Technical Report

UCAM-CL-TR-369
ISSN 1476-2986

Number 369

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

A comparison of HOL-ST and Isabelle/ZF

Sten Agerholm

July 1995

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

ISSN 1476-2986

A Comparison of HOL-ST and Isabelle/ZF

Sten Agerholm
University of Cambridge Computer Laboratory
New Museums Site, Cambridge CB2 3QG, UK

Abstract

The use of higher order logic (simple type theory) is often limited by its restrictive type system. Set theory allows many constructions on sets that are not possible on types in higher order logic. This paper presents a comparison of two theorem provers supporting set theory, namely HOL-ST and Isabelle/ZF, based on a formalization of the inverse limit construction of domain theory; this construction cannot be formalized in higher order logic directly. We argue that whilst the combination of higher order logic and set theory in HOL-ST has advantages over the first order set theory in Isabelle/ZF, the proof infrastructure of Isabelle/ZF has better support for set theory proofs than HOL-ST. Proofs in Isabelle/ZF are both considerably shorter and easier to write.

Contents

1 Introduction

Though higher order logic (simple type theory) is a useful framework for doing mathematics, there are situations where it is too weak, due to its simple type system. One can then use a stronger type theory, or observe that many constructions that are not possible on types in higher order logic would be possible on sets in set theory (which is completely untyped). Set theory might therefore provide a simple alternative to the increasing interest in applying stronger type theories in theorem proving.

Paulson has done a lot of pioneering work on mechanizing set theory. He has developed a very large amount of set theory in his Isabelle/ZF system [7, 8], which is an extension of a first order logic instantiation of the generic theorem prover Isabelle [9] with axioms of Zermelo-Fraenkel (ZF) set theory. Gordon has also been experimenting with mechanizing set theory in an attempt to combine the usefulness of higher order logic with the expressive power of set theory in a single system [4]. A prototype system, called HOL-ST, has been implemented by extending the existing HOL system [3] with axioms of ZF set theory (this is not a conservative extension).

A larger case study on HOL-ST was presented in [1]. By formalizing the inverse limit construction of domain theory, which would not be possible in HOL directly [2], the case study demonstrated how one can make essential use of the additional expressive power of set theory. The inverse limit construction is a method to give solutions to recursive domain equations that may involve non-trivial constructions such as the (continuous) function space. In [1], it was used to obtain a non-trivial model D_∞ of the untyped λ-calculus, i.e. D_∞ was proved to be isomorphic to its own (continuous) function space:

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

This paper presents a comparison of HOL-ST and Isabelle/ZF based on a formalization of the inverse limit construction in both systems. We concentrate on their different support for the formalization, i.e. for definitions, theorems and proofs, but do not give a detailed presentation of the formalization (see [1]). The version of the inverse limit construction employed here is based on categorical methods using embedding project pairs, see e.g. [6, 11, 10].

Comparing systems is difficult. The lack of some feature supported by one system does not mean that it could not be supported by another. In this paper, we have chosen to freeze time in the sense that the systems are compared in their state when this project started (Autumn 1994). The size and scope of the paper could easily grow out of hand if we had to argue about the alternatives for implementing all differences (and better proof support in general).

The rest of the paper is organized as follows. In Section 2, we introduce Isabelle and its first order logic instantiation FOL briefly. Then the set theories of HOL-ST and Isabelle/ZF are introduced in Section 3. The comparison starts in Section 4 where we consider the definitions of the formalizations. It continues in Section 5 which discusses how the systems support the proofs of theorems of the formalization. Section 6 summarizes the conclusions and can be read immediately after this introduction.

NB! This paper is written for someone who is familiar with HOL and may know little or nothing about Isabelle.

2 Isabelle

Isabelle [9] is a generic theorem prover. It provides support for mechanizing logics, called object logics, in its own logic, called the meta logic. Isabelle comes with various object logics, including Zermelo-Fraenkel (ZF) set theory. Isabelle/ZF [7, 8] was developed as an extension of another object logic, namely a many-sorted first order logic, called FOL. Below we first very briefly introduce the Isabelle meta logic and proof infrastructure, along with a few comments on the object logic FOL. The set theories of Isabelle/ZF and HOL-ST are presented in Section 3. A gentle introduction to Isabelle is provided in [5] and for a full introduction (and manual) see [9], which also contains references to a good number of papers about Isabelle.

2.1 Meta Logic

The Isabelle meta logic is a polymorphic (intuitionistic) higher order logic (simply typed λ-calculus) with type classes to control polymorphism. It is not meant for theorem proving in itself, but for providing a logical framework in which to state other logics, the so-called object logics. As usual, a term can be a constant, a variable, an abstraction, or an application. A term must be well-typed in the sense that we know from HOL. We shall often use the function type, which is written as "α `=>` β". All functions must be curried and function application is written $f(t)$, not $f\ t$ as in HOL; curried application is written $f(t_1, \ldots, t_n)$. The notation "`[`α_1`,`α_2`,...,`α_n`] =>` β" is a short-hand for the function type "α_1 `=> (`α_2 `=>...=> (`α_n `=>` β`)...)`".

The meta logic only provides the following three connectives:

- Implication: "`a ==> b`".
- Universal quantification: "`!!x. P(x)`".
- Equality: "`a == b`".

Implication is useful to represent object logic inference rules like the conjunction introduction rule, which is often written using the following graphical (natural deduction style) notation:

$$\frac{P \quad Q}{P \wedge Q}.$$

This rule is represented as "`P ==> (Q ==> P & Q)`", usually written using the short-hand "`[| P; Q |] ==> P & Q`". Note that `&` is object logic conjunction (the meta logic does not have one). The universal quantifier is useful for expressing generality in rules and axiom schemes. For instance, the forall introduction rule of first order logic

$$\frac{P}{\forall x.\ P},$$

which has the side condition that x is not free in the assumptions, can be formalized by "`(!!x. P(x)) ==> (ALL x. P(x))`". Note that there are two different universal quantifiers here, the meta level `!!` and an object level `ALL`. Finally, equality is useful for expressing definitional axioms, usually just called definitions; at present, there is no freeness check as in HOL to ensure that definitions are sound.

2.2 Object Logic (Isabelle/FOL)

Logical connectives of object logics are introduced as new constants of the meta logic. For instance, the FOL connectives for implication, universal quantification and equality are `-->` of type `"[o,o] => o"`, `ALL` of type `"('a => o) => o"` and `=` of type `"['a,'a] => o"`, where `'a` is a type variable of the meta logic and `o` is the meta logic type of FOL formulas. FOL also provides conjunction (`&`), disjunction (`|`), bi-implication (`<->`) and existential quantification (`EX`). It is important that one does not confuse the connectives of the meta logic with those of the object logics. In HOL, we do not have a similar distinction between meta and object logic.

Object logic extensions of the meta logic are organized hierarchically in theories, which consist of a signature and a list of axioms. The signature contains various type and parsing/pretty-printing information; in particular, it specifies constants and their type, types, and type classes (which are not considered in this paper).

2.3 Theorem Proving

The theorem proving infrastructure of Isabelle is mainly provided by the principle of resolution, which is based on (higher-order) unification. This elegantly supports a very small number of tools for forward and backward proof. By forward resolution, a theorem can be used like a HOL inference rule, whilst backward resolution turns the theorem into a tactic. Roughly speaking, forward proof by resolution is unifying the conclusion of one theorem with some assumption of another; the assumption is then replaced with the assumptions of the first theorem. Backward proof by resolution is unifying the conclusion of a theorem with the conclusion of a goal; the assumptions of the theorem become new subgoals. The subgoal module supports interactive tactic-based proofs.

In addition, Isabelle provides a simplification package, which implements tactics for contextual conditional rewriting, and the classical reasoning package, which implements tactics based on some simple proof procedures (see [9] for more information). These packages were only used a little in this work.

Finally, Isabelle implements a notion of schematic variables, or unknown variables as they are also called, which are written with a question mark in front, e.g. `?P`. Free variables of axioms, definitions, and theorems are translated automatically to schematic variables. Existentially quantified variables of object logics are also represented by unknowns via axioms. The purpose of schematic variables is to support and ease quantifier reasoning. Schematic variables can be instantiated, possibly in stages, by employing resolution in proofs. Hence, it is only rarely necessary to provide witnesses for existentials explicitly as in HOL; these are constructed behind the scenes. Similarly, explicit instantiations of universally quantified variables are rarely necessary.

3 Set Theory

HOL-ST and Isabelle/ZF provide two slightly different ZF-like set theories. Isabelle/ZF is an axiomatic extension of the object logic FOL, which, as mentioned above, provides a first order logic. HOL-ST is an axiomatic extension of HOL's higher order logic. This means that HOL's set theory is slightly more powerful than Isabelle's (see [4]), though we shall not exploit this in an essential way in this paper. Apart from this difference in logic, the two axiomatizations of set theory are essentially the same. We will not consider the axioms of the set theories in this paper since they are not important; they are easy to look up in [4] and [9]. Instead we will focus on syntactic issues, to introduce set theory syntax used later.

3.1 Representation and Notation

HOL is extended with set theory by declaring a new type `V` and a new constant '`::`' (set membership) of type `":V#V->bool"`,[1] and then postulating eight new axioms about `V` and '`::`'. Similarly, Isabelle/FOL is extended by declaring a new type `i` and a new constant '`:`' (set membership) of type `"[i,i] => o"`, and then postulating the eight axioms of set theory. In Isabelle/ZF, the type `i` (for individuals) instantiates the many-sorted first order logic. Hence, we can quantify over sets as in `ALL x. x:X --> f(x):Y` (elements of sets are themselves sets), which could equivalently be written as `ALL x:X. f(x):Y`, and compare sets by the equality `=`. The subset relation $\subseteq$ is written `Subset` in HOL-ST and `<=` in Isabelle/ZF.

Both systems provide a notation for set abstraction $\{x \in X \mid P[x]\}$. In Isabelle/ZF this is written as `{x:X . P(x)}` and in HOL-ST as `{x::X | P x}`.

3.2 Pairs and Numbers

Isabelle/FOL does not provide pairs or natural numbers but these are provided in set theory. Pairs are written `<x,y>`, which is a set and therefore has type `i`, and the usual destructors `fst` and `snd` are provided (both have the type `"i => i"`). The binary product `X * Y` consists of all pairs whose first component is in `X` and whose second component is in `Y`. In HOL-ST, we use pairs of higher order logic, but pairs are also available in set theory.

In Isabelle/ZF, the set `nat` provides natural numbers, so `0:nat` and `succ(n):nat`, for any `n:nat`. In HOL-ST, natural numbers are available as both the type `":num"` and the set `Num`. The type and the set are isomorphic with translation functions called `num2Num` and `Num2num`. The HOL constants `0` and `SUC` are used with the translation functions to build members of `Num`.

3.3 Functions and Dependent Products

We distinguish between set and logical functions. Set functions are elements of the function set, written `X->Y` in both systems. A set function is represented by a set of pairs, which must satisfy the obvious conditions to specify a function (definedness and uniqueness). In HOL-ST, logical functions are functions of higher order logic. In Isabelle/ZF, logical functions are functions of the meta logic (the object logic FOL does not provide functions). Set function application is written `f ^^ x` in HOL-ST and `` f ` x `` in Isabelle/ZF. If `f` is in `X->Y` and `x` in `X` then we can conclude that `f` applied to `x` is in `Y`, otherwise not.

[1]In the type of the set membership operator, note that elements of sets are themselves sets. Generally speaking, new sets must be constructed from existing sets some way, in principle starting from the empty set and then using axioms.

Set functions may be written using a (dependent) lambda abstraction. The syntax is `Fn x::X. b[x]` in HOL-ST and `lam x:X. b[x]` in Isabelle/ZF. The lambda abstraction consists of all set theory pairs of the form `<x,b[x]>`, where `x` is in `X`. Set function identity is written as `id` in Isabelle/ZF and as `Id` in HOL-ST. Set function composition is written as `O` in both systems.

Finally, both systems also provide a dependent product construction. The syntax is `PI x::X. Y[x]` in HOL-ST (see [1]) and `PROD x:X. Y[x]` in Isabelle/ZF. The elements of a dependent product are sets of pairs, corresponding to set functions that map elements `x` of the first set `X` to elements of the second (dependent) set `Y[x]`.

4 Definitions

In this section, we study the definitions of two formalizations of the inverse limit construction in HOL-ST and Isabelle/ZF. We try to compare the definitions and discuss the issue of which notions to represent in the meta logic of Isabelle/ZF and which to represent in first order logic and set theory. This is related to the question of whether to work in higher order logic or set theory in HOL-ST (discussed in [1]).

4.1 Basic Concepts

The differences between HOL-ST and Isabelle/ZF appear already in the first definitions of basic concepts of domain theory. Recall that domain theory is the study of complete partial orders (cpos) and continuous functions. A partial order is usually thought of as a set with an associated ordering relation which is reflexive, transitive and antisymmetric on all elements of the set. A complete partial order is a partial order in which all non-decreasing chains (sequences) of elements of the partial order have a least upper bound. Continuous functions are monotonic functions that preserve such least upper bounds.

In HOL-ST, we can represent the notion of partial order as a pair consisting of a set and a relation. The predicate po for partial orders can then be defined by

```
|- !D.
     po D =
     (!x :: set D. rel D x x) /\
     (!x y z :: set D. rel D x y /\ rel D y z ==> rel D x z) /\
     (!x y :: set D. rel D x y /\ rel D y x ==> (x = y)),
```

where the type of po is `":V#(V->V->bool)"`, and for convenience

```
|- !D. set D = FST D
|- !D. rel D = SND D.
```

Hence, we have used the type constructor `'#'` for pairs of higher order logic in the representing type.

Neither Isabelle's meta logic nor Isabelle/FOL support pairs. So, our only choice is to represent the set and relation as a pair in set theory. Alternatively, we could represent a partial order as a relation, defining the set as the reflexive elements. We chose the pair approach because it is closer to the HOL-ST formalization, which was done first, and because we then avoid proofs about what the elements of the set component of cpo constructions are. Hence, the type of the constant po is "i=>i" in Isabelle. Its definition is the same as the one above, though the right-hand side uses symbols of first order logic instead of higher order logic. The set and rel constants are defined by

```
"set(D) == fst(D)"
"rel(D, x, y) == <x, y> : snd(D)".
```

This makes rel a meta logic function of type "[i,i,i] => o"; set has type "i => i". It later turns out (when we consider subcpos) that it would have been more appropriate to define rel slightly differently as a meta logic function of just one argument (D) that returns a set function (or relation) of two arguments (x and y). In terms, we would then write the less convenient rel(D) ‘ x ‘ y (or <x, y> : rel(D)) instead of the present rel(D, x, y).

Next, we introduce the notion of a chain, which is a sequence of values in non-decreasing order:

```
|- !D X.
    chain D X = (!n. (X n) :: (set D)) /\ (!n. rel D(X n)(X(n + 1))).
```

Hence, in HOL-ST a chain is represented as a function of type ":num->V". Complete partial orders are then defined as follows

```
|- !D. cpo D = po D /\ (!X. chain D X ==> (?x. islub D X x))
```

where the notion of least upper bound (lub) is defined by

```
|- !D X x. isub D X x = x :: (set D) /\ (!n. rel D(X n)x)
|- !D X x.
    islub D X x = isub D X x /\ (!y. isub D X y ==> rel D x y).
```

Using Hilbert's choice operator we can give an expression for the least upper bound (if it exists):

```
|- !D X. lub D X = (@x. islub D X x).
```

In Isabelle/ZF, neither the meta logic nor first order logic support natural numbers so we must turn to the set of natural numbers, called nat, to represent infinite sequences. One could represent chains as a meta logic function of type "i=>i", where the first i would correspond to the set of natural numbers and the second would correspond to the underlying set of a cpo. However, this representation is problematic. In the definition of cpo we must quantify over chains but in first order logic this is impossible since we can only quantify over individuals, not meta logic functions like such chains of type "i=>i". Hence, in Isabelle chains must be represented as functions in set theory; thus chains have type i. The Isabelle definition of chain looks as follows:

```
"chain(D,X) == X:nat->set(D) & (ALL n:nat. rel(D,X‘n,X‘(succ(n))))".
```

The first-order definition of cpo is

```
"cpo(D) == po(D) & (ALL X. chain(D,X) --> (EX x. islub(D,X,x)))",
```

where islub is defined as in HOL-ST. As above, we define a constant for the least upper bound but we use the definite description operator instead of Hilbert's choice operator (which is not available, though it probably could be axiomatized):

```
"lub(D, X) == THE x. islub(D, X, x)".
```

The term "THE x. P(x)" is read 'the x such that $P(x)$' and requires both existence and uniqueness. In contrast, the HOL logic provides the choice operator, which just requires existence, and this is inherited by set theory, which thus automatically satisfies the axiom of choice. The use of the definite description operator made a few proofs slightly more complicated in Isabelle/ZF than in HOL-ST, due to the additional obligation of proving uniqueness.

A consequence of representing chains as functions in set theory is that the type checking, which ensures arguments of chains are numbers, must be done manually. Similarly, proving that chains are functions, and not just relations, and proving that they are functions on the right domains must be done manually as well (though usually the λ-abstraction is used and then it is only necessary to check the body of this due to a preproved theorem). Thus, proving that terms are chains is more complicated in Isabelle/ZF than in HOL-ST.

4.2 Continuous Functions

Monotonic and continuous functions have essentially the same definitions in the two systems. We only list the HOL-ST definitions:

```
|- !D E.
    mono(D,E) =
    {f :: (set D) -> (set E) |
     (!x y :: set D. rel D x y ==> rel E(f ^^ x)(f ^^ y))}
|- !D E.
    cont(D,E) =
    {f :: mono(D,E) |
     (!X. chain D X ==> (f ^^ (lub D X) = lub E(\n. f ^^ (X n))))}.
```

Functions are represented in set theory because we wish continuous functions to constitute a cpo, called the continuous function space, and the underlying set of a cpo must be a set. The continuous function space construction is defined as follows in HOL-ST:

```
|- !D E.
    cf(D,E) = cont(D,E),(\f g. !x :: set D. rel E(f ^^ x)(g ^^ x)).
```

However, the construction is defined slightly differently in Isabelle/ZF, due to the fact that the cpo pair, and more importantly the underlying relation, must be defined in set theory entirely:

```
"cf(D,E) ==
 <cont(D,E),
  {y : cont(D,E) * cont(D,E).
   ALL x:set(D). rel(E,(fst(y))`x,(snd(y))`x)}>".
```

In Isabelle/ZF, the relation must be constructed from existing sets, i.e. it must be constructed from the domains D and E in the function space. In contrast, the HOL-ST relation is just a higher order logic function. The Isabelle relation not only looks more complicated: due to additional type checking, it is also more complicated to use. Each time we define a construction on cpos, which we do twice below, we will experience a similar complication due to the set relation.

4.3 The Inverse Limit Construction

Next, we consider the definitions of some concepts associated with the inverse limit construction. Inverse limits may be viewed as "least upper bounds" of "chains" of cpos, not just of chains of elements of cpos as above. The ordering on elements of cpos is generalized to the notion of embedding morphisms between cpos. A certain constant `Dinf`, parameterized by a chain of cpos, can be proven once and for all to yield the inverse limit of the chain. This cannot be defined in higher order logic directly (assuming that the underlying set of a cpo is represented as a subset of a HOL type, as in [2]), since it yields a cpo of infinite tuples whose components may be in different cpos (subsets of types). Defining the construction in higher order logic would require a (probably difficult) conservative derivation of a universal type with dependent products. However, formalizing the construction is straightforward in set theory, exploiting the dependent product construction on sets.

Embedding morphisms come in pairs with projections, forming the so-called embedding projection pairs. The HOL-ST definition of (embedding) projection pairs is stated as follows:

```
|- !D E e p.
    projpair(D,E)(e,p) =
    e :: (cont(D,E)) /\
    p :: (cont(E,D)) /\
    (p O e = Id(set D)) /\
    rel(cf(E,E))(e O p)(Id(set E)).
```

The Isabelle/ZF definition is similar. 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

```
|- !D E e. emb(D,E)e = (?p. projpair(D,E)(e,p))
```

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

```
|- !D E e. Rp(D,E)e = (@p. projpair(D,E)(e,p)).
```

Again, these are the HOL-ST definitions; the Isabelle/ZF definitions are similar (though the definite description operator is used instead of the choice operator).

Embeddings are used to form chains of cpos in a similar way to the formation of chains from elements of cpos. Recall that standard chains are represented as logical functions of type `":num->V"` in HOL-ST and as set functions of type `i` in Isabelle/ZF. We choose to stick to this difference when representing embedding chains of cpos. Hence, the HOL-ST definition is stated like this:

```
|- !DD ee.
    emb_chain DD ee =
    (!n. cpo(DD n)) /\ (!n. emb(DD n,DD(SUC n))(ee n)).
```

And the Isabelle/ZF definition is:

```
"emb_chain(DD, ee) ==
   (ALL n:nat. cpo(DD ‘ n)) &
   (ALL n:nat. emb(DD ‘ n, DD ‘ succ(n), ee ‘ n))".
```

We do not quantify over embedding chains in any definitions immediately and therefore we could perhaps represent such chains as meta logic functions of type "`i=>i`" in Isabelle. However, the above choice is safer, in case it turns out that we later wish to quantify over chains.

One is often in a situation where a function can be represented in the meta logic or in the object logic (set theory). In general, one should only choose the first alternative if the function is not really part of a formalization and thus never would appear in the right-hand side of definitions (without its arguments). Hence, it is fine to use meta logic functions for constants in definitions (but one must be careful which I was not when I defined `rel` and `rho_emb`, see below). However, a choice must be made when functions are arguments of constants. For instance, due to the above criteria, we would use a meta logic function for the predicate of the following construction

```
"mkcpo(D, P) ==
   <{x: set(D) . P(x)},
    {x: set(D) * set(D) . rel(D, fst(x), snd(x))}>",
```

which is useful for constructing a subcpo of a cpo by restricting the set component according to the predicate. Thus, the type of `mkcpo` is "`[i,i=>o]=>i`". But most constants with function arguments would have a type of the form "`[i,i]=>o`", e.g. `emb_chain` above, where functions are set functions.

The notion of subcpo is defined as follows in Isabelle/ZF:

```
"subcpo(D, E) ==
   set(D) <= set(E) &
   (ALL x:set(D). ALL y:set(D). rel(D, x, y) <-> rel(E, x, y)) &
   (ALL X. chain(D, X) --> lub(E, X) : set(D))".
```

Both this and the previous definition of a subcpo constructor have simpler formulations in HOL-ST:

```
|- !D P. mkcpo D P = {x :: set D | P x},rel D
|- !D E.
    subcpo D E =
    (set D) Subset (set E) /\
    (rel D = rel E) /\
    (!X. chain D X ==> (lub E X) :: (set D)).
```

In both Isabelle/ZF definitions, the complications are due to a mismatch between the type of "`rel(D)`", namely "`[i,i]=>o`", and the type of the relation component of cpos, namely `i`. Probably, we made a bad choice in not representing "`rel(D)`" as a set function (or a set relation) instead of a logical function. The problem in the `mkcpo` definition is due to the fact that each component of a pair must be a set. The problem in the `subcpo` definition is that meta logic functions cannot be compared using FOL equality =.

The constant `mkcpo` is used to define the inverse limit construction on cpos as a subcpo of the infinite Cartesian product cpo. Let us first consider the HOL-ST definition of the infinite product:

```
|- !DD.
    iprod DD =
    (PI n :: Num. set(DD(Num2num n))),
    (\x y. !n. rel(DD n)(x ^^ (num2Num n))(y ^^ (num2Num n))).
```

The relation is defined componentwise and the set is the infinite tuples whose `i`'th component is in "`DD i`"; in general, the dependent product "`PI x :: X. Y[x]`" consists of the functions that map an element `x` of `X` to an element of `Y[x]`. This construction cannot be defined on HOL types (though it might be possible to derive a universal type with this construction). The annoying `num2Num` and `Num2num` conversions 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 in the long run (see [1]). The reason for this is associated with the choice of using the type of numbers in the representation of ordinary chains. To avoid the translation functions, we would have to stay within set theory all the time, since the dependent product construction is only available there.

The Isabelle/ZF definition of the infinite product construction on cpos is:

```
"iprod(DD) ==
   <PROD n:nat. set(DD ` n),
    {x: (PROD n:nat. set(DD ` n)) * (PROD n:nat. set(DD ` n)) .
     ALL n:nat. rel(DD ` n, fst(x) ` n, snd(x) ` n)}>".
```

Here, the translation functions are avoided since we do not have the choice of leaving set theory. However, for the same reason, the definition and use of the componentwise relation is much more complicated, since the relation must be a set constructed from existing sets.

The definitions of the inverse limit construction are essentially the same in the two system, both use the `mkcpo` constant. The HOL-ST definition is stated as follows:

```
|- !DD ee.
    Dinf DD ee =
    mkcpo
    (iprod DD)
    (\x.
      !n.
       (Rp(DD n,DD(SUC n))(ee n)) ^^ (x ^^ (num2Num(SUC n))) =
       x ^^ (num2Num n)).
```

The only difference is that the Isabelle/ZF definition quantifies over elements of the set of natural numbers (instead of over elements of the type as above) and it does not use translation functions. Informally, the underlying set of `Dinf` is defined as the subset of all infinite tuples x on which the n-th projection (retract) 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 inherited from the infinite product construction.

It takes a fairly large development to prove that `Dinf` yields an inverse limit of any chain of cpos. For the proof, we need an embedding of any element "`DD n`" of the chain "`(DD,ee)`" into the inverse limit "`Dinf DD ee`". This embedding is defined as follows in HOL-ST

```
|- !DD ee n.
     rho_emb DD ee n =
     (Fn x :: set(DD n). Fn m :: Num. (eps DD ee n(Num2num m)) ^^ x),
```

which was copied almost directly to Isabelle/ZF (removing the translation function):

```
"rho_emb(DD, ee, n) ==
   lam x:set(DD ‘ n). lam m:nat. eps(DD, ee, n, m) ‘ x".
```

The definitions of the constant called `eps` in both systems are not important here[2]. While the definition of `rho_emb` worked fine in HOL-ST we realized at a later stage that the Isabelle/ZF definition should have been

```
"rho_emb(DD, ee) ==
  lam n:nat. lam x:set(DD ‘ n). lam m:nat. eps(DD, ee, n, m) ‘ x",
```

where `rho_emb` is a logical function of just two arguments instead of three; thus, while "`rho_emb(DD,ee)`" before was a logical function of type "`i => i`", it is now a set function. The present representation would be unfortunate if we wanted to define a constant for the property that `Dinf` always yields the inverse limit of a cpo. This is not possible using a meta logic function for "`rho_emb(DD,ee)`", since the definition would need to quantify over such sequences of embeddings. Furthermore, similar sequences like the sequences of cpos `DD` and embeddings `ee` are represented as object logic functions. So, a constant may be a meta logic function of some arguments and an object logic function of other arguments. If one is not careful the wrong choices are made.

5 Proofs

In the previous section, we concentrated on the differences between using HOL-ST and Isabelle/ZF to formalize the definitions of some basic concepts and the inverse limit construction of domain theory. In this section, we dive into a discussion of how the two systems support the proofs of related theorems.

Due to limitations of Isabelle's first-order set theory, we were forced to work in set theory in situations where we could stay in higher order logic in HOL-ST. As mentioned above, this obviously yields more complicated proofs in Isabelle, in the sense of more typing conditions to prove. For instance, in a backward proof of a statement saying that two functions are related by the continuous function space relation, we would first rewrite with the HOL-ST theorem

[2]By composing embeddings (and projections) eps generalizes the embeddings "ee n" between consecutive cpos of a chain to convert between any two cpos.

```
|- !D E f g.
    rel(cf(D,E))f g = (!x. x :: (set D) ==> rel E(f ^^ x)(g ^^ x)),
```

but in the Isabelle/ZF the first step would be to resolve with the theorem

```
"[| !!x. x : set(D) ==> rel(E, f ` x, g ` x); f : cont(D, E);
    g : cont(D, E) |] ==> rel(cf(D, E), f, g)",
```

which, in contrast to the HOL-ST theorem, contains typing assumptions. The same thing is true of the other constructions on cpos, like the infinite Cartesian product and the inverse limit constructions. Similarly, the necessity of representing chains as set functions yields a number of additional proof obligations in the Isabelle/ZF proofs.

Despite these additional proof obligations, Isabelle proofs are usually shorter in terms of number of lines, and easier to write. Usually, backward proofs are reduced in size (number of lines) by more than 50% and in some cases by 75%. The main reasons for this are Isabelle's support for unknown variables for quantifier reasoning and the design of its proof infrastructure.

The main method of proof in Isabelle is based on resolution using higher order unification, which supports both the forward and the backward style of proof. In fact, the same theorem can be used as an inference rule by forward resolution and as a tactic by backward resolution. In this way, Isabelle elegantly avoids the need for a large collection of ML functions implementing derived inference rules and tactics. Furthermore, it supports a compact notation for proofs since the main resolution tactic can be employed with a theorem list argument, and repeated.

Further, the notion of resolution in Isabelle supports 'real' backward proof better than in HOL. One almost always works from the conclusion of a goal backward towards the assumptions, which is supported by Isabelle resolution tactics. In HOL, one often ends up doing a lot of sometimes ugly assumption hacking working forward using HOL resolution from the assumptions towards the conclusion. 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 version of Isabelle resolution) are not supported well in HOL.

Real backward strategies are useful due the fact that many theorems have assumptions. It is irritating to have to first derive the antecedents of theorems for HOL resolution. On the other hand, a negative consequence of using theorems in a real backward fashion is that existential quantifiers are often introduced. For instance, this happens when we employ the transitivity of a cpo relation or the fact that function composition preserves the function set (or continuity or embeddings):

```
"[| g : A -> B; f : B -> C |] ==> f O g : A -> C".
```

In HOL, we must provide witnesses for existentials on the spot and manually. But in Isabelle, both universally and existentially quantified variables are represented as unknown variables that are usually instantiated behind the scenes in proofs, possibly in stages.

Finally, the subgoal module provides a kind of flat structure on proof states which makes it possible to access all goals at any time and to prove many (or all) subgoals by just repeating a tactic—no matter where the subgoals would appear in a HOL proof tree.

In the rest of this section, we illustrate some of the points made above by looking at a couple of examples of proof in Isabelle/ZF.

5.1 Backward Proof

The purpose of the first example is to illustrate the benefits of the subgoal module. We prove the following theorem about relating elements of chains:

```
"[| chain(D, X); cpo(D); n : nat; m : nat |] ==>
 rel(D, X ‘ n, X ‘ (m #+ n))".
```

The theorem states that an element of a chain is related to (itself or) any element which appears later in the chain. Thus, it generalizes the definition of chains, which just says that an element is related to its successor. The theorem clearly holds due to the (reflexivity and) transitivity of cpos.

The goal is set up as follows:

```
> val prems = goal basic.thy
# "[|chain(D,X);cpo(D);n:nat;m:nat|] ==> rel(D,X‘n,(X‘(m #+ n)))";
Level 0
rel(D, X ‘ n, X ‘ (m #+ n))
 1. rel(D, X ‘ n, X ‘ (m #+ n))
val prems = ["chain(D, X)  [chain(D, X)]", "cpo(D)  [cpo(D)]",
   "n : nat  [n : nat]", "m : nat  [m : nat]"] : thm list
```

The goal command returns the assumed premises of the statement, which are not part of the proof state of the subgoal module. These are bound to the theorem list called prems. The output from the subgoal module is the three lines starting with 'Level 0'. The level of a proof is increased every time we apply a tactic. The second line is really redundant since it remains the same throughout the goal; it just reminds us of what the original goal of the proof was. Finally, the third line, labelled '1', shows subgoal one. Since there are no more labelled lines, this is the only subgoal at this level.

The proof is conducted by an induction on the natural number m. Induction is provided by the theorem

```
nat_induct:
  "[| ?n : nat; ?P(0);
     !!x. [| x : nat; ?P(x) |] ==> ?P(succ(x)) |] ==> ?P(?n)",
```

which supports the proof of any property P by induction on a variable n. Free variables of theorems are represented by unknowns, which may be interpreted as if they are implicitly universally quantified (above we removed question marks for readability). In order to apply the theorem, it is necessary to instantiate the variable n to m manually, then the proper instantiation of P, which is a certain meta level function, is constructed automatically, and meta level β-conversion is performed behind the scenes:

```
> by(res_inst_tac [("n","m")] nat_induct 1);
Level 1
rel(D, X ‘ n, X ‘ (m #+ n))
 1. m : nat
 2. rel(D, X ‘ n, X ‘ (0 #+ n))
 3. !!x. [| x : nat; rel(D, X ‘ n, X ‘ (x #+ n)) |] ==>
         rel(D, X ‘ n, X ‘ (succ(x) #+ n))
val it = () : unit
```

The by command applies a tactic to the proof state. The res_inst_tac tactic first instantiates the theorem argument according to the list argument, and then resolves the specified subgoal (here no. 1) with the resulting theorem. Above the result is three subgoals, corresponding to the three assumptions of induction. The subgoal module allows us to access any one of them, or all of them at the same time, in the following steps.

Our next step is to simplify all subgoals using theorems about addition. Therefore we apply the simplifier tactic[3] with the built-in simplification set about arithmetic (which is not very powerful):

```
> by(ALLGOALS(simp_tac arith_ss));
Level 2
rel(D, X ‘ n, X ‘ (m #+ n))
 1. m : nat
 2. rel(D, X ‘ n, X ‘ n)
 3. !!x. [| x : nat; rel(D, X ‘ n, X ‘ (x #+ n)) |] ==>
         rel(D, X ‘ n, X ‘ succ(x #+ n))
val it = () : unit
```

Subgoal 1 is proved below by a premise; it is left unchanged by simplification since the simplifier does not have acces to the premise (unless we provide it manually). Subgoal 2 follows from the reflexivity property of cpos (see below) and subgoal 3 from the transitivity property and the induction hypothesis. Transitivity is stated by

```
cpo_trans:
  "[| cpo(?D); rel(?D, ?x, ?y); rel(?D, ?y, ?z); ?x : set(?D);
     ?y : set(?D); ?z : set(?D) |] ==> rel(?D, ?x, ?z)".
```

We first resolve with transitivity on subgoal 3:

[3]Simplification supports contextual conditional rewriting. It only uses a conditional rewrite theorem if it can prove its assumptions. Some of my proofs could have been easier if there was a contextual conditional rewriter which created a subgoal for the assumptions that it could not prove.

```
> br cpo_trans 3;
Level 3
rel(D, X ‘ n, X ‘ (m #+ n))
 1. m : nat
 2. rel(D, X ‘ n, X ‘ n)
 3. !!x. [| x : nat; rel(D, X ‘ n, X ‘ (x #+ n)) |] ==> cpo(D)
 4. !!x. [| x : nat; rel(D, X ‘ n, X ‘ (x #+ n)) |] ==>
         rel(D, X ‘ n, ?y1(x))
 5. !!x. [| x : nat; rel(D, X ‘ n, X ‘ (x #+ n)) |] ==>
         rel(D, ?y1(x), X ‘ succ(x #+ n))
 6. !!x. [| x : nat; rel(D, X ‘ n, X ‘ (x #+ n)) |] ==> X ‘ n : set(D)
 7. !!x. [| x : nat; rel(D, X ‘ n, X ‘ (x #+ n)) |] ==> ?y1(x): set(D)
 8. !!x. [| x : nat; rel(D, X ‘ n, X ‘ (x #+ n)) |] ==>
         X ‘ succ(x #+ n) : set(D)
val it = () : unit
```

This yields six new subgoals, labelled 3 to 8. Now, the remaining eight subgoals are all proved in a single stroke:

```
> brr(cpo_refl::chain_in::chain_rel::nat_succI::add_type::prems)1;
Level 4
rel(D, X ‘ n, X ‘ (m #+ n))
No subgoals!
val it = () : unit
```

where the following theorems are used

```
cpo_refl:
  "[| cpo(?D); ?x : set(?D) |] ==> rel(?D, ?x, ?x)"
chain_in:
  "[| chain(?D, ?X); ?n : nat |] ==> ?X ‘ ?n : set(?D)"
chain_rel:
  "[| chain(?D, ?X); ?n : nat |] ==> rel(?D, ?X ‘ ?n, ?X ‘ succ(?n))"
nat_succI:
  "?n : nat ==> succ(?n) : nat"
add_type:
  "[| ?m : nat; ?n : nat |] ==> ?m #+ ?n : nat",
```

and brr is a short-hand defined by

```
fun brr thl n = by(REPEAT(ares_tac thl n));
```

which repeatedly applies a tactic to a subgoal. The brr command can solve many consecutive subgoals since when a subgoal, n say, has been proved it is removed from the goal stack, and subgoal n+1 becomes the new subgoal n; all subgoals below subgoal n are shifted. The tactic ares_tac tries to solve a goal by assumption and, if this fails, it tries to resolve with one of the theorems in the theorem list argument (if this is impossible it fails). A subgoal is solved, or proved, by assumption if its conclusion can be unified with one of the assumptions. In the proof, we did not apply transitivity repeatedly since this would loop. Note that applying transitivity (early) introduces an unknown, called `?y1(x)` above, representing an existential quantified variable for which a witness is constructed automatically in the proof. Also note that we applied transitivity early. In a HOL proof, we would apply transitivity late using `IMP_RES_TAC` (see below), having first derived the antecedents of the theorem by forward proof. Real backwards proofs are more direct and natural than the semi-backwards proof one gets from using HOL resolution too much.

This was a four lines long proof:

```
val prems = goal basic.thy
    "[|chain(D,X);cpo(D);n:nat;m:nat|] ==> rel(D,X`n,(X`(m #+ n)))";
by(res_inst_tac [("n","m")] nat_induct 1);
by(ALLGOALS(simp_tac arith_ss));
br cpo_trans 3;
brr(cpo_refl::chain_in::chain_rel::nat_succI::add_type::prems)1;
val chain_rel_gen_add = result();
```

A typical HOL proof could be this eleven lines long proof:

```
let chain_rel_gen_add = prove_thm(`chain_rel_gen_add`,
 "!D X. chain D X ==> cpo D ==> (!n m. rel D(X n)(X(m+n)))",
 GEN_TAC THEN GEN_TAC THEN DISCH_TAC THEN DISCH_TAC THEN GEN_TAC
 THEN INDUCT_TAC THEN PORT[ADD_CLAUSES]
 THENL
 [IMP_RES_THEN (ASSUME_TAC o SPEC "n:num") chain_in
  THEN IMP_RES_TAC cpo_refl
 ;IMP_RES_THEN (ASSUME_TAC o SPEC "m+n") chain_rel
  THEN IMP_RES_THEN (\th.
    ASSUME_TAC(SPEC"n:num"th)
    THEN ASSUME_TAC(SPEC"m+n"th)
    THEN ASSUME_TAC(SPEC"SUC(m+n)"th)) chain_in
  THEN IMP_RES_TAC cpo_trans]);;
```

which exploits the HOL theorems

```
chain_in:
  |- !D X. chain D X ==> (!n. (X n) :: (set D))
chain_rel:
  |- !D X. chain D X ==> (!n. rel D(X n)(X(n + 1)))
cpo_refl:
  |- !D. cpo D ==> (!x. x :: (set D) ==> rel D x x)
cpo_trans:
  |- !D.
       cpo D ==>
       (!x y z.
         rel D x y ==> rel D y z ==> x :: (set D) ==>
         y :: (set D) ==> z :: (set D) ==> rel D x z)
```

Thus, the HOL proof is essentially the same as the Isabelle proof above, but it is a strange mixture of forward and backward proof and requires a larger effort to do; for instance, variables are often instantiated manually. Using transitivity early as in the Isabelle proof, employing a variant of MATCH_MP_TAC, would introduce an existential, which we would have to provide a witness for on the spot.

In one sense, the HOL proof is simpler than the Isabelle proof: it applies fewer theorems. The Isabelle proof must prove the additional type assumptions on the chain theorems but this is straightforward in this example. However, the additional theorems are a product of the representation in set theory, not of the proof system (see Section 4).

The above observations also hold in general for the more difficult proofs: proofs are usually shorter and easier to write in Isabelle, despite the fact that they perform more steps. For instance, one example had a 42 lines long HOL proof with a lot of ugly assumption hacking whereas the Isabelle proof was only 10 lines long, and a lot easier to write. In most cases, Isabelle proofs were more than 50% shorter than HOL proofs. Further, they required less thought to write. Lists of theorems for brr, which was used a lot, were easy to write by just looking at goals. Isabelle takes care of applying the theorems, instantiating unknowns and proving the assumptions by adding them as new subgoals; we do not have to think about the tree structure of proofs and about which tactics (or theorems) are applied where.

5.2 Forward Proof

Isabelle forward proofs by resolution can also be performed very elegantly. However, in the present work forward proof is used much less than in HOL, due to better support for real backward proofs. Meta logic theorems can be joined by the unification-based resolution and therefore interpreted as inference rules. As an example consider the following (useful) theorem

```
> cont_fun RSN(2,cont_fun RS comp_fun_apply);
val it =
"[| ?g : cont(?D1, ?E2); ?f : cont(?E2, ?E4); ?a : set(?D1) |] ==>
 (?f O ?g) ` ?a = ?f ` (?g ` ?a)" : thm
```

which states a composition application theorem for continuous functions. It is obtained easily from the corresponding theorem about set functions by exploiting that continuous functions constitute a subset of the function set:

```
cont_fun:
  "?f : cont(?D, ?E) ==> ?f : set(?D) -> set(?E)"
comp_fun_apply:
  "[| ?g : ?A -> ?B; ?f : ?B -> ?C; ?a : ?A |] ==>
   (?f O ?g) ` ?a = ?f ` (?g ` ?a)".
```

The RSN function resolves the conclusion of the first theorem with a specified assumption of the second theorem. RS resolves with the first assumption. The corresponding forward proof in HOL-ST could be

```
GEN_ALL(DISCH_ALL
(MATCH_MP ApO
 (CONJ(MATCH_MP cont_fun(ASSUME "g :: cont(D,E)"))
      (MATCH_MP cont_fun(ASSUME "f :: cont(E,E')"))))),
```

where

```
cont_fun:
  |- !f D E. f :: cont(D,E) ==> f :: (set D) -> (set E)
ApO:
  |- !f g X Y Z.
      f :: Y -> Z /\ g :: X -> Y ==>
      (!x. x :: X ==> ((f O g) ^^ x = f ^^ (g ^^ x))).
```

This proof is both longer and less convenient to write. Note that the theorem would introduce three existentially quantified variables if it was used in a backward fashion.

6 Conclusions

We have presented a comparison of HOL-ST and Isabelle/ZF based on a case study from domain theory. The case study formalizes a construction for inverse limits of embedding projection chains of cpos. This formalization exploits set theory in an essential way since it requires a dependent product construction that cannot be defined on HOL types. The main observations, which are summarized below, say that HOL-ST is supported by the powerful HOL logic, which provides a more convenient set theory allowing set and type theoretic reasoning to be mixed to advantage. Isabelle/ZF provides better proof support for set theory. Set theory introduces a lot of set membership assumptions in theorems as well as the need for real backward proof strategies and good support for quantifier reasoning. Generally speaking, HOL lacks ways of handling conditional theorems conveniently, and does not provide the support for unknown variables for quantifier reasoning available in Isabelle.

It is advantageous to be able to exploit higher order logic where possible, as in HOL-ST, since 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. On the other hand, a disadvantage of mixing higher order logic and set theory as in HOL-ST is the need for translation functions to identify HOL types with their corresponding sets. Therefore, it is not obvious whether set theory in higher order logic is right, or just more support for set theory in first order logic is needed.

6.1 First-order versus Higher-order Set Theory

The HOL-ST formalization presented above (see [1] for more details) 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 represented using the set `Num` and cpos could have been represented by (non-reflexive) relations of set theory.

It makes a difference whether sets of set theory or types of HOL are used. Using the latter, set membership (type) conditions are avoided and furthermore, type checking is automatic (static) in ML. Using sets, type checking, i.e. ensuring terms are in the right sets, is done by theorem proving (dynamic), 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 formalization should be done in set theory and which should be done in higher order logic. As noted in Section 4, 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 dealing with the inconvenience of the translation functions at this stage since many theorems and proofs became quite horrible later, with the set approach [1]. This in turn is related to the choice of representing ordinary chains in higher order logic.

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. Isabelle's polymorphic weak higher-order meta logic is meant for expressing and reasoning in logic instantiations of Isabelle, the so-called object-logics, and it does not provide basic types such as pairs and numbers. Thus, chains must be represented as set functions and cpos must be represented as pairs of set theory such that both the set and the relation components must be sets. In principle, the function type of the 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. This is a difficult part of using Isabelle/ZF, deciding whether to represent functions in set theory or in the meta logic.

The consequence of working mostly 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 we must prove that it is a set function, i.e. that it is a relation on the right sets that defines a function. 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.

6.2 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 typing conditions, they are in fact much shorter in terms of number of lines, and easier to write.

Isabelle provides an elegant proof infrastructure. Meta logic theorems can express both object logic theorems, inference rules and tactics. There is not 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 only 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.

Backward proofs are often more than 50% shorter in Isabelle. The subgoal module provides a kind of flat structure on proof states which makes it possible to access all goals at any 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. Further, the notion of resolution in Isabelle supports 'real' backward proofs better than they are supported in HOL. Isabelle resolution tactics allowed one to almost always work from the conclusion of a goal backward towards the assumptions. In HOL, one often ends up doing a lot of sometimes ugly assumption hacking, working forward from the assumptions towards the conclusion. More natural backward strategies like conditional rewriting and a matching modus ponens style strategy are not well supported in HOL. These are useful due the fact that many theorems have set membership assumptions.

HOL 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. Existential quantifiers are often introduced by the backward strategies mentioned above, for instance, when employing the transitivity of a cpo relation or the facts that function composition preserves the function set, continuity or embeddings. In HOL proofs, it is also often necessary to instantiate universally quantified variables manually.

These cases are usually handled automatically in Isabelle. It provides a notion of unknown variables, which can be instantiated in proofs. This means that witnesses and proper instantiations of universally quantified variables are constructed behind the scenes, possibly in stages. Unification is essential for allowing this.

Acknowledgements

The research described here is partly supported by EPSRC grant GR/G23654 and partly by an HCMP fellowship under the EuroForm network. I am grateful to Mike Gordon for encouragements and discussions concerning this work. I would also like to thank Larry Paulson for answering my questions about Isabelle promptly and the Isabelle users at Cambridge, in particular Sara Kalvala, for help and discussions. Mark Staples commented on a draft of the paper.

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

[4] M.J.C. Gordon. Merging HOL with set theory: preliminary experiments. Technical Report 353, University of Cambridge Computer Laboratory, November 1994.

[5] S. Kalvala. A gentle introduction to Isabelle. Draft, University of Cambridge Computer Laboratory, 1994.

[6] L. C. Paulson. *Logic and Computation: Interactive Proof with Cambridge LCF.* Cambridge Tracts in Theoretical Computing 2, Cambridge University Press, 1987.

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

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

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

[10] G. Plotkin. *Domains.* Course notes, Department of Computer Science, University of Edinburgh, 1983.

[11] M. Smyth and G. D. Plotkin. The category-theoretic solution of recursive domain equations. *SIAM Journal of Computing*, 11, 1982.