `TYPE` takes a list of terms of the form $x \in X$ and a term t, and generates a type membership property for t assuming the types proved in the list for any free variables.
For example:

```
TYPE ["x ∈ |num|";"y ∈ |num|"] "(|+| ◇ x) ◇ y"
```

proves the theorem:

```
x ∈ |num|, y ∈ |num| ⊢ (|+|◇x)◇y ∈ |num|
```

and

```
TYPE ["y ∈ |num|"] "Fn x ∈ |num|. (|+| ◇ x) ◇ y"
```

proves:

```
y ∈ |num| ⊢ (Fn x ∈ |num|. (|+|◇x)◇y) ∈ |num| → |num|
```

The simplifier `ST_SIMP` invokes the ‘typechecker’ `TYPE` when applying the laws listed in Section 6.2.1.

6.7 The transfer principle `ST2HOL`

The transfer principle `HOL2ST` infers theorems of ST from theorems of pure HOL. The transfer principle `ST2HOL` goes the other way. It is an inverse to `HOL2ST` in that if `HOL2ST` is applicable to a theorem *th*, then `ST2HOL(HOL2ST` *th*`)` evaluates to *th*. For example, `ST2HOL` applied to:

```
⊢ (|∀| |num|)◇
    (Fn m ∈ |num|. ((|=| |num|)◇((|+|◇m)◇|0|))◇m)  =
    |T|
```

yields $\vdash\ \forall\mathtt{m.\ m\ +\ 0\ =\ m}$ and applied to:

$$\begin{array}{l}
\mathsf{Inhab}\ \alpha\mathtt{,}\ \mathsf{Inhab}\ \beta \\
\vdash\ \mathtt{((}\mathsf{|{=}|}\mathtt{(}\alpha\ \rightarrow\ \mathtt{(}\beta\ \rightarrow\ \alpha\mathtt{)))}\diamond\mathtt{(}\mathsf{|K|}\ \alpha\ \beta\mathtt{))}\diamond \\
\quad \mathtt{(Fn\ x}\ \in\ \alpha\mathtt{.\ Fn\ y}\ \in\ \beta\mathtt{.\ x)\ =} \\
\quad \mathsf{|T|}
\end{array}$$

yields $\vdash\ \mathsf{K}\ \mathtt{=\ (}\lambda\mathtt{x\ y.\ x)}$. Notice how `ST2HOL` removes the non-emptiness assumptions $\mathsf{Inhab}\ \alpha$ and $\mathsf{Inhab}\ \beta$.

6.8 The principle of ST type definition

A new type definition principle called the *principle of* ST *type definition* allows HOL types to be defined to be equal to non-empty ST sets.

Suppose $t\colon V{\rightarrow}V{\rightarrow}\cdots{\rightarrow}V$ is a closed term such that:

$$\mathsf{Inhab}\ \mathtt{X1},\ \mathsf{Inhab}\ \mathtt{X2},\ \ldots\ \mathsf{Inhab}\ \mathtt{X}n\ \vdash\ \mathsf{Inhab(}t\ \mathtt{X1}\ \mathtt{X2}\ \ldots\ \mathtt{X}n\mathsf{)}$$

Then the principle of ST type definition allows the introduction of a new n-ary type operator, *op* say, with defining axiom:

$$\vdash\ \mathsf{|}op\mathsf{|}\ \mathtt{=}\ t$$

6.8.1 Example: polymorphic type of lists

Since $\vdash\ \mathsf{True}\ \in\ \mathsf{List}\ \mathtt{X}$, it follows *a fortiori*, that

$$\mathsf{Inhab}\ \mathtt{X}\ \vdash\ \mathsf{Inhab(List}\ \mathtt{X}\mathsf{)}$$

hence the principle of ST type definition can be used to define a new 1-ary HOL type operator *lst* with defining axiom $\vdash\ \mathsf{|}lst\mathsf{|}\ \mathtt{=}\ \mathsf{List}$.

6.9 The principle of ST constant definition

A new constant definition principle called the *principle of* ST *constant definition* allows HOL constants to be defined so that their ST counterparts are equal to a given ST term.

The principle requires an ST type membership theorem to be supplied, which has the form:

$$\vdash\forall\,\alpha_1\ \cdots\ \alpha_n\,.\ t\ \alpha_1\ \cdots\ \alpha_n\ \in\ [\![\sigma[\alpha_1,\ldots,\alpha_n]]\!]$$

where t is a closed term.

The result is the declaration of a constant, c say, with type $\sigma[\alpha_1,\ldots,\alpha_n]$ satisfying the definitional axiom:

$$\vdash |\mathsf{c}| = t$$

6.9.1 Example: list operators

If stnil and stcons are defined by:

$$\vdash \mathsf{stnil}\ (\mathtt{X}\!:\!V) = \mathsf{True}$$

$$\vdash \mathsf{stcons}\ \mathtt{X} = \mathtt{Fn}\ \mathtt{x} \in \mathtt{X}.\ \mathtt{Fn}\ \mathtt{l} \in \mathsf{List}\ \mathtt{X}.\ \langle \mathtt{x},\mathtt{l}\rangle$$

then if $\vdash |\mathit{lst}| = \mathsf{List}$, it follows that:

$$\vdash \forall\alpha.\ \mathsf{stnil}\ \alpha \in |\mathit{lst}|\ \alpha$$

$$\vdash \forall\alpha.\ \mathsf{stcons}\ \alpha \in \alpha \to (|\mathit{lst}|\ \alpha \to |\mathit{lst}|\ \alpha)$$

and so by the principle of ST constant definition, constants

$$\mathsf{nil} : \alpha\ \mathit{lst}, \qquad \mathsf{cons} : \alpha \to (\alpha\ \mathit{lst} \to \alpha\ \mathit{lst})$$

can be defined satisfying:

$$\vdash |\mathsf{nil}| = \mathsf{stnil}, \qquad \vdash |\mathsf{cons}| = \mathsf{stcons}$$

Recall the induction rule for ST lists:

$$\begin{array}{l}
\forall \mathtt{P}\ \mathtt{X}.\\
\quad \mathtt{P}\ \mathsf{True} \wedge (\forall \mathtt{l} \in \mathsf{List}\ \mathtt{X}.\ \mathtt{P}\ \mathtt{l} \Rightarrow \forall \mathtt{x} \in \mathtt{X}.\ \mathtt{P}\langle \mathtt{x},\mathtt{l}\rangle)\\
\quad \Rightarrow\\
\quad \forall \mathtt{l} \in \mathsf{List}\ \mathtt{X}.\ \mathtt{P}\ \mathtt{l}
\end{array}$$

This can be 'lifted' from ST into HOL to provide list induction for the type *lst*. The first step is to instantiate P to "$\lambda \mathtt{v}.\ \mathtt{P}\diamond \mathtt{v} = |\mathsf{T}|$" to get (after some simplification):

$$\begin{array}{l}
\vdash \forall \mathtt{P} \in (|\mathit{lst}|\ \mathtt{X}) \to |\mathit{bool}|.\\
\qquad (\mathtt{P}\diamond(|\mathsf{nil}|\ \mathtt{X}) = |\mathsf{T}|) \wedge\\
\qquad (\forall \mathtt{t} \in |\mathit{lst}|\ \mathtt{X}.\\
\qquad\quad (\mathtt{P}\diamond \mathtt{t} = |\mathsf{T}|) \Rightarrow (\forall \mathtt{h} \in \mathtt{X}.\ \mathtt{P}\diamond(((|\mathsf{cons}|\ \mathtt{X})\diamond \mathtt{h})\diamond \mathtt{t}) = |\mathsf{T}|))\\
\qquad \Rightarrow\\
\qquad (\forall \mathtt{l} \in |\mathit{lst}|\ \mathtt{X}.\ \mathtt{P}\diamond \mathtt{l} = |\mathsf{T}|)
\end{array}$$

Next, this has to be put entirely 'inside' set theory. This process is roughly inverse to what is done by `ST_SIMP` and is performed with a tool called `ST_CANON`. The result of applying `ST_CANON` to the last theorem is:

```
Inhab X
⊢ (|∀|((|lst| X)→|bool|))◇
   (Fn P ∈ (|lst| X)→|bool|.
     (|⇒|◇
      ((|∧|◇(P◇(|nil| X)))◇
       ((|∀|(|lst| X))◇
        (Fn t ∈ |lst| X.
          (|⇒|◇(P◇t))◇
          ((|∀| X)◇(Fn h ∈ X. P◇(((|cons| X)◇h)◇t)))))))◇
     ((|∀|(|lst| X))◇(Fn l ∈ |lst| X. P◇l))) =
   |T|
```

This is now in a form for lifting into HOL using `ST2HOL` to obtain:

```
⊢ ∀P. P nil ∧ (∀t. P t ⇒ (∀h. P(cons h t))) ⇒ (∀l. P l)
```

The details just shown are quite messy, but it should be possible to automate most of them, so that the process of lifting an ST set function to a HOL type operator is largely automatic.

6.10 A group theory example

Assume that • (an infixed binary operation), ∼ (a unary operation) and $\imath$ (an element) satisfy the following group axioms:

```
x • (y • z) = (x • y) • z
x • ı = x
x • ∼x = ı
```

It follows from these three properties that `ı • x = x`. The standard proof of this proceeds by first establishing the lemma `∼x • x = ı`:

```
∼x • x = ∼x • (x • ı)
       = (∼x • x) • ı
       = (∼x • x) • (∼x • ∼(∼x))
       = ((∼x • x) • ∼x) • ∼(∼x)
       = (∼x • (x • ∼x)) • ∼(∼x)
       = (∼x • ı) • ∼(∼x)
       = ∼x • ∼(∼ x)
       = ı
```

and then:

```
ı • x = (x • ∼x) • x
      = x • (∼x • x)
      = x • ı
      = x
```

These kinds of lemmas can be proved fully-automatically by some theorem provers (e.g. Otter [24]), but in HOL the proofs must be performed manually.[6] The complexity of the proof depends on how the group axioms are formalised. Formalisations that make the proof simple can make the resulting theorems hard to apply. However, by transferring theorems in and out of set theory, one can get the best of both worlds: simple abstract proofs that result in applicable theorems. These remarks are illustrated in the rest of this section and also in the next one.

6.10.1 The group example in HOL

A simple formulation of a group in HOL is to define the triple ⟨•,∼,ı⟩ to be a group iff ⊢ Group • ∼ ı, where:

```
⊢ Group(•:α→α→α) ∼ ı =
   (∀x y z. x • (y • z) = (x • y) • z) ∧
   (∀x. x • ı = x) ∧
   (∀x. x • ∼x = ı)
```

It is then very easy to prove:

```
⊢ Group • ∼ ı ⇒ (∼x • x = ı)

⊢ Group • ∼ ı ⇒ (ı • x = x)
```

6.10.2 The group example in ST

Unfortunately, as first pointed out by Elsa Gunter [14], this definition of a group in HOL does not support the easy formulation of theorems about subgroups. If HOL had subtypes (as, for example, PVS does [30]) this might not be a problem, but HOL doesn't.

Gunter's solution is to make the domain of the group not the whole argument type of the operators, but a subset of it, defined by a predicate `G` in the following definition.

---

[6] A package written by Konrad Slind that provides Knuth Bendix completion as a HOL derived inference rule can prove such lemmas automatically.

```
Group (G:α→bool) • ∼ ı  =
 (∀x y. G x ∧ G y ⇒ G(x • y)) ∧
 (∀x. G x ⇒ G(∼x)) ∧
 G ı ∧
 (∀x y z. G x ∧ G y ∧ G z ⇒ (x • (y • z) = (x • y) • z)) ∧
 (∀x. G x ⇒ (x • ı = x)) ∧
 (∀x. G x ⇒ (x • ∼x = ı))
```

With this formalisation, properties that were previously encoded as type information become implications. For example, the property that `G` is closed under • and ∼ corresponds to the conjuncts `∀x y. G x ∧ G y ⇒ G(x • y)` and `∀x. G x ⇒ G(∼x)`, respectively.

Rather than pursue this domains-as-predicates approach here, a similar set-theoretic version will be examined that is closer to the textbook definition of a group. For clarity, `∼◇x` will be written as `∼x` and `(• ◇ x)◇ y` as `x • y`. This notation is supported in the current prototype HOL-ST system (∼ has to be declared as a 'set function' and • as a 'curried set infix').

```
ST_Group G • ∼ ı =
 (• ∈ G → G → G) ∧
 (∼ ∈ G → G) ∧
 (ı ∈ G) ∧
 (∀x y z ∈ G. x • (y • z) = (x • y) • z) ∧
 (∀x ∈ G. x • ı = x) ∧
 (∀x ∈ G. x • ∼x = ı)
```

With this formulation, the carrier of the group is a set in ST, with the previous formulation it is a subtype (specified by a predicate) of a type. With either of these formulations the two lemmas are significantly more messy to prove. For example, with the ST formulation the lemmas are:

```
⊢ ST_Group G • ∼ ı ⇒ ∀x ∈ G. ∼x • x = ı

⊢ ST_Group G • ∼ ı ⇒ ∀x ∈ G. ı • x = x
```

To establish these, it is necessary to use properties like:

```
⊢ x ∈ G ∧ • ∈ G→(G→G) ∧ ∼ ∈ G→G ⇒ ∼x • (x • ∼x) ∈ G
```

In the formulation in Section 6.10.1 this sort of thing is handled automatically by typechecking. Using the transfer principle `HOL2ST`, this simple HOL proof can be used to establish the ST version.

The first step is to take the HOL versions of the lemmas:

```
⊢ Group • ∼ ı ⇒ (∼x • x = ı)

⊢ Group • ∼ ı ⇒ (ı • x = x)
```

and to transfer them into ST using `HOL2ST`. The results (after a bit of simplification with `SIMP_ST`) are:

```
⊢ • ∈ (G → (G → G)) ⇒
   ∼ ∈ (G → G) ⇒
   ı ∈ G ⇒
   x ∈ G ⇒
   ((((|Group| G) ◇ •) ◇ ∼) ◇ ı = |T|) ⇒
   (∼x • x = ı)

⊢ • ∈ (G → (G → G)) ⇒
   ∼ ∈ (G → G) ⇒
   ı ∈ G ⇒
   x ∈ G ⇒
   ((((|Group| G) ◇ •) ◇ ∼) ◇ ı = |T|) ⇒
   (ı • x = x)
```

The second step is to relate `ST_Group` to `|Group|`. This is accomplished with the theorem:

```
⊢ ST_Group G • ∼ ı =
   • ∈ (G → (G → G)) ∧
   ∼ ∈ (G → G) ∧
   ı ∈ G ∧
   ((((|Group| G) ◇ •) ◇ ∼) ◇ ı = |T|)
```

From this, and the ST versions of the two lemmas, it easily follows that:

```
⊢ ST_Group G • ∼ ı ⇒ (∀x ∈ G. ∼x • x = ı)

⊢ ST_Group G • ∼ ı ⇒ (∀x ∈ G. ı • x = x)
```

The actual proofs in the current version of HOL-ST require quite a lot of low-level manual fiddling, the details of which have been skipped in this account. However, it is hoped that eventually such detail can be automated.

7 Creating ST theories from HOL theories

The transfer principle `HOL2ST` moves individual theorems from HOL to ST. An alternative approach moves whole theories. This will be motivated via the group theory example.

7.1 The group example revisited

A particularly simple encoding of groups in HOL is as follows:

1. Start a new theory, `G` say.
2. Declare a new type, G say.
3. Declare constants $\bullet\ :\ G{\to}(G{\to}G)$, $\sim\ :\ G{\to}G$ and $\imath\ :\ G$.
4. Assert as axioms:

```
⊢ x • (y • z) = (x • y) • z
⊢ x • ı = x
⊢ x • ∼x = ı
```

The informal proof of the lemmas `∼x • x = ı` and `ı • x = x` given in Section 6.10 can then easily be performed line-by-line in HOL (details omitted). The result is the theory:

```
The Theory AbsGroup
Parents --  ST
Types --  G
Constants --  • : G → (G → G)      ∼ : G → G      ı : G
Infixes --  • : G → (G → G)
Axioms --
  Assoc  ⊢ ∀x y z. x • (y • z) = (x • y) • z
  Inv  ⊢ ∀x. x • ∼x = ı
  Id  ⊢ ∀x. x • ı = x
Theorems --
  Lemma1  ⊢  ∼x • x = ı      Lemma2  ⊢  ı • x = x
```

Unfortunately, this theory is useless because there is no way that it can be applied. One might, for example, want to show that certain operations on some particular type satisfy the axioms and hence conclude the two lemmas for those operations, but without an IMPS-style theory interpretation mechanism [10] (which HOL lacks) this is impossible.

However, it is intuitively clear that this theory justifies the following purely definitional theory in ST.

```
The Theory AbsGroupST
Parents --  ST
Definitions --
  AbsGroupAxioms
   ⊢ AbsGroupAxioms G • ∼ ı =
       (• ∈ G → G → G) ∧
       (∼ ∈ G → G) ∧
       (ı ∈ G) ∧
       Assoc G • ∧
       Inv G • ∼ ı ∧
       Id G • ı
  Assoc  ⊢ Assoc G • = ∀x y z ∈ G. x • (y • z) = (x • y) • z
  Inv  ⊢ Inv G • ∼ ı = ∀x ∈ G. x • ∼x = ı
  Id  ⊢ Id G • ı = ∀x ∈ G. x • ı = x
Theorems --
  Lemma1
    AbsGroupAxioms G • ∼ ı  ⊢  ∀x ∈ G. ∼x • x = ı
  Lemma2
    AbsGroupAxioms G • ∼ ı  ⊢  ∀x ∈ G. ı • x = x
```

In this theory, HOL types and constants have become ST variables and axioms have become assumptions. Such an ST theory can be applied using methods similar to those described in Section 6.10.2.

The current HOL-ST prototype only has a primitive HOL-to-ST theory translator. In particular, the translated theory is entirely inside ST, and requires tools like `ST_SIMP` to generate the theory shown above. Further experiments are needed to evaluate the utility of this approach.

8 Conclusions

This paper explores various possibilities for improving the power of HOL by adding a ZF-style set-theory (called ST). I view it as a contribution to the HOL2000 initiative [17].

The first step in merging HOL with set theory was to axiomatically specify the type V and the ZF-like axioms of ST. Just with this, useful new things become possible such as Agerholm's construction of D_∞. In Section 4 it was shown how the essentially set-theoretic construction:

$$\{\mathsf{True}\} \cup (\mathtt{X} \times \{\mathsf{True}\}) \cup (\mathtt{X} \times \mathtt{X} \times \{\mathsf{True}\}) \cup \cdots$$

could be used to define lists in set theory which could then be lifted into a HOL type of lists for a given type of elements.

This approach is inadequate for defining polymorphic lists (i.e. type operators), so in Section 6 a number of transfer principles between HOL and ST were explored. These provide a powerful mechanism for moving between the typed world of HOL and the untyped world of ST, but it requires new rules of inference that destroy the simplicity of the pure HOL system.

Finally, the translation of axiomatic HOL theories into definitional ST theories was envisioned. This provides some of the power of both 'abstract theories' and of IMPS-style theory interpretations.

The transfer principles and theory translation mechanisms, though related, meet different needs and neither subsumes the other. Further experiments are needed to establish whether either or both are really useful in practice, and are worth the major loss of logical simplicity they entail. My own opinion is that in the short term just the type V will be useful, without any transfer or translation principles (but possibly with the techniques of Section 5). In the longer term, the transfer principles and theory translation mechanisms might turn out to be worthwhile, but considerable work is needed to re-engineer and reconceptualise the current prototype HOL-ST system.

Acknowledgements

I have learnt much from talking to Larry Paulson and reading his pioneering papers on ZF in Isabelle. Francisco Corella, whose PhD thesis [7] contains many insights into mechanizing set theory, has had a significant influence on my thinking. Sten Agerholm's work on constructing Scott's D_∞ has provided valuable feedback on set theory in HOL. Finally, Thomas Forster, Tom Melham, Andy Pitts, John Harrison and Ken Kunen (by email) have helped me with various foundational questions and Richard Boulton and Jim Grundy supplied me with lists of errors in the first version of the paper. The research described here is partly supported by EPSRC grant `GR/G23654`.

References

[1] S. Agerholm. Formalising a model of the λ-calculus in HOL-ST. Technical Report 354, University of Cambridge Computer Laboratory, 1994.

[2] F. Borceux, editor. *Handbook of categorical algebra: Basic category theory.* Cambridge University Press, 1994.

[3] R. J. Boulton, A. D. Gordon, M. J. C. Gordon, J. R. Harrison, J. M. J. Herbert, and J. Van Tassel. 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: Proceedings of the IFIP TC10/WG 10.2 International Conference*, IFIP Transactions A-10, pages 129–156. North-Holland, June 1992.

[4] D. Cantone, A. Ferro, and E. Omodeo, editors. *Computable Set Theory*, volume 1. Clarendon Press, Oxford, 1989.

[5] P. M. Cohn. *Universal Algebra (Revised Edition)*, volume 6 of *Mathematics and Its Applications*. D. Reidel Publishing Company, 1981.

[6] R. L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, 1986.

[7] F. Corella. Mechanizing set theory. Technical Report 232, University of Cambridge Computer Laboratory, 1991.

[8] 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.

[9] 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).

[10] W. M. Farmer. Theory interpretation in simple type theory. In J. Heering, K. Meinke, B. Möller, and T. Nipkow, editors, *Higher-Order Algebra, Logic, and Term Rewriting*, volume 816 of *Lecture Notes in Computer Science*, pages 96–123. Springer-Verlag, September 1993. First International Workshop, HOA '93, Amsterdam, The Netherlands.

[11] 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.

[12] S. Finn and M. P. Fourman. *L2 – The LAMBDA Logic*. Abstract Hardware Limited, September 1993. In LAMBDA 4.3 Reference Manuals.

[13] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: A Theorem-proving Environment for Higher-Order Logic*. Cambridge University Press, 1993.

[14] E. L. Gunter. Doing algebra in simple type theory. Technical Report MS-CIS-89-38, Dept. of Computer and Information Science, Moore School of Engineering, University of Pennsylvania, June 1989. Distributed with the HOL system in `Training/studies/int_mod/doing_alg_paper`.

[15] E. L. Gunter. The implementation and use of abstract theories in HOL. In G. Winskel, editor, *Proceedings of the Third HOL Users Meeting, Aarhus University, 1-2 October 1990*, pages 47–64. Computer Science Department, Aarhus University, December 1990. Technical report DAIMI PB – 340, ISSN 0105-8517.

[16] 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.

[17] HOL2000 is a community wide initiative to build a successor to HOL on the shoulders of the existing system and other related ones. See the WWW URL: `http://lal.cs.byu.edu/lal/hol-documentation.html` for more details.

[18] P. T. Johnstone. *Notes on logic and set theory*. Cambridge University Press, 1987.

[19] C. B. Jones. *Systematic Software Development using VDM.* Prentice Hall International, 1990.

[20] L. Lamport and S. Merz. Specifying and verifying fault-tolerant systems. In *Proceedings of FTRTFT'94*, Lecture Notes in Computer Science. Springer-Verlag, 1994.

[21] 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.

[22] L. Magnusson and B. Nordström. The ALF proof editor and its proof engine. In *Types for Proofs and Programs: International Workshop TYPES '93*, pages 213–237. Springer, published 1994. LNCS 806.

[23] D. A. McAllester. *ONTIC: A Knowledge Representation System for Mathematics.* MIT Press, 1989.

[24] W. McCune. OTTER 2.0 users guide. Technical Report ANL-90/9, Argonne National Laboratory, 1990.

[25] C. McLarty. *Elementary categories, elementary toposes.* Number 21 in Oxford logic guides. Clarendon Press, Oxford, 1992.

[26] 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.

[27] L. C. Paulson. Set theory for verification: I. From foundations to functions. *Journal of Automated Reasoning*, 11(3):353–389, 1993.

[28] L. C. Paulson. Set theory for verification: II. Induction and Recursion. Technical Report 312, University of Cambridge Computer Laboratory, 1993.

[29] L. C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.

[30] PVS World Wide Web page. http://www.csl.sri.com/pvs/overview.html.

[31] 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.

[32] M. Saaltink. Z and EVES. Technical Report TR-91-5449-02, Odyyssey Research Associates, 265 Carling Avenue, Suite 506, Ottawa, Ontario K1S 2E1, Canada, October 1991.

[33] J. M. Spivey. *The Z Notation: A Reference Manual.* Prentice Hall International Series in Computer Science, 2nd edition, 1992.

[34] Phillip J. Windley. Abstract theories in HOL. In Luc Claesen and Michael J. C. Gordon, editors, *Proceedings of the 1992 International Workshop on Higher Order Logic Theorem Proving and its Applications.* North–Holland, November 1992.