LCF Examples in HOL

STEN AGERHOLM

BRICS, Department of Computer Science, University of Aarhus, DK-8000 Aarhus C, Denmark

The LCF theorem prover provides a logic of domain theory and is useful for reasoning about nontermination, general recursive definitions and infinite-valued datatypes like lazy lists. Because of the continued presence of bottom (undefined) elements, it is clumsy for reasoning about finite-valued datatypes and strict functions. By contrast, the HOL theorem prover provides a logic of set theory (without a notion of undefinedness) and supports reasoning about finite-valued datatypes and primitive recursive functions well. In this paper, a number of examples are used to demonstrate that an extension of HOL with concepts of domain theory combines the best of both systems. The examples illustrate how domain and set theoretic reasoning can be mixed to advantage, allowing the full use of both. Moreover, a larger example presents a proof of correctness of the unification algorithm which shows how the painful reasoning about bottom in LCF can be eliminated.

1. INTRODUCTION

The LCF system (Gordon *et al.*, 1979; Paulson, 1987) is a theorem prover which implements a version of Scott's Logic of Computable Functions, a first order logic of domain theory. The LCF logic has a domain theoretic semantics and provides the concepts and techniques of fixed point theory to reason about nontermination and arbitrary recursive (continuous) functions. For instance, LCF has been successfully applied to reason about lazy (infinite-valued) datatypes and lazy evaluation. However, reasoning about obviously total (or strict) functions and finite-valued datatypes is clumsy in LCF (Paulson, 1984b).

By contrast, the HOL system (Gordon *et al.*, 1993) provides a version of Church's higher order logic (simple type theory), which has a set theoretic semantics. It has no built-in notion of nontermination (or undefined elements) and all functions are total. It supports reasoning about finite-valued datatypes and primitive recursive functions well, but has no support for non-primitive recursive functions and non-wellfounded datatypes.

The HOL-CPO system (Agerholm, 1994c) attempts to combine the best of both LCF and HOL within a single system. It provides a formalization of central concepts of domain theory in HOL along with a number of proof functions and other tools to support the use of the formalization in practice. One may view HOL-CPO as an embedding of the LCF system in HOL.

Roughly speaking, HOL-CPO supports all LCF reasoning. However, it has advantages over LCF since (1) it inherits the underlying higher order logic and proof infrastructure of the HOL system, and (2) it provides direct access to domain theory. These points are the consequences of *embedding* semantics (as in HOL-CPO) rather than *implementing* logic (as in LCF). The main advantage of (1) is that we become able to mix domain and set theoretic reasoning in HOL-CPO. In particular, by exploiting the set theoretic basis of higher order logic, the continual and very annoying fiddling with the bottom element in LCF (Paulson, 1985) can be essentially eliminated.

In contrast to (2), domain theory is only present in the underlying logic of the LCF system through axioms and primitive rules of inference. For instance, there is no semantic definition of the fixed point operator, nor of the admissibility condition on predicates for fixed point induction. The fixed point operator is axiomatized by the fixed point property (an axiom) and fixed point induction (a primitive rule of inference). Further, the fixed point induction rule implements a syntactic check for admissibility which is not complete.

By exploiting the semantic definitions of these concepts in HOL-CPO, we have no such limitations. Fixed point induction can be derived as a theorem from the definition of the fixed point operator and reasoning directly about fixed points allows more theorems to be proved than with just fixed point induction. Moreover, syntactic checks for admissibility can be implemented as in LCF, but admissible predicates not accepted by the syntactic checks can be proved to be admissible from the semantic definition manually, using the proof infrastructure of HOL.

In this paper, we focus on the advantages of (1). A number of examples are presented to demonstrate that HOL-CPO supports and extends both the HOL and the LCF worlds. We define nonterminating and arbitrary recursive functions in domain theory and reason about finite-valued types and total functions in 'set theory' (higher order logic) before turning to domain theory. The examples have already been done in LCF by Paulson which makes a comparison of the two systems possible. The first two examples, on natural numbers and lazy sequences, are described in Chapter 10 of the LCF book (Paulson, 1987) and the third and larger example is based on Paulson's version of a correctness proof of the unification algorithm (Paulson, 1985), originally presented in (Manna *et al.*, 1981). Before we turn our attention to the examples we review the HOL, HOL-CPO and LCF systems in each of the following three sections respectively.

2. THE HOL SYSTEM

The HOL theorem prover (Gordon *et al.*, 1993) is a a mechanized proof-assistant for proving theorems in higher order logic. The HOL logic, a version of Church's simple type theory, and all theorem proving support is built on top of a functional programming environment ML (Meta Language). Terms, types and theorems of the logic are represented as datatypes in ML. Inference rules are ML functions which take a number of theorems (the premises) as arguments and produce a theorem as a result (the conclusion); primitive inference rules are constructors of the abstract datatype of theorems. A proof is a derivation using a number of inference rules, proved theorems and axioms (the HOL logic has 5 axioms and 8 primitive rules of inference).

Inference rules support forward proofs of theorems. However, a more natural goal-directed (backwards) proof style is also supported, by the subgoal package. Proofs can be constructed by applying tactics interactively, in order to reduce goal terms to truth. A tactic is an ML function which typically implements the backwards use of one or more inference rules (and theorems).

The terms of the HOL logic may be variables, constants, λ-abstractions or applications. The usual logical connectives are represented as constants. All terms must be well-typed; e.g. in applications $t_1\,t_2$, t_1 must have a function type $t_1 : \alpha \rightarrow \beta$ where $t_2 : \alpha$. Types can be atomic types (like *bool* for the boolean truth values), type variables (like α and β), compound types (like $\alpha \times \beta$ for the product type), and function types. Some built-in types are *num* for the natural numbers and (α)*list* for finite lists of elements of any type α.

The HOL logic has a set theoretic semantics (Gordon *et al.*, 1993, Chapter 15). All types denote sets and the function type denote total functions of set theory.

Among its more unusual notions, the HOL logic provides a choice operator ϵ which can be used to select some element of a type such that a predicate holds, e.g. $\epsilon x.\, P[x]$. If this is not possible, i.e. if the predicate is everywhere false, then it returns an arbitrary value of the type; any type must be non-empty. There is a built-in HOL constant called ARB which equals $\epsilon x : \alpha.\, \mathsf{T}$, where T is the boolean value for truth, and always gives an arbitrary but fixed value of any type α.

The HOL logic is organized in hierarchies of theories which contain collections of types, constants, definitions, axioms, and theorems. Once theorems have been proved, they can be saved and used over and over again. The purpose of the HOL system is to provide tools for constructing such theories.

Theories can be extended with new constants and types by giving definitions and axioms. Definitional extension preserves consistency of the HOL logic, because new constants and types are defined in terms of existing ones. Axiomatic extension may not be safe in this sense and is usually not accepted in the HOL community. The present developments are purely definitional.

In general, it is not easy to define recursive functions and types since one must first prove their existence in the logic. However, HOL supports certain concrete recursive finite-valued datatypes and primitive recursive functions on these types, due to the type definition package (Melham, 1989). Though definitional extensions, which have not been automated, can sometimes require a lot of work, the HOL system supports extensions well, through its expressive underlying logic and the meta language ML which can be used to program special-purpose proof functions and other tools.

HOL has a large collection of built-in types, theorems and proof tools to support all kinds of reasoning. The presence of these is important since then one does not have to start from scratch when a new extension is considered. For instance, the predicate sets library (Melham, 1992) was used in the present development. Sets are represented as subsets of HOL types by predicates of type $\alpha \rightarrow$ *bool*. It provides set notation and the usual operations on sets. In particular, we shall use a universal set constructor UNIV : $\alpha \rightarrow$ *bool* which equals the always true predicate $\lambda x.\, \mathsf{T}$ and therefore contains all elements of any underlying HOL type α.

3. HOL-CPO

In this section we provide an overview of the formalization of domain theory and some of the associated tools (Agerholm, 1994a; Agerholm, 1994c). This extension of HOL, called HOL-CPO, constitutes an integrated system where the domain theory constructs look almost primitive (built-in) to the user. Many facts are proved behind the scenes to support this view. In order to read the paper, it is not necessary to know the semantic definitions of the domain theory which is used. Therefore, the presentation below shall be very brief. More details can be sought in (Agerholm, 1994c). A good introduction on domain theory is provided in (Winskel, 1993), on which the formalization is based.

3.1. Basic Concepts

Domain theory is the study of complete partial orders (cpos) and continuous functions. These notions are introduced in HOL by their semantic definitions. A *complete partial order* is a pair which consists of a set and a relation satisfying the predicate

$$\mathsf{cpo} : (\alpha \rightarrow bool) \times (\alpha \rightarrow \alpha \rightarrow bool) \rightarrow bool.$$

If (A, R) is a cpo then the underlying relation R is a partial ordering (reflexive, transitive and antisymmetric) on all elements of the underlying set A and there exists a *least upper bound* (lub) for all non-decreasing sequences $X : num \to \alpha$ of elements in A; here, non-decreasing means that $R(Xn)(X(n+1))$ holds for all n. Such sequences are called *chains* (or ω-chains).

In the literature, a cpo is usually thought of as a set with an associated ordering relation. We can provide much the same useful notion in HOL by introducing rel_D to stand for the underlying relation of a cpo D and using $t \in D$ to say that a term t is an element of the underlying set of D.

Note that we do not require cpos to have a so-called *bottom* (or undefined) element, i.e. a least element $\mathsf{bot}_D \in D$ such that $\mathsf{rel}_D\,\mathsf{bot}_D\,x$ for all $x \in D$. Cpos which have a bottom are called *pointed* cpos and satisfy the HOL predicate pcpo. In the following, a cpo may or may not contain a bottom element unless we say explicitly that it is pointed.

A *continuous* function from a cpo D to a cpo E is a HOL function $f : \alpha \to \beta$ such that the term $\mathsf{cont}\,f(D,E)$ is true (α and β are the underlying types of D and E, respectively). It must be monotonic with respect to the underlying relations and preserve lubs of chains in the sense that f applied to the lub of a chain X in D is equal to the lub of the chain $f(X\,n)$ in E. In addition, f must be *determined* by its action on elements of the domain cpo D. This means that on elements outside D it should always return the fixed arbitrary value ARB. The determinedness restriction is necessary to prove that continuous functions constitute a cpo and is induced by the fact that we work with partial HOL functions between subsets of HOL types (corresponding to the underlying sets of cpos). Determinedness occurs everywhere and is one of the main disadvantages of the formalization. In particular, functions must be written using a dependent lambda abstraction

$$(\mathsf{lambda}\,D\,f)(x) = \begin{cases} f(x) & \text{if } x \in D \\ \mathsf{ARB} & \text{otherwise} \end{cases}$$

to ensure they are determined. Therefore, many functions (like Fix and Ext below) become parameterized by cpo variables, which are the free term variables of the domains on which they work.

The conditions on complete partial orders and continuous functions ensure the existence of a *fixed point operator*, called Fix, which is useful to define general (non-primitive) recursive functions and other infinite values. Applied to a continuous function f on a pointed cpo E, it yields a fixed point of f:

$$[\mathsf{cont}\,f(E,E);\mathsf{pcpo}\,E] \vdash f(\mathsf{Fix}\,E\,f) = \mathsf{Fix}\,E\,f.$$

In fact, it yields the *least* fixed point:

$$[\mathsf{cont}\,f(E,E);\mathsf{pcpo}\,E; x \in E] \\ \vdash (f\,x = x) \Rightarrow \mathsf{rel}_E(\mathsf{Fix}\,E\,f)x.$$

Terms in square brackets are the assumptions of theorems. The term $\mathsf{Fix}\,E\,f$ equals the least upper bound of the chain $\bot \sqsubseteq f(\bot) \sqsubseteq f(f(\bot)) \sqsubseteq \ldots$, where $\bot$ stands for bot_E and $\sqsubseteq$ for rel_E. Fix is defined using the dependent lambda abstraction and is therefore parameterized by E above.

The proof principle of fixed point induction has been derived as a theorem from the definition of the fixed point operator:

$$[\mathsf{incl}\,P\,E;\mathsf{cont}\,f(E,E);\mathsf{pcpo}\,E] \\ \vdash P(\mathsf{bot}_E) \wedge (\forall x.\ P\,x \Rightarrow P(f\,x)) \Rightarrow P(\mathsf{Fix}\,E\,f).$$

It can be used to prove properties of fixed points which are stated as inclusive predicates. A predicate is an *inclusive* subset of a cpo if it contains the lubs of chains in the subset.

The theorems presented above may be read as inference rules, where the left-hand side of an implication is interpreted as the premise(s) and the right-hand side as the conclusion. The assumptions, i.e. the terms in square brackets, may be interpreted as the side conditions. The domain theory constructs in the side conditions have complex (but straightforward) definitions in higher order logic. Therefore, it would be arduous to prove the conditions manually in HOL each time a theorem is applied. A number of syntactic-based proof functions have been implemented to automate such proofs. These support the informal notations for cpos, typable terms and inclusive predicates presented next.

3.2. Notations for Cpos and Pointed Cpos

Two special-purpose proof functions implement informal syntactic notations for cpos and pointed cpos. These functions are called the *cpo* and *pcpo provers*.

A notation for cpos may be described as follows

$$D ::= \mathsf{discrete}\,Z \mid \mathsf{lift}\,D \mid \mathsf{cf}(D,E) \mid \mathsf{sum}(D,E) \mid \ldots,$$

where Z is some HOL set and D and E are cpos. The constructors of the notation are described below. The notation can be extended with new constructions at any time. A new constructor of the form $\mathsf{C}(D_1,\ldots,D_n)$ can extend the notation in two ways. Either it abbreviates a term which fits within the notation already, or a theorem states that it yields a cpo if its arguments are cpos.

The *discrete* construction associates the discrete ordering (identity) with a set and is useful for making HOL sets into cpos. For instance, the type of natural numbers can be used to define the discrete cpo of natural numbers discrete(UNIV : *num* $\to$ *bool*) using the universal set UNIV, which equals the predicate $\lambda x.\mathsf{T}$. We can extend the notation with Nat by defining

$$\vdash \mathsf{Nat} = \mathsf{discrete}\,\mathsf{UNIV}.$$

A construction called *lifting* can be used to extend a cpo with a bottom element. The bottom of a lifted cpo $\mathsf{lift}\,D$, where D is any cpo, is written as Bt and all other elements are written as $\mathsf{Lft}\,d$ for some $d \in D$. The constants Bt and Lft are the constructors of a new datatype in HOL which associates a new element with any type. It is the underlying relation of the lifting construction which makes Bt into a bottom of $\mathsf{lift}\,D$.

Next, the *continuous function space* associates the pointwise ordering relation with the set of all continuous functions between two cpos. Assuming D and E are cpos, their continuous function space is written as $\mathsf{cf}(D,E)$. Therefore, the two statements $f \in \mathsf{cf}(D,E)$ and $\mathsf{cont}\,f(D,E)$ are equivalent, and we shall usually use the former.

Finally, the *sum* construction takes the disjoint sum of two cpos. It makes a copy of the elements of each of the cpos, and the underlying ordering relations are inherited. There is also a similar product construction, in fact, but it is not used in this paper.

A notation for pointed cpos may be described by

$$E ::= \mathsf{lift}\,D \mid \mathsf{cf}(D,E) \mid \ldots,$$

where D is a cpo (see above) and E is a pointed cpo. The continuous function space yields a pointed cpo if just the range is pointed. The discrete construction does not yield a pointed cpo, except if its set argument is a singleton set. The sum construction never yields a pointed cpo. Extensions are possible as above.

The syntactic notations allow us to use a term like

$$\mathsf{cf}(\mathsf{cf}(\mathsf{Nat},\mathsf{Nat}),\mathsf{cf}(\mathsf{Nat},\mathsf{lift}\,\mathsf{Nat}))$$

without proving it is a cpo or a pointed cpo. This is done automatically by proof functions.

3.3. Notation for Typable Terms

A term is (rather informally) called *cpo-typable* or just *typable* if it can be proved to be an element of some cpo. An interface and a proof function called the type checker implement a notation for typable terms. The interface provides a more convenient external syntax for the user than the internal one which is used by the type checker. It consists of a parser and a pretty-printer which extend the built-in parser and pretty-printer of HOL. The parser calculates and inserts certain cpo parameters on function constructors like $\mathsf{Fix}\,E$ which are parameterized by the (free cpo variables of the) domains on which they work, and the pretty-printer removes these parameters. The interface makes terms a lot easier to read and write. A secondary purpose of the interface is to provide a nicer syntax for the dependent lambda abstraction.

It is illustrated in Figure 1 how the parser and pretty-printer interact with the type checker to implement the notation for typable terms. If a term fits within the notation, then the type checker can reconstruct the cpo of the term.

The notation for typable terms may be described by

$$t ::= x \mid c \mid \lambda x \in D.\ t \mid t_1\, t_2,$$

where x is a variable of a restricted λ-abstraction, c is a constant which has been declared to the system,

$$\begin{tikzcd} t \arrow[r, dashed] \arrow[d, "\text{parse}"'] & \vdash t \in D \\ \mathit{internal}(t) \arrow[r, "\text{type check}"'] & \vdash \mathit{internal}(t) \in D \arrow[u, "\text{pretty-print}"'] \end{tikzcd}$$

FIGURE 1. Implementation of notation for typable terms.

and t, t_1 and t_2 are typable terms. The restricted λ-abstraction is translated into the dependent lambda abstraction. The collection of built-in declared constants can be extended by providing theorems which state which cpos the constants belong to. In fact, any term can extend the notation in this way.

The built-in constants are Fix, Lift, and Ext (there are others which are not used in this paper):

$$\begin{aligned}
&\vdash \forall E.\ \mathsf{pcpo}\,E \Rightarrow \mathsf{Fix}\,E \in \mathsf{cf}(\mathsf{cf}(E,E),E)\\
&\vdash \forall D.\ \mathsf{cpo}\,D \Rightarrow \mathsf{Lift}\,D \in \mathsf{cf}(D,\mathsf{lift}\,D)\\
&\vdash \forall DE.\ \mathsf{cpo}\,D \Rightarrow \mathsf{pcpo}\,E \Rightarrow\\
&\qquad \mathsf{Ext}(D,E) \in \mathsf{cf}(\mathsf{cf}(D,E),\mathsf{cf}(\mathsf{lift}\,D,E)).
\end{aligned}$$

The constant Lift is defined as a determined version of the constant Lft, using the dependent lambda abstraction. It takes an element $d \in D$ and produces an element of the lifted cpo $\mathsf{lift}\,D$. The constant Ext is used for extending the domain cpo of a function to the lifted domain in a strict way:

$$\begin{aligned}
&[f \in \mathsf{cf}(D,E); x \in D]\\
&\vdash (\mathsf{Ext}(D,E)\,f\,\mathsf{Bt} = \mathsf{bot}_E) \wedge\\
&\quad (\mathsf{Ext}(D,E)\,f\,(\mathsf{Lift}\,D\,x) = f\,x).
\end{aligned}$$

The disadvantage of the function constructors is that they are parameterized by cpos; in order to be determined on their domains, they are defined using the dependent lambda abstraction. The parser supports the omission of these parameters in terms which fit within the notation. However, it only works if constructors are actually applied to terms of the notation; their cpos are used to calculate the parameters (Agerholm, 1994c).

Let us consider an example. The following term fits within the notation for typable terms:

$$\mathsf{Fix}(\lambda f \in \mathsf{cf}(D,\mathsf{lift}\,\mathsf{Nat}).\ \lambda d \in D.\ \mathsf{Ext}\,f\,(\mathsf{Lift}\,d)).$$

The parser inserts the parameters on the function constructors and generates the following internal syntax, where also the internal version of the dependent lambda abstraction is used:

```
Fix(cf(D, lift Nat)
(lambda
 (cf(D, lift Nat))
 (λf. lambda D(λd. Ext(D, lift Nat) f(Lift D d)))).
```

Assuming that the variable D is a cpo, then the type checker reconstructs the cpo of the term. The theorem it proves is pretty-printed as follows:

$[\mathsf{cpo}\,D]$
$\vdash \mathsf{Fix}(\lambda f \in \mathsf{cf}(D, \mathsf{lift\,Nat}).\ \lambda d \in D.\ \mathsf{Ext}\,f\,(\mathsf{Lift}\,d)) \in \mathsf{cf}(D, \mathsf{lift\,Nat})$.

The type checker uses the proof functions for cpos to prove this fact.

3.4. Universal Cpos

The underlying set of a cpo is represented by a subset of a HOL type. When this subset corresponds to the whole type, then the cpo is called a universal cpo and any element of that type is trivially an element of the cpo. Discrete universal cpos are terms like Nat which are defined using the discrete construction with the universal set UNIV. The continuous function space of discrete universal cpos is itself equal to a discrete universal cpo. Hence, we trivially, and automatically, obtain that operations like successor and addition are continuous on the cpo of natural numbers:

$$\vdash \mathsf{SUC} \in \mathsf{cf}(\mathsf{Nat}, \mathsf{Nat})$$
$$\vdash \$+ \in \mathsf{cf}(\mathsf{Nat}, \mathsf{cf}(\mathsf{Nat}, \mathsf{Nat})).$$

Of course, we also have $\vdash 0 \in \mathsf{Nat}$. These theorems can then be used to extend the notation for typable terms.

3.5. Notation for Inclusive Predicates

The *inclusive prover* implements syntactic checks for inclusiveness. It automatically proves that a predicate of the form $\mathsf{mkpred}_D(\lambda x.\ e[x])$ is an inclusive subset of D (mkpred makes the subset) provided D has finite chains (e.g. if it is a discrete or a lifted discrete cpo) or $e[x]$ fits within the following notation

$$e ::= b \mid t_1 = t_2 \mid \forall y \in E.\ e[y] \mid \ldots,$$

where b is a boolean term which does not contain x, t_1 and t_2 are continuous in the variable x in the sense that $\lambda x \in D.\ t_i[x]$ fits within the notation for typable terms (i = 1, 2), $e[y]$ is an inclusive subset of D in the variable x, and finally, E is a non-empty cpo. The implementation of the inclusive prover is based on the description of the checks in the LCF system (Paulson, 1987, pp. 199–200).

4. THE LCF SYSTEM

The LCF theorem prover (Gordon *et al.*, 1979; Paulson, 1987) is very similar to the HOL system; in fact, HOL is a direct descendant of LCF. It has a meta language ML (or Standard ML) in which the logic and theorem proving tools are implemented. Theorems are implemented by an abstract datatype for security and axioms and primitive rules of inference are constructors of this datatype. Derived inference rules are ML functions. The subgoal package allows proofs in a backwards fashion using tactics. Constants, axioms, theorems and so on are organized in hierarchies of theories.

4.1. Logic

The central difference between LCF and HOL lies in their logics. The HOL system supports a version of Church's higher order logic (simple type theory), which has a set theoretic semantics. The LCF system supports a version of Scott's Logic of Computable Functions, which is a first order logic of domain theory; it has a domain theoretic semantics. The LCF logic differs from higher order logic since it is first order and types denote pointed cpos rather than just sets (cpos may be viewed as sets with structure). Further, the function type denotes the continuous function space whereas HOL functions are total functions of set theory.

Domain theory constructs are provided in LCF through axioms and primitive rules of inference. For instance, a certain constant of the logic denotes the fixed point operator, due to an axiom which states it yields a fixed point and due to the primitive rule of fixed point induction which states it yields the least fixed point. There is no semantic definition of the fixed point operator. Similarly, the notion of inclusive predicate (for fixed point induction) is not defined in LCF. It is only present via an incomplete syntactic check performed by fixed point induction in ML. Paulson provides an example of an inclusive predicate which is not accepted for fixed point induction in LCF (Paulson, 1984a).

Though this paper shall not make essential use of the presence of semantic definitions in HOL-CPO, it is still an important point to make. HOL-CPO provides fixed point induction and a syntactic check for inclusiveness as in LCF; in fact, the implementation of this check is based on the description in (Paulson, 1987, pp. 199–200). But HOL-CPO allows more theorems to be proved since it provides access to the semantic definitions of the fixed point operator and inclusiveness. This is useful when induction is not feasible or the check fails. Furthermore, HOL-CPO is more trust-worthy since fixed point induction and the inclusiveness check are derived from the semantic definitions by proof in HOL. In contrast, the LCF check (and fixed point induction) is implemented directly in ML.

4.2. Extensions

There are different traditions of extending theories in LCF and HOL. In HOL, there is a sharp distinction between definitional and axiomatical extensions. LCF only supports axioms.

It is not always easy to know whether an LCF axiom is safe or not since this must be justified in domain theory (outside LCF). In particular, an axiom should not violate the continuity of functions; functions are assumed to be continuous since the function type denotes the continuous function space. Paulson shows how easy it is to go wrong (Paulson, 1987, pp. 116–117).

In HOL-CPO, terms are not assumed but proved to be continuous functions. Due to the notation for typable terms, implemented by the type checker, such proofs are usually automatic and do not impose a burden on the user.

5. NATURAL NUMBERS

In LCF, natural numbers are introduced as a recursive datatype where a constant 0 and a strict successor function *SUCC* are the constructors (Paulson, 1987, Section 10.1). New constant names for the constructors are declared, then axioms about the constants are postulated. The axioms specify the partial ordering on natural numbers and state strictness and definedness of the constructors. The exhaustion (or cases) axiom is also postulated. It states that any natural number either equals bottom, zero or the successor of some natural number. Distinctness of the constructors and the structural induction rule are then derived from these axioms and fixed point induction. The axiomatization is performed automatically by a few ML functions.

It is also easy to define a cpo of natural numbers in HOL, though the method is very different. Instead of introducing a new recursive cpo, we exploit the built-in natural numbers using the discrete construction. If the natural numbers were not built-in we would use the type definition package (Melham, 1989) to define a new recursive type first.

We already defined the cpo of natural numbers called Nat above (Section 3.2.). Using lifting, we obtain the pointed cpo lift Nat which corresponds to the recursive type of natural numbers in LCF. The zero element of lift Nat is Lift 0 and a strict successor is defined using the built-in successor SUC and function extension:

$$\vdash \mathsf{Suc} = \mathsf{Ext}(\lambda p \in \mathsf{Nat}.\ \mathsf{Lift}(\mathsf{SUC}\, p)).$$

The right-hand side fits within the notation for typable terms (we assume the extensions presented in Section 3.3.). Hence, the type checker can prove that Suc is a continuous function on lift Nat, and this fact can then be used to extend the type checker itself, by extending the notation for typable terms.

In LCF, a strict addition on the type of natural numbers is axiomatized using the eliminator functional:

$$\begin{aligned}&NAT\_WHEN\, x\, f\, \bot \equiv \bot\\ &NAT\_WHEN\, x\, f\, 0 \equiv x\\ &\forall m.\, m \not\equiv \bot \Rightarrow NAT\_WHEN\, x\, f\, (SUCC\, m) \equiv f\, m.\end{aligned}$$

This is useful for defining continuous functions on natural numbers by cases. Note that *NAT_WHEN* must assume the argument of the strict LCF successor is defined, otherwise there would be a conflict with the bottom case. A consequence of this is that most theorems stated about *NAT_WHEN*, and in turn addition, inherit this assumption. Such definedness assumptions make reasoning about strict functions in LCF tedious and difficult (Paulson, 1984b).

In HOL-CPO, the easiest approach is to define a function in the set theoretic HOL world of natural numbers first, and then extend this to a strict function using Ext. Since addition is a built-in operation, it is particularly easy to introduce a strict addition:

$$\vdash \mathsf{Add} = \mathsf{Ext}(\lambda p \in \mathsf{Nat}.\ \mathsf{Ext}(\lambda q \in \mathsf{Nat}.\ \mathsf{Lift}(p+q))).$$

Since the right-hand side fits within the notation for typable terms, the type checker automatically proves it is a continuous operation on the lifted natural numbers:

$$\vdash \mathsf{Add} \in \mathsf{cf}(\mathsf{lift\,Nat}, \mathsf{cf}(\mathsf{lift\,Nat}, \mathsf{lift\,Nat})).$$

From the axiom for addition a number of recursion equations matching the cases in definition of *NAT_WHEN* are derived by proof. These are important in proofs of properties of addition, which are conducted by natural number induction.

The LCF recursion equations for addition have been proved in HOL but a reduction theorem is more useful:

$$\begin{aligned}\vdash\ &(\forall n.\ \mathsf{Add\,Bt}\, n = \mathsf{Bt}) \wedge\\ &(\forall n.\ \mathsf{Add}\, n\, \mathsf{Bt} = \mathsf{Bt}) \wedge\\ &(\forall pq.\ \mathsf{Add}(\mathsf{Lift}\, p)(\mathsf{Lift}\, q) = \mathsf{Lift}(p+q)).\end{aligned}$$

It states that addition is strict in both arguments and behaves as the built-in addition on lifted arguments. Statements about Add can then be reduce to statements about the built-in addition.

For instance, the reduction theorem is used to prove the following two theorems stating that strict addition is associative and commutative:

$$\begin{aligned}&\vdash \forall kmn.\ \mathsf{Add}(\mathsf{Add}\, k\, m)n = \mathsf{Add}\, k(\mathsf{Add}\, m\, n)\\ &\vdash \forall mn.\ \mathsf{Add}\, m\, n = \mathsf{Add}\, n\, m.\end{aligned}$$

The proofs are almost exactly the same in HOL. After a case analysis on elements of the lifted cpo of natural numbers, which are equal to either Bt or Lift p for some $p \in$ Nat, the reduction theorem is applied to reduce the statements into properties of the built-in addition:

$$\begin{aligned}&\vdash \forall mnp.\ m + (n+p) = (m+n) + p\\ &\vdash \forall mn.\ m + n = n + m.\end{aligned}$$

Such reductions by cases could be automated easily. The set theoretic statements can be proved by induction on the natural numbers in HOL; though, in this particular example, the statements are built-in facts.

In LCF, the statements about strict addition are proved directly by structural induction. Compared to HOL induction proofs, such inductions have an extra case, dealing with the bottom element of the type of LCF natural numbers. Furthermore, treating strictness of addition is done in the induction proof, whereas this is a separate (and trivial) part of the proof in HOL-CPO.

6. LAZY SEQUENCES

In the previous section, we showed how to reason about finite-valued types like natural numbers in HOL-CPO, by mixing set and domain theoretic reasoning to advantage. In such applications, we eliminate most of the reasoning about bottom elements which is painful in LCF. In this section, we show how HOL-CPO extends HOL by supporting LCF reasoning about infinite-valued types like lazy sequences.

The LCF datatype of lazy sequences is introduced axiomatically like the natural numbers, using the same ML functions in fact. Developing a cpo of lazy sequences in HOL-CPO was difficult and time-consuming, though we could have axiomatized the cpo as in LCF.

A purely definitional development of a theory of lazy sequences is presented in (Agerholm, 1994c). It provides a constructor seq for pointed cpos of partial and infinite sequences of data. Hence, if D is a cpo then seq D is a pointed cpo.

The bottom sequence is called $\mathsf{Bt\_seq}_D$ and the lazy constructor function is called Cons_seq D. The latter is a continuous function and can therefore be used to extend the notation of typable terms. They satisfy the following cases theorem:

$$\begin{array}{l} \vdash \forall D s. \\ \quad s \in \mathsf{seq}\, D = \\ \quad (s = \mathsf{Bt\_seq}_D) \lor \\ \quad (\exists x s'.\ x \in D \land s' \in \mathsf{seq}\, D \land (s = \mathsf{Cons\_seq}\, x\, s')). \end{array}$$

Furthermore, they are distinct and Cons_seq is one-one. A theorem for "structural induction" on lazy sequences has also been derived, from fixed point induction as in (Paulson, 1984a). Structural induction is used to show an inclusive property holds of all partial (finite) sequences, and the inclusiveness ensures that it holds also of the infinite sequences.

There is an eliminator functional called Seq_when which can be used to write continuous functions on sequences by cases. It is itself continuous and extends the notation of typable terms. The following reduction theorem specifies the eliminator:

$$\begin{array}{l} [x \in D; s \in \mathsf{seq}\, D; h \in \mathsf{cf}(D, \mathsf{cf}(\mathsf{seq}\, D, E))] \\ \vdash (\mathsf{Seq\_when}\, h\, \mathsf{Bt\_seq}_D = \mathsf{bot}_E) \land \\ \quad (\mathsf{Seq\_when}\, h (\mathsf{Cons\_seq}\, x\, s) = h\, x\, s). \end{array}$$

Definitions and theorems of this theory of lazy sequences are presented in (Agerholm, 1994c).

A mapping functional for lazy sequences can be defined using the eliminator and the fixed point operator:

$$\begin{array}{l} \vdash \forall D E. \\ \quad \mathsf{Maps}(D, E) = \\ \quad \mathsf{Fix} \\ \quad (\lambda g \in \mathsf{cf}(\mathsf{cf}(D, E), \mathsf{cf}(\mathsf{seq}\, D, \mathsf{seq}\, E)). \\ \quad\quad \lambda f \in \mathsf{cf}(D, E). \\ \quad\quad\quad \lambda s \in \mathsf{seq}\, D. \\ \quad\quad\quad\quad \mathsf{Seq\_when} \\ \quad\quad\quad\quad (\lambda x \in D.\ \lambda t \in \mathsf{seq}\, D.\ \mathsf{Cons\_seq}(f\, x)(g\, f\, t))s). \end{array}$$

The type checker automatically proves that Maps is continuous:

$$\begin{array}{l} \vdash \forall D E. \\ \quad \mathsf{cpo}\, D \Rightarrow \mathsf{cpo}\, E \Rightarrow \\ \quad \mathsf{Maps}(D, E) \in \mathsf{cf}(\mathsf{cf}(D, E), \mathsf{cf}(\mathsf{seq}\, D, \mathsf{seq}\, E)). \end{array}$$

This theorem can be used to extend the notation of typable terms. Using the reduction theorem for Seq_when and the fact that Fix yields a fixed point of a continuous function on a pointed cpo, we can prove the following reduction equations for Maps easily:

$$\begin{array}{l} [x \in D; s \in \mathsf{seq}\, D; f \in \mathsf{cf}(D, E)] \\ \vdash (\mathsf{Maps}\, f\, \mathsf{Bt\_seq}_D = \mathsf{Bt\_seq}_E) \land \\ \quad (\mathsf{Maps}\, f (\mathsf{Cons\_seq}\, x\, s) = \mathsf{Cons\_seq}(f\, x)(\mathsf{Maps}\, f\, s)). \end{array}$$

A tactic which takes such reduction theorems as arguments can be used to reduce occurrences of Maps and other function constructors, using the type checker to prove the assumptions automatically.

We can prove that the mapping functional preserves functional composition:

$$\begin{array}{l} [f \in \mathsf{cf}(D_2, D_3); g \in \mathsf{cf}(D_1, D_2)] \\ \vdash \mathsf{Maps}(\mathsf{Comp}(f, g)) = \mathsf{Comp}(\mathsf{Maps}\, f, \mathsf{Maps}\, g). \end{array}$$

The constant Comp is defined as a determined version of the built-in functional composition. It is continuous as expected, which must be proved manually, and we assume it is in the notation of typable terms.

The proof of the above equality is conducted by observing that the two continuous functions are equal iff they are equal for all sequences of values in D_1, i.e. iff the following term holds:

$$\begin{array}{l} \forall s.\ s \in \mathsf{seq}\, D_1 \Rightarrow \\ \quad (\mathsf{Maps}(\mathsf{Comp}(f, g))s = \mathsf{Comp}(\mathsf{Maps}\, f, \mathsf{Maps}\, g)s). \end{array}$$

Then we employ an induction tactic based on the structural induction theorem for lazy sequences. This tactic uses the various provers behind the scenes, in particular the inclusive prover to prove the statement is inclusive (seq D_1 is non-empty since it is a pointed cpo). The proof is finished off using the reduction tactic with theorems for Maps and Comp .

Finally, we present a functional Seqof which given a continuous function f and any starting point value x generates an infinite sequence of the form

$$\mathsf{Cons\_seq}\, x (\mathsf{Cons\_seq}(f\, x)(\mathsf{Cons\_seq}(f (f\, x)) \ldots)),$$

or written in a more readable way $[x; f\, x; f(f\, x); \ldots]$. The function Seqof is defined as a fixed point as follows:

$$\begin{array}{l} \vdash \forall D. \\ \quad \mathsf{Seqof} = \\ \quad \mathsf{Fix} \\ \quad (\lambda g \in \mathsf{cf}(\mathsf{cf}(D, D), \mathsf{cf}(D, \mathsf{seq}\, D)). \\ \quad\quad \lambda f \in \mathsf{cf}(D, D).\ \lambda x \in D.\ \mathsf{Cons\_seq}\, x (g\, f (f\, x))). \end{array}$$

The type checker then proves

$$\vdash \forall D.\ \mathsf{cpo}\, D \Rightarrow \mathsf{Seqof} \in \mathsf{cf}(\mathsf{cf}(D, D), \mathsf{cf}(D, \mathsf{seq}\, D)),$$

which allows us to extend the notation of typable terms.

The following statement can be proved to relate Maps and Seqof:

$$\begin{array}{l} [f \in \mathsf{cf}(D, D); \mathsf{cpo}\, D] \\ \vdash \forall x.\ x \in D \Rightarrow (\mathsf{Seqof}\, f (f\, x) = \mathsf{Maps}\, f (\mathsf{Seqof}\, f\, x)). \end{array}$$

Informally, the two sequences are equal since they are both equal to a term corresponding to $[f\,x; f(f\,x); \ldots]$. The proof of the theorem is conducted by fixed point induction on both occurrences of Seqof; inclusiveness and other side conditions are proved behind the scenes (a case analysis is made on whether or not D is empty).

The recursive functions of this section could not have been defined easily in HOL, which has no support for non-primitive recursive functions. However, using domain theory in HOL-CPO, we were able to define and reason about general recursive partial functions as in LCF.

7. THE UNIFICATION ALGORITHM

The problem of finding a common instance of two expressions is called *unification*. The unification algorithm generates a substitution to yield this instance, and returns a failure if a common instance does not exist. Expressions, also called *terms*, can be constants, variables and applications of one expression to another:

$$term ::= \mathsf{Const}\ name \mid \mathsf{Var}\ name \mid \mathsf{Comb}\ term\ term.$$

Variables are regarded as empty slots for which expressions can be substituted. A substitution is a set of pairs of variables and expressions that specifies which expressions should be substituted for which variables in an expression.

Manna and Waldinger synthesized a unification algorithm by hand using their deductive tableau system (Manna *et al.*, 1981) and Paulson made an attempt to translate their proof of correctness to LCF (Paulson, 1985). Paulson did not deduce the algorithm from the proof as Manna and Waldinger did; he stated the algorithm first and then proved it was correct.

A version of Paulson's proof has been conducted in HOL-CPO. In this section we shall not go into the details of this proof but mainly discuss a few points made by Paulson on the LCF proof. The details are presented in (Agerholm, 1994c).

Although this example is considerably larger than the examples above it does not require deeper insights into domain theory. In fact, domain theory is used very little and only in the last stages of the proof. But the formalization is exploited in an essential way. The unification algorithm is recursive but not also primitive recursive. Therefore, HOL does not support its definition. However, it can be defined as a fixed point in HOL-CPO easily.

Once we have proved that the unification algorithm defined in domain theory always terminates—this proof is conducted by well-founded induction—we can define a pure set theoretic HOL function. This approach provides a method for defining non-primitive recursive functions by well-founded induction in HOL. The method could (probably) be automated in such a way that no domain theory constructs appear to the user.

Paulson says that LCF does not provide an ideal logic for verifying the unification algorithm since it clutters up everything with the bottom element. For instance, the syntax type of terms and the type of constant and variable names must contain a bottom element, just like all other LCF types. Hence, definedness assertions of the form $t \not\equiv \bot$ occur everywhere because constructor functions for terms are only defined if their arguments are (strictness). To indicate the influence of this problem on the complexity of statements and proofs, we show the LCF definitional properties for substitution:

$$\begin{array}{l}
\bot\, SUBST\, s \equiv \bot \\
\forall c.\, c \not\equiv \bot \Rightarrow (CONST\, c)\, SUBST\, s \equiv CONST\, c \\
\forall v.\ v \not\equiv \bot \Rightarrow \\
\qquad (VAR\, v)\, SUBST\, s \equiv ASSOC\, (VAR\, v)\, v\, s \\
\forall t_1 t_2.\ t_1 \not\equiv \bot \Rightarrow t_2 \not\equiv \bot \Rightarrow \\
\qquad (COMB\, t_1\, t_2)\, SUBST\, s \equiv \\
\qquad COMB(t_1\, SUBST\, s)(t_2\, SUBST\, s).
\end{array}$$

In HOL, substitution is introduced by a primitive recursive definition:

$$\begin{array}{l}
\vdash (\forall cs.\ (\mathsf{Const}\, c)\, \mathsf{subst}\, s = \mathsf{Const}\, c) \wedge \\
\quad (\forall vs.\ (\mathsf{Var}\, v)\, \mathsf{subst}\, s = \mathsf{assoc}(\mathsf{Var}\, v) v\, s) \wedge \\
\quad (\forall t_1 t_2 s. \\
\qquad (\mathsf{Comb}\, t_1\, t_2)\, \mathsf{subst}\, s = \\
\qquad \mathsf{Comb}(t_1\, \mathsf{subst}\, s)(t_2\, \mathsf{subst}\, s)).
\end{array}$$

Note that this is pure HOL; it is not necessary to use domain theory to define a type (or cpo) of terms nor to define substitution. Further, we avoid LCF's explicit statements of totality for functions which are obviously total, such as $SUBST$

$$\forall t\, s.\, t \not\equiv \bot \Rightarrow s \not\equiv \bot \Rightarrow t\, SUBST\, s \not\equiv \bot,$$

since HOL functions are always total.

All functions on terms used in the proof, except unification itself, can be defined by primitive recursion like subst above. Hence, we can do the set theoretic developments first and then turn to domain theory later when it becomes necessary. In this way, we eliminate essentially all reasoning about bottom which makes the LCF proof painful and messy. Moreover, it is easy to shift to domain theory, by exploiting the discrete construction on cpos as in the natural number example above.

The unification algorithm is defined as a fixed point of a certain functional. A number of recursion equations, stated without the use of Fix, are then derived from the fixed point property. The type checker automatically proves continuity:

$$\vdash \mathsf{unify} \in \mathsf{cf}(\mathsf{Term}, \mathsf{cf}(\mathsf{Term}, \mathsf{lift}\ \mathsf{Attempt})).$$

The cpo of terms Term is just the discrete universal cpo of all HOL terms of type *term*, which can be introduced by the above specification. The cpo of attempts is the sum cpo of two discrete universal cpo with underlying type *one* and a discrete universal cpo with underlying type $(name \times term)list$, corresponding to the type of substitutions. The first component of the sum can be interpreted as failure and the second as success. The correctness and totality of unify is stated as the theorem:

$$\vdash \forall t u.\ \exists a.\ (\mathsf{unify}\, t\, u = \mathsf{Lift}\, a) \wedge \mathsf{best\_unify\_try}(a, t, u).$$

The first conjunct states unify is total and the second states that it yields the best (most general and idempotent) unifier in a certain sense if a unifier exists, otherwise it yields a failure. The predicate best_unify_try is defined in pure HOL; no domain theory is used.

The unification algorithm is recursive on terms but it is not primitive recursive. In order to unify two combinations $\mathsf{Comb}\, t_1\, t_2$ and $\mathsf{Comb}\, u_1\, u_2$ the algorithm first attempts to unify t_1 and u_1 and if it succeeds with the substitution s as a result it attempts to unify $t_2\, \mathsf{subst}\, s$ and $u_2\, \mathsf{subst}\, s$. The latter two terms may not be subterms of the original combinations and therefore a primitive recursive definition does not work. However, when this is the case then the total number of variables in the terms are reduced. This argument induces a well-founded relation which can be used to prove termination. It is a kind of lexicographic combination of a proper subset ordering on sets of variables and an 'occurs-in' ordering.

A theory of well-founded induction has been developed in HOL (Agerholm, 1992) but never in LCF; it is not clear whether this is possible or not. Therefore, well-founded induction is translated to two structural inductions in LCF, one on natural numbers and one on terms. This makes certain statements more complicated than necessary and makes the proof less elegant as well.

Since we have proved unify is total, it defines a HOL function:

$$\vdash \forall t u.\ \mathsf{Unify}\, t\, u = (\epsilon a.\ \mathsf{unify}\, t\, u = \mathsf{Lift}\, a)$$

Here, the choice operator (see Section 2.) is used to choose the attempt that we know exists by the correctness theorem above. From this definition, we can then prove the recursion equations which states how Unify behaves on various kinds of term arguments. These define Unify without any domain theory. Furthermore, we can prove Unify yields a best unifier for terms of type *term*:

$$\vdash \forall t u.\ \mathsf{best\_unify\_try}(\mathsf{Unify}\, t\, u, t, u).$$

This approach supports non-primitive recursive definitions by well-founded induction in HOL. Probably, the domain theory constructs could be hidden completely from the user, by an automated tool which does the domain theoretic reasoning behind the scenes and just need theorems from the user for the well-founded induction.

8. CONCLUSION

A contribution of this work is a comparison of two systems supporting domain theoretic reasoning, namely, LCF and HOL-CPO. Using examples we show how HOL-CPO supports the best of the domain theoretic LCF and the set theoretic HOL worlds, by allowing set and domain theoretic reasoning to be mixed.

We presented the mechanization of a number of examples in HOL-CPO which have already been done in LCF by Paulson. The natural number example illustrates how we can mix set and domain theoretic reasoning and thereby ease reasoning about finite-valued LCF types and strict functions. The example on lazy sequences presents fixed point definitions of recursive nonterminating functions and illustrates that we can conduct LCF proofs by fixed point induction and structural induction on infinite-valued recursive datatype cpos. This kind of reasoning is not easy in 'pure' HOL, since it is not directly supported.

The unification example shows that we can eliminate essentially all reasoning about the bottom element that infests the proof in LCF. In HOL, most of the verification is conducted in the set theoretic HOL world, and it is only at a very late stage of the proof that domain theory constructs are introduced, in order to give a fixed point definition of the unification algorithm which is not primitive recursive and therefore cannot be defined easily in HOL. Moreover, domain theory is only introduced temporarily, since once we have proved the algorithm always terminates we can define a total unification function in HOL and forget about domain theory. This method of defining functions via domain theory and a proof of termination could (probably) be automated to extend the present tools for recursive function definitions in HOL.

Further, the example shows that we are not restricted to use fixed point induction for reasoning about recursive functions. The proof of termination of the unification algorithm is conducted by well-founded induction, but the LCF proof uses two nested structural inductions derived from fixed point induction to simulate well-founded induction. This makes the proof more complicated, and less elegant too.

Some disadvantages of the embedding of domain theory in HOL have also been mentioned. As a negative consequence of being an embedding rather than a direct implementation of a logic (as in LCF), side conditions on domain theoretic properties appear everywhere in HOL-CPO. However, in most cases these are proved automatically by proof functions implementing syntactic notations of cpos and typable terms.

One main problem is that it is time-consuming and not at all straightforward to introduce new recursive datatype cpos like lazy sequences. Recursive datatypes are just axiomatized in LCF, but axiomatic extensions are against the tradition of a purely definitional approach in the HOL community. Further, a standard method like the inverse limit construction for solving recursive domain equations (Smyth *et al.*, 1982) is not easily formalized in HOL, since it requires the presence of dependent products over different types, which cannot be defined in HOL (this requires an encoding in a 'large' type). However, as demonstrated in (Agerholm, 1994b), its formalization is straightforward in HOL-ST, an experimental extension of HOL with a ZF-like set theory (Gordon, 1994).

Another problem is that due to the need for the dependent lambda abstraction, function constructors become parameterized by (the free variables of) the domains on which they work. This inconvenience is handled by an interface in most cases but the problem also affects the convenience of proofs, since arguments of functions must be proved to be in the right cpos. This is automated in most cases by the type checker, which implements the notation of typable terms.

One may compare the problems in LCF due to bottom to the problems in HOL-CPO due to the parameters on some function constructors. An interface could also be implemented in LCF to hide bottom in many cases, but it would always appear in proofs. Quite often, we avoid parameters in HOL-CPO because we work in higher order logic or with concrete cpos like natural numbers.

HOL-CPO is a semantic embedding of domain theory in a powerful theorem prover. It was an important goal of this embedding to preserve a direct correspondence between elements of domains and elements of HOL types. This allows us to exploit the types and tools of HOL directly and hence, to benefit from mixing domain and set theoretic reasoning as discussed above. A semantic embedding does not always have this property. The formalization of Pω (Petersen, 1993) builds a separate Pω world inside HOL so there is no direct relationship between, for instance, natural numbers in the Pω model and in the HOL system. The same thing would be true about a formalization of information systems (Winskel, 1993), if it was done. On the other hand, formalizations of Pω and information systems would allow recursive domain equations to be solved fairly easily using a fixed point operator.

ACKNOWLEDGEMENTS

This work was supported in part by the DART project funded by the Danish Research Council and in part by BRICS (Basic Research in Computer Science), Centre of the Danish National Research Foundation. Thanks to Flemming Andersen, Kim Dam Petersen and Glynn Winskel for discussions concerning this work. Glynn made comments on a draft of this paper. I am grateful to Larry Paulson for digging up the LCF proof of correctness of the unification algorithm.

REFERENCES

Agerholm, S. (1992) Mechanizing Program Verification in HOL. In: Archer, M., Joyce, J. J., Levitt, K. N., et al (Eds), *Proceedings of the 1991 International Workshop on the HOL Theorem Proving System and Its Applications*, pp. 208–222, IEEE Computer Society Press. For a full report see: Agerholm, S. (1992) Mechanizing Program Verification in HOL. M.Sc. Thesis, Report IR-111, Aarhus University, Computer Science Department.

Agerholm, S. (1994a) Domain Theory in HOL. In: Joyce, J. J. and Seger, C. H. (Eds.), *Proceedings of the 6th International Workshop on Higher Order Logic Theorem Proving and its Applications*, pp. 295–309, LNCS 780, Springer-Verlag.

Agerholm, S. (1994b) Formalising a Model of the λ-calculus in HOL-ST. Technical Report no. 354, University of Cambridge Computer Laboratory.

Agerholm, S. (1994c) *A HOL Basis for Reasoning about Functional Programs*. PhD thesis, BRICS Report RS-94-44, Department of Computer Science, University of Aarhus.

Gordon, M. J. C. (1994) Merging HOL with Set Theory: preliminary experiments. Technical Report no. 353, University of Cambridge Computer Laboratory.

Gordon, M. J. C. and Melham, T. F. (1993) *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press.

Gordon, M. J. C., Milner, R. and Wadsworth, C.P. (1979) *Edinburgh LCF: A Mechanised Logic of Computation*. LNCS 78, Springer-Verlag.

Manna, Z. and Waldinger, R. (1981) Deductive Synthesis of the Unification Algorithm. *Science of Computer Programming*, Vol. 1, pp. 5–48.

Melham, T. F. (1989) Automating Recursive Type Definitions in Higher Order Logic. In: Birtwistle, G. and Subrahmanyam, P. A. (Eds.), *Current Trends in Hardware Verification and Theorem Proving*, pp. 341–386, Springer-Verlag.

Melham, T. F. (1992) The HOL pred_sets Library. University of Cambridge, Computer Laboratory. (Appears in *The HOL System: Libraries*, documentation distributed with the HOL system.)

Paulson, L. C. (1984a) Deriving Structural Induction in LCF. In: Kahn, G. MacQueen, D. B., and Plotkin, G. (Eds), *Semantics of Data Types*, pp. 197–214, LNCS 173, Springer-Verlag.

Paulson, L. C. (1984b) Lessons Learned from LCF. Technical Report no. 54, University of Cambridge Computer Laboratory.

Paulson, L. C. (1985) Verifying the Unification Algorithm in LCF. *Science of Computer Programming*, Vol. 5, pp. 143–169.

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

Petersen, K. D. (1994) Graph Model of LAMBDA in Higher Order Logic. In: Joyce, J. J. and Seger, C. H. (Eds.), *Proceedings of the 6th International Workshop on Higher Order Logic Theorem Proving and its Applications*, pp. 16–28, LNCS 780, Springer-Verlag.

Smyth, M. and Plotkin, G. D. (1982) The Category-theoretic Solution of Recursive Domain Equations. *SIAM Journal of Computing*, Vol. 11, pp. 761–783.

Winskel, G. (1993) *The Formal Semantics of Programming Languages*. The MIT Press.