BRICS

Basic Research in Computer Science

BRICS RS-94-18 S. Agerholm: LCF Examples in HOL

LCF Examples in HOL

Sten Agerholm

BRICS Report Series RS-94-18

ISSN 0909-0878 June 1994

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

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

Telephone: +45 8942 3360
Telefax: +45 8942 3255
Internet: BRICS@daimi.aau.dk

LCF Examples in HOL

Sten Agerholm
BRICS[*], Computer Science Dept.,
Aarhus University, DK-8000 Aarhus C., Denmark

June 19, 1994

Abstract

The LCF system provides a logic of fixed point theory and is useful to reason about nontermination, recursive definitions and infinite-valued types such as lazy lists. Because of continual presence of bottom elements, it is clumsy for reasoning about finite-valued types and strict functions. The HOL system provides set theory and supports reasoning about finite-valued types and total functions well. In this paper a number of examples are used to demonstrate that an extension of HOL with domain theory combines the benefits of both systems. The examples illustrate reasoning about infinite values and nonterminating functions and show how domain and set theoretic reasoning can be mixed to advantage. An example presents a proof of correctness of a recursive unification algorithm using well-founded induction.

1 Introduction

The LCF system [GMW79, Pa87] is a theorem prover based on a version of Scott's Logic of Computable Functions (a first order logic of domain theory). It provides the concepts and techniques of fixed point theory to reason about nontermination and arbitrary recursive (computable) functions. For instance, it has been successfully applied to reason about infinite data structures and lazy evaluation [Pa84b]. On the other hand, the HOL system [GM93] supports set theoretic reasoning. It has no inbuilt notion of nontermination, all functions are total, and only primitive recursive definitions are supported. It has mainly been used for reasoning about finite data structures and terminating primitive recursive functions.

In a way, extending HOL with domain theory as described in [Ag93, Ag94] corresponds to embedding the logic of the LCF system within HOL-this extension is called HOL-CPO in this paper. Thus, any proof conducted in LCF can be conducted in HOL-CPO as well, and axioms of LCF theories can be introduced as definitions, or derived from definitions, provided of course they are consistent extensions of LCF. This correspondence breaks for difficult recursive domains with infinite values. It is not easy to define such domains in HOL and LCF could just axiomatize the domains; still this has its theoretical difficulties in general, but has been automated in certain cases [Pa84a].

However, HOL-CPO is not just another LCF system. Ignoring the problems with recursive domains, we claim it is more powerful and usable than LCF since (1) it inherits

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

the underlying 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 rather than *implementing* logic. One advantage of (1) is that we can exploit the rich collection of built-in types, theorems and tools provided with the HOL system. LCF has almost nothing like that. Another advantage is that we become able to mix domain and set theoretic reasoning in HOL such that reasoning about bottom can be deferred until the late stages of a proof. To support this point, experience shows that the continual fiddling with bottom in LCF is very annoying. Its presence in all types makes LCF clumsy for reasoning about finite-valued types and strict functions [Pa85, Pa84b].

In contrast to (2), domain theory is only present in the logic of LCF through axioms and primitive rules of inference. Therefore fixed point induction is the only way to reason about recursive definitions. Testing that a predicate admits fixed point induction can only be performed in ML by an incomplete syntactic check. By exploiting the semantic definitions of these concepts in domain theory, HOL-CPO does not impose such limitations. Fixed point induction can be derived as a theorem and syntactic checks for admissibility, called inclusiveness, can be implemented, just as in LCF. But using other techniques for recursion or reasoning directly about fixed points allows more theorems to be proved than with just fixed point induction. Inclusive predicates not accepted by the syntactic checks can be proved to be inclusive from the semantic definition.

In this paper we present a number of examples 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 [Pa87] and the third example is based on Paulson's version of a correctness proof of a unification algorithm by Manna and Waldinger [MW81, Pa85]. The unification algorithm is defined as a fixed point and proved total afterwards. Termination is non-trivial and proved by well-founded induction [Ag91].

Before we turn our attention to the examples we give an overview of the formalization of domain theory in section 2 and describe the LCF system in section 3. In section 4 we introduce a cpo of natural numbers and present a few theorems about addition. In section 5 a mapping function for lazy sequences and a generator for infinite sequences are introduced. The correctness proof of the unification algorithm is discussed in section 6. Finally, the conclusions are summarized in section 7.

2 HOL-CPO

In this section we provide an overview of the formalization of domain theory and some of the associated tools [Ag93, Ag94]. This extension of HOL, called HOL-CPO, constitutes an integrated system where domain theoretic concepts 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 subset of domain theory which is used. Therefore the presentation below shall be very brief. More details can be sought in [Ag94], or in [Wi93] on which the formalization is based.

2.1 Basic Concepts

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

```
cpo:(*->bool)#(*->*->bool)->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 (unique) *least upper bound* (lub) for all non-decreasing chains "X:num->*" of elements in A (X is a chain if "R(X n)(X(n+1))" holds for all n).

The underlying relation of a cpo D is obtained by writing "rel D". If x and y are elements of D , written as "x ins D" and "y ins D", then "rel D x y" can be read as 'x approximates y' or 'x is less defined than (or equals) y'.

Note that we do not require cpos to have a *least defined* element, also called a *bottom* element, w.r.t. the underlying ordering relation. Cpos which have a bottom are called *pointed* cpos and satisfy the HOL predicate pcpo (same type as cpo). If E is a pointed cpo then the term "bottom E" equals the bottom element of E . In the following, cpos are usually cpos *without* bottom unless we say explicitly that a cpo is pointed.

A *continuous* function from a cpo D1 to a cpo D2 is a HOL function "f:*1->*2" such that the term "cont f(D1,D2)" is true. It must be monotonic w.r.t. the underlying relations and preserve lubs of chains in D1 in the sense that f applied to the lub of a chain X in D1 is equal to the lub of the chain "f(X n)" in D2 . The results of applying f constitute a chain due to monotonicity and therefore have a lub in D2 . In addition, f must be *determined* by its action on elements of the domain cpo D1 . This means that on elements outside D1 it should always return a fixed arbitrary value called ARB (a predefined HOL constant). 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 the main disadvantage of the formalization. In particular, functions must be written using a dependent lambda abstraction "lambda D f" to ensure they are determined. Therefore, many functions become parameterized by cpo variables (corresponding to the free cpo variables of the right-hand side of their definition).

The conditions on cpos and continuous functions ensure the existence of a *fixed point operator*, called FixI (the 'I' is explained later), which is useful to define arbitrary recursive functions and other infinite values. Applied to a continuous function f on a pointed cpo E , it yields a fixed point: |- f(FixI E f) = FixI E f , and in fact the *least* fixed point: |- !x. x ins E ==> (f x = x) ==> rel E(FixI E f)x . The term "FixI E f" equals the least upper bound of the non-decreasing chain $\bot \sqsubseteq$ f($\bot$) $\sqsubseteq$ f(f($\bot$))... where $\bot$ stands for "bottom E" and $\sqsubseteq$ stands for "rel E".

The proof principle of fixed point induction has been derived as a theorem from the definition of the fixed point operator. It can be used to prove properties of fixed points stated as *inclusive* (or *admissible*) predicates. A predicate is inclusive if it contains lubs of chains of elements in the predicate. Fixed point induction says that "P(FixI E f)" follows from "P(bottom E)" and "!x. P x ==> P(f x)", assuming a pointed cpo E , a continuous function f from E to E and an inclusive predicate P on E . There are a few syntactic-based proof functions to prove these semantic conditions: the cpo prover, the type checker and the inclusive prover (based on the LCF check in [Pa87] on page 199–200), respectively. They only work in certain cases (see below).

2.2 Constructions

There are various standard ways of constructing cpos and continuous functions which allow proofs to be automated in HOL.

The *discrete* construction associates the discrete ordering (identity) with a set and it is therefore 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->bool)"` using the universal set `UNIV` (a predicate which is always true, here corresponding to the set of all elements of `":num"`). A construction called *lifting* can then be used to extend the cpo with a bottom element as follows `"lift(discrete UNIV)"`. The bottom element of a lifted cpo `"lift D"` is written as `Bt` and all other elements are written as `"Lft d"` for some `d` in `D`. It can be proved that `"bottom(lift D)"` is equal to `Bt`. The constants `Bt` and `Lft` are the constructors of a new datatype in HOL which associates a new element with a type. It is the underlying relation of the lifting construction which makes `Bt` into a bottom element of `"lift D"`.

There is also a construction for the cpo of continuous functions, relating functions by the pointwise ordering relation. Assuming two cpos `D1` and `D2` this construction is written as `"cf(D1,D2)"`. Note that the two statements `"f ins (cf(D1,D2))"` and `"cont f(D1,D2)"` are equivalent. Finally, we provide a product construction and sum construction written as `"prod(D1,D2)"` and `"sum(D1,D2)"`, respectively.

A proof function called the *cpo prover* automatically proves any term written using the constructors is a cpo. There is a similar function for pointed cpos.

The constructions on continuous functions include the well-known projection and injection functions associated with the product and sum cpos respectively, and functional composition and currying as well. We also consider the fixed point operator to be a constructor.

In addition, there are two useful constructors associated with the lifting construction on cpos. A determined version of the constant `Lft`, called `LiftI`, takes an element `d` of a cpo `D` and lifts it to an element of the lifted cpo `"lift D"`. A construction called *function extension* can be used to extend the domain of a function to the lifted domain in a strict way. It works as follows:

```
|- (ExtI(D,E) f Bt = bottom E) /\
   (!x. x ins D ==> (ExtI(D,E) f (LiftI D x) = f x))
```

where `E` is pointed cpo and `f` is a continuous function from `D` to `E`.

It is also possible to write continuous functions using the dependent lambda abstraction `"lambda D(\x. e[x])"` where `e[x]` must be written using only continuous constructions and variables and constants in appropriate cpos. To prove a function, or more generally any term, is in some cpo (e.g. the continuous function space) a proof function called the *type checker* can be used, provided the term fits within an informal notation. The constructions above and lambda abstraction are part of the notation which can be extended interactively with any terms in cpos (see below). Function application is also part of the notation.

Further, any function between discrete universal cpos as the cpo of natural numbers is trivially continuous. Hence, the cpo of continuous functions between such cpos is itself a discrete universal cpo.

2.3 Adding New Constructions

The collection of constructors for cpos and continuous functions can at any time be extended with user-defined constructors. An ML function is provided to define a new cpo constructor in terms of existing constructors, and similarly new function constructors can be introduced. We make a distinction between new constants that are elements of some cpo and new function constructors of some cpo. The latter are parameterized by cpo variables, like the constructors above. A new constant *to the system* can be any proper left-hand side of a HOL definition which can be proved to belong to some cpo. All of this has been automated such that there are only a few proof functions to use which prove the necessary cpo and membership facts behind the scenes.

However, the constructions do not provide all cpos and continuous functions that we might want. In particular, recursive domains and their associated function constructions must be introduced manually. A fairly tough development gave us lazy sequences and lazy lists using HOL lists `":(*)list"` and functions of the form `":num->*"` to represent infinite values. These developments are described in [Ag94] which also provides some ideas on how to introduce such cpos more generally using infinite labelled trees. It might be possible to automate these ideas. The cpo of lazy sequences and its associated constructor and eliminator functions are used in section 5.

2.4 Interface

The cpo parameters on function constructions quite quickly become a pain. They make terms difficult to read and write. Fortunately an extension of the built-in HOL parser and pretty-printer can hide the annoying extra information in most cases. This provides two levels of syntax, the internal level of syntax where all parameters occur and the external (interface) level of syntax where the parameters are ignored. Hence, `FixI`, `LiftI` and `ExtI` above are part of the internal level syntax. The last letter '`I`' on (internal) names is used to distinguish the constants of the two levels. At the external interface level the terms `"FixI E"`, `"LiftI D"` and `"ExtI(D,E)"` are written simply as `Fix`, `Lift` and `Ext`, respectively. New function constructors are also introduced in two versions. Furthermore, the interface provides a nicer syntax for the dependent lambda abstraction. The term `"\x::Dom D. e[x]"` can be used for `"lambda D(\x. e[x])"`.

3 The LCF System

The LCF system is very similar to the HOL system (or vice versa, since 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 inference rules 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. The main properties of LCF may be summarized as follows:

- LCF supports a first order logic of domain theory.
- The use of LCF to reason about recursive definitions (fixed points) is restricted since only fixed point induction can be used. Besides, fixed point induction is based on an incomplete syntactic check of inclusiveness.
- Extending theories in LCF is done by an axiomatic approach and is therefore unsafe. Checking whether an axiom is safe is difficult since it must be done in domain theory (outside LCF).

Each of these points are discussed below.

The central difference between LCF and HOL lies in their logics. The logic of the HOL system is an implementation of a version of Church's higher order logic. The logic of the LCF system is an implementation of a version of Scott's Logic of Computable Functions, usually abbreviated LCF. In order to be able to distinguish the logic and the system the logic was renamed to PPλ, an acronym of Polymorphic Predicate λ-calculus. PPλ is a first order logic of domain theory, it has a domain theoretic semantics. It differs from higher order logic since it is a first order logic and types denote pointed cpos rather than just sets (cpos can be seen as sets with structure). The function type denotes the cpo of continuous functions whereas HOL functions are total functions of set theory.

Fixed point theory is provided in LCF through axioms and primitive rules of inference. 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. In LCF there is no domain theoretic definition of the fixed point operator. Therefore, fixed point induction is the *only* way to reason about recursive definitions. However, structural induction for many datatypes can be derived from fixed point induction [Pa84a], but well-founded induction cannot. Admissibility of predicates for induction is not defined either; a syntactic check is performed by the rule of fixed point induction. This check is not complete and examples of inclusive predicates exist that are not accepted for fixed point induction in LCF. Paulson gives an example in [Pa84a].

There are quite different traditions of extending theories in LCF and HOL. In HOL there is a sharp distinction between purely definitional extensions and axiomatical extensions. Definitional extensions are conservative (or safe), i.e. they always preserve consistency of the logic. Stating a new axiom is not a conservative extension, it might introduce inconsistency. In LCF there is no such distinction between axioms and definitions. The only way to extend theories with new concepts is by introducing new axioms.

It is not always easy to know whether an LCF axiom is safe or not since this must be justified in domain theory. In particular, an axiom should not violate the continuity of a function. All functions are assumed to be continuous in PPλ since the function type denotes the cpo of continuous functions. Paulson shows how easy it is to go wrong in example 4.11 of his book [Pa87].

4 Natural Numbers

In this section we start the comparison of LCF and HOL-CPO. As a first simple example, we define a cpo of natural numbers and consider a few properties about addition: addition is total, associative and commutative.

In LCF natural numbers are introduced as a recursive datatype where a constant 0 and a strict successor function *SUCC* are the constructors. Names of constants for the type and for the constructor functions are declared and then axioms about the new 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 there are three possible kinds of values of a natural number, namely bottom, zero and 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. This 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 and define `|- Nat = discrete(UNIV:num->bool)`. Using lifting `"lift Nat"`, we obtain the pointed cpo corresponding to the recursive type of natural numbers in LCF.

The zero element of `"lift Nat"` is `"Lift 0"` and a strict successor is obtained from the built-in successor `SUC` by function extension:

```
|- Suc = Ext(\nn :: Dom Nat. Lift(SUC nn))
|- Suc ins (cf(lift Nat,lift Nat)).
```

Note that `SUC` is trivially a continuous function from `Nat` to `Nat` since the term `"cf(D1,D2)"` is a discrete universal cpo when `D1` and `D2` are.

In LCF, addition is introduced by a recursion equation using an eliminator functional, called *NAT_WHEN*,

$$\begin{array}{l} \mathit{NAT\_WHEN}\; x\, f\, \bot \equiv \bot \\ \mathit{NAT\_WHEN}\; x\, f\, 0 \equiv x \\ \forall m.\, m \not\equiv \bot \Rightarrow \mathit{NAT\_WHEN}\; x\, f\, (\mathit{SUCC}\; m) \equiv f\, m \end{array}$$

which is useful to define continuous functions on natural numbers by cases. From the axiom for addition the usual recursion equations matching the cases above are derived by proof. 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 addition inherit this assumption. Definedness assumptions make reasoning about strict functions difficult [Pa85].

In HOL-CPO, a strict addition on `"lift Nat"` is introduced in the same way as the strict successor, by extending a built-in HOL function `$+`:

```
|- Add = Ext(\nn :: Dom Nat. Ext(\mm :: Dom Nat. Lift(nn+mm)))
|- Add ins (cf(lift Nat,cf(lift Nat,lift Nat))).
```

Note, by the way, that neither `Suc` nor `Add` are parameterized by any cpo variables since we work with the 'concrete' cpo of (lifted) natural numbers.

In LCF the recursion equations for addition are important in proofs because properties of addition are proved using natural number induction. In HOL we can reuse built-in theorems about addition which probably have been proved by similar inductions once, but without considering the bottom element as in LCF induction. For finite-valued types we can do the set theoretic developments in HOL before adding bottom. It is advantageous to defer reasoning about bottom until as late as possible in a proof, e.g. definedness assumptions tend to accumulate.

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

```
|- (!n. Add Bt n = Bt) /\
   (!n. Add n Bt = Bt) /\
   (!nn mm. Add(Lift nn)(Lift mm) = Lift(nn+mm)).
```

It states that addition is strict in both arguments and behaves as the built-in addition on lifted arguments.

The next fact we consider states that strict addition is total. That is, provided the arguments of `Add` are not bottom the result of applying `Add` will not be bottom. In LCF this fact would be stated by a theorem of the following form:

```
|- !n m. ~(n=Bt) ==> ~(m=Bt) ==> ~(Add n m = Bt).
```

Since we use lifting an equivalent statement in HOL is

```
|- !nn mm. ~(Add(Lift nn)(Lift mm) = Bt).
```

This can be derived immediately from the third clause of the above reduction theorem for addition using the facts that `Bt` and `Lift` are distinct and exhaustive on a lifted cpo.

Finally, let us consider two theorems stating that strict addition is associative and commutative:

```
|- !k m n. Add (Add k m) n = Add k(Add m n)
|- !m n. Add m n = Add n m
```

Their proofs are almost exactly the same in HOL; do a case split on the universally quantified variables (lifted numbers) one by one and reduce using the reduction theorem for addition after each case split. We end up with goals stating that the properties we wish to prove must hold for the built-in addition. So we finish off the proofs by using the desired built-in HOL facts:

```
|- !m n p. m + (n + p) = (m + n) + p
|- !m n. m + n = n + m
```

Such proofs by cases could be automated easily. The LCF proofs require much more thought. They use induction, in fact two nested inductions for commutativity, and rewriting.

5 A Mapping Functional for Lazy Sequences

In this section we define a mapping functional for lazy sequences and an infinite sequence constructor. One theorem is proved by fixed point induction and another is proved by "structural induction" on lazy sequences [Pa84a], i.e. structural induction is used to show the inclusive property holds of all finite sequences; the inclusiveness ensures it holds also of the infinite sequences. The purpose of this section is to show in which way HOL-CPO extends HOL with techniques for reasoning about infinite values, and recursive definitions in general.

The cpo of lazy sequences and its associated constructor and eliminator functions correspond exactly to the LCF type of sequences and its associated functions. The LCF type and constructor functions are introduced automatically by axioms similar to the axioms for natural numbers, using the same ML functions too. Developing the lazy sequences in HOL-CPO was difficult and time-consuming but we reason about sequences using the same techniques as in LCF.

A purely definitional development of a theory of lazy sequences is presented in [Ag94]. It provides a constructor called `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 `Bt_seq` and the lazy constructor function is called `Cons_seq`. These satisfy the following cases theorem

```
|- !D s.
    s ins (seq D) =
    (s = Bt_seq) \/
    (?x s'. x ins D /\ s' ins (seq D) /\ (s = Cons_seq x s'))
```

Further, they are distinct and `Cons_seq` is one-one. There is also an eliminator functional called `Seq_when` which can be used to write continuous functions on sequences by cases. Assuming `"x ins D"`, `"s ins (seq D)"` and `"h ins (cf(D,cf(seq D,E)))"` for a cpo `D` and a pointed cpo `E`, the following reduction theorem specifies the behavior of the eliminator:

```
|- (Seq_when h Bt_seq = bottom E) /\
   (Seq_when h(Cons_seq x s) = h x s)
```

The constants `Seq_when` and `Cons_seq` belong to the interface level syntax, internally they are parameterized by cpo variables (and called `Seq_whenI` and `Cons_seqI` respectively). In addition, we have derived a theorem for "structural induction" on lazy sequences from fixed point induction, following Paulson's approach [Pa84a].

All definitions, theorems and proofs about lazy sequences are very similar to the ones in LCF. The mapping functional is defined as the fixed point of a suitable functional as follows:

```
|- !D E.
    Maps =
    Fix
    (\g :: Dom(cf(cf(D,E),cf(seq D,seq E))).
      \f :: Dom(cf(D,E)).
       \s :: Dom(seq D).
        Seq_when
        (\x :: Dom D.\t :: Dom(seq D). Cons_seq(f x)(g f t))s)
|- !D E.
    cpo D ==> cpo E ==> Maps ins (cf(cf(D,E),cf(seq D,seq E)))
```

Internally, the constant `Maps` is parameterized by the cpo variables `D` and `E` of the definition. Using the reduction theorem for `Seq_when` and the fact that `Fix` yields a fixed point of a continuous function we can prove the following reduction equations easily:

```
|- (Maps f Bt_seq = Bt_seq) /\
   (Maps f(Cons_seq x s) = Cons_seq(f x)(Maps f s))
```

where `"x ins D"`, `"s ins (seq D)"` and `"f ins (cf(D,E))"` for cpos `D` and `E`. A tactic which takes such theorems as arguments can be used to reduce occurrences of `Maps` and other function constructors using a theorem like this one and the type checker to prove the assumptions automatically.

We can prove that the mapping functional preserves functional composition, i.e. assuming `"f ins (cf(D2,D3))"` and `"g ins (cf(D1,D2))"` for cpos `D1`, `D2` and `D3`, the following equation holds

```
|- Maps(Comp(f,g)) = Comp(Maps f,Maps g)
```

The constant `Comp` is defined as a determined version of the built-in functional composition (internally it is called `CompI`). The proof is conducted by observing that the two continuous functions are equal iff they are equal for all sequences of values in `D1`, i.e. iff the following term holds:

```
"!s.
  s ins (seq D1) ==>
  (Maps(Comp(f,g))s = Comp(Maps f,Maps g)s)".
```

Then we use an induction tactic based on the structural induction theorem for lazy sequences. This uses the inclusive prover behind the scenes to prove the equation admits induction. The proof is finished off using reduction tactics for `Maps` and `Comp`.

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

```
"Cons_seq x(Cons_seq(f x)(Cons_seq(f(f x))...))"
```

or written in a more readable way `[x; f(x); f(f(x)); ...]`. The function `Seq_of` is defined as a fixed point as follows:

```
|- !D.
    Seq_of =
    Fix
    (\sf :: Dom(cf(cf(D,D),cf(D,seq D))).
      \f :: Dom(cf(D,D)). \x :: Dom D. Cons_seq x(sf f(f x)))
|- !D. cpo D ==> Seq_of ins (cf(cf(D,D),cf(D,seq D)))
```

The internal version of `Seq_of` is parameterized by a cpo corresponding to the variable `D` in the definition. We have proved the following statement about `Maps` and `Seq_of`

```
|- !x. x ins D ==> (Seq_of f(f x) = Maps f(Seq_of f x))
```

where `D` is a cpo and `f` is a continuous function from `D` to `D`. Informally, the two sequences are equal since they are both equal to a term corresponding to `[f x; f(f x); ...]`. The proof of the theorem is conducted by fixed point induction on both occurrences of `Seq_of`; inclusiveness is proved behind the scenes.

The proofs in LCF and HOL-CPO are based on the same overall idea but tend to be longer in HOL. We must do many simplifications explicitly which are taken care of by LCF rewriting. We must use the reduction tactic to type check arguments of functions before their definitions can be expanded (by applying reduction theorems). LCF rewriting with definitions corresponds to such reductions since it also performs β-conversion.

6 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 = Const name | Var name | 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 [MW81] and Paulson made an attempt to translate their proof of correctness to LCF [Pa85]. 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 of the HOL proof are presented in [Ag94]. Although this example is considerably larger than the examples above it does not require deeper insights in 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 cannot be defined in pure HOL (at least not directly) since it is not primitive recursive. However, it can be defined as a fixed point 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. One may therefore argue that this approach provides a method, though probably not the simplest and most direct one, for defining recursive function by well-founded induction in HOL.

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 type of constant and variable names and the syntax type of terms must contain a bottom element, just like all other LCF types. Hence, definedness assertions of the form $t \not\equiv \perp$ 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 (derived from a recursion axiom):

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

In HOL substitution is introduced by a primitive recursive definition:

```
|- (!c s. (Const c) subst s = Const c) /\
   (!v s. (Var v) subst s = assoc(Var v)v s) /\
   (!t1 t2 s.
     (Comb t1 t2) subst s = Comb(t1 subst s)(t2 subst s))
```

Note this is pure HOL, we do not need to use domain theory to define a type of terms and `subst` . Terms and names of constants and variables are represented by HOL types which do not contain bottom, in contrast to the LCF types. 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. We can define discrete cpos of terms and names and lift these to contain a bottom when necessary, just as we did in the natural number example. Besides, we avoid PPλ's explicit statements of totality for functions such as SUBST which are obviously total,

$$\forall t\, s.\, t \not\equiv \perp \Rightarrow s \not\equiv \perp \Rightarrow t\, \mathit{SUBST}\ s \not\equiv \perp,$$

since HOL functions are always total.

The unification algorithm is stated as a collection of recursion equations in LCF. In HOL, the unification algorithm is defined as a fixed point of a certain functional, which unfortunately is too large (one page) to be presented here, and the recursion equations are then derived from the fixed point property. It is a continuous partial function as stated by:

```
|- unify ins (cf(term,cf(term,lift attempt)))
```

The cpo of terms 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 a discrete universal cpo with underlying type `":one"` and a discrete universal cpo with underlying type `":(name#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 of `unify` is stated as the theorem:

```
|- !t u. ?a. (unify t u = Lift a) /\ best_unify_try(a,t,u)
```

The first conjunct states `unify` is total and the second states it yields the best 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).

The unification algorithm is recursive on terms but it is not primitive recursive. In order to unify two combinations `"Comb t1 t2"` and `"Comb u1 u2"` the algorithm first attempts to unify `t1` and `u1` and if it succeeds with the substitution `s` as a result it attempts to unify `"t2 subst s"` and `"u2 subst s"`. The latter two terms may be bigger than 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 [Ag91] but never in LCF, because it is not possible to derive this general kind of induction from fixed point induction. 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.

Though the unification algorithm is a total function it is not straightforward to define it in 'pure' HOL since it is not primitive recursive. However, going via domain theory and well-founded induction to prove termination it is possible to introduce a pure HOL unification function. We can simply define this function using the choice operator as follows

```
|- !t u. Unify t u = (@a. unify t u = Lift a)
```

Furthermore, we can prove this function yields a best unifier for terms of type `":term"`.

```
|- !t u. best_unify_try(Unify t u,t,u)
```

From its definition, the recursion equations stating how it behaves on various kinds of arguments can be derived. This approach to derive a pure HOL unification function via domain theory and well-founded induction may be seen as a recursive definition by well-founded induction.

7 Conclusion

A contribution of this work is a comparison of two systems supporting domain theoretic reasoning, namely, LCF and the extension of HOL with domain theory. Using examples we show how HOL-CPO supports a mix of the two different kinds of reasoning provided in HOL and LCF, respectively. In a way, HOL-CPO can be seen as an embedding of the LCF system in HOL which is performed in such a way that the benefits of the HOL world are preserved.

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 gives a definition of an infinite sequence constructor functional as a fixed point and illustrates that we can conduct LCF proofs by fixed point induction and structural induction on infinite-valued recursive domains in HOL-CPO. This kind of reasoning is not possible in 'pure' HOL.

The unification example shows that we can avoid almost all reasoning about bottom that infests the LCF proof since it is an element of the type of expressions. In HOL, bottom is only introduced to allow a fixed point definition of the unification algorithm which is not primitive recursive and therefore cannot be defined in HOL directly. Other recursive functions of the example can be defined by primitive recursion in pure HOL, without using the formalization of domain theory at all.

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. The LCF proof uses two nested structural inductions to simulate well-founded induction which makes the proof more complicated, and less elegant too. Once it has been shown that the algorithm is total we can be define a total HOL function with the same behavior. Hence, the development can be seen as a way of defining a total HOL unification function by well-founded induction (see the end of section 6).

Some disadvantages of the embedding of domain theory in HOL have also been mentioned. One main problem is that it is time-consuming and not at all straightforward to introduce new recursive domains. Axiomatizing certain recursive types has been automated in LCF. Another problem is that constructors must be parameterized by the domains on which they work. This inconvenience is handled by an interface in most cases but the problem also affects the efficiency of proofs greatly since checking arguments of functions are in the right domains (called type checking) is inefficient.

One may compare the problems in LCF due to bottom to the problems in HOL-CPO due to the parameters on the dependent lambda abstraction and some function constructions. An interface could also be implemented in LCF to hide bottom in many cases but it would always be there in proofs. Often we avoid type checking in HOL-CPO. For instance, in the unification example where the bottom element was a major nuisance in LCF we worked most of the time in set theory where the problem of dependent functions (or bottom) does not exists. Domain theory was only used to define the recursive unification algorithm at a late stage of the proof.

HOL-CPO is a semantic embedding of domain theory in a powerful theorem prover. It was an important goal of this embedding that 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 in [Pe93] 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 [Wi93], if it was done. On the other hand, formalizations of Pω and information systems allow recursive domain equations to be solved fairly easily using the fixed point operator.

Franz Regensburger[1] is working on a very similar project in Isabelle HOL but the formalizations seem to be quite different. Pointed cpos are introduced using type classes and continuous functions constitute a type. Type checking arguments of functions seems not to be necessary but before β-reduction can be performed functions must be shown to be continuous (unlike in our formalization). Recursive domains can be axiomatized in a similar way as in LCF, though this has not been automated as in LCF. He is currently writing a Ph.D. thesis about the work (in German unfortunately). Bernhard Reus[2] works on synthetic domain theory in the LEGO system which implements a strong type theory

[1]Technical University, Munich. Email: `regensbu@informatik.tu-muenchen.de`

[2]Ludwig-Maximilian University, Munich. Email: `reus@informatik.uni-muenchen.de`

(ECC) with dependent sums and products. Dependent families can be exploited for the inverse limit construction of solutions to recursive domain equations. This is work in progress for a Ph.D. and the formalization has not been published yet.

Acknowledgements

This work was supported in part by the DART project funded by the Danish Research Council and in part by BRICS funded by 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 final draft. I am grateful to Larry Paulson for digging up the LCF proof of correctness of the unification algorithm.

References

[Ag91] S. Agerholm, 'Mechanizing Program Verification in HOL'. In the *Proceedings of the 1991 International Workshop on the HOL Theorem Proving System and Its Applications*, Davis California, August 28–30, 1991 (IEEE Computer Society Press). Also in Report IR-111, M.Sc. Thesis, Aarhus University, Computer Science Department, April 1992.

[Ag93] S. Agerholm, 'Domain Theory in HOL'. In the *Proceedings of the 6th International Workshop on Higher Order Logic Theorem Proving and its Applications*, Jeffrey J. Joyce and Carl-Johan H. Seger (Eds.), Vancouver, B.C., Canada, August 11–13 1993, LNCS 780, 1994.

[Ag94] S. Agerholm, *A HOL Basis for Reasoning about Functional Programs.* Ph.D. Thesis, Aarhus University, Computer Science Department, June 1994.

[GM93] M.J.C. Gordon and T.F. Melham, *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic.* Cambridge University Press, 1993.

[GMW79] M.J.C. Gordon, R. Milner and C.P. Wadsworth, *Edinburgh LCF: A Mechanised Logic of Computation.* Springer-Verlag, LNCS 78, 1979.

[MW81] Z. Manna and R. Waldinger, 'Deductive Synthesis of the Unification Algorithm'. *Science of Computer Programming*, Vol. 1, 1981, pp. 5–48.

[Me89] T.F. Melham, 'Automating Recursive Type Definitions in Higher Order Logic'. In G. Birtwistle and P.A. Subrahmanyam (eds.), *Current Trends in Hardware Verification and Theorem Proving*, Springer-Verlag, 1989.

[Pa84a] L.C. Paulson, 'Structural Induction in LCF'. Springer-Verlag, LNCS 173, 1984. Also in Technical Report No. 44, University of Cambridge, Computer Laboratory, February 1984.

[Pa84b] L.C. Paulson, 'Lessons Learned from LCF'. Technical Report No. 54, University of Cambridge, Computer Laboratory, August 1984.

[Pa85] L.C. Paulson, 'Verifying the Unification Algorithm in LCF'. *Science of Computer Programming*, Vol. 5, 1985, pp. 143–169. Also in Technical Report No. 50, University of Cambridge, Computer Laboratory, March 1984.

[Pa87] L.C. Paulson, *Logic and Computation: Interactive Proof with Cambridge LCF*. Cambridge Tracts in Theoretical Computing 2, Cambridge University Press, 1987.

[Pe93] K.D. Petersen, 'Graph Model of LAMBDA in Higher Order Logic'. In the *Proceedings of the 6th International Workshop on Higher Order Logic Theorem Proving and its Applications*, Jeffrey J. Joyce and Carl-Johan H. Seger (Eds.), Vancouver, B.C., Canada, August 11–13 1993, LNCS 780, 1994.

[Wi93] G. Winskel, *The Formal Semantics of Programming Languages*. The MIT Press, 1993.

Recent Publications in the BRICS Report Series

RS-94-18 **Sten Agerholm.** ***LCF Examples in HOL.*** **June 1994, 16 pp. To appear in:** ***Proceedings of the 7th International Workshop on Higher Order Logic Theorem Proving and its Applications*****, LNCS, 1994.**

RS-94-17 **Allan Cheng.** ***Local Model Checking and Traces.*** **June 1994, 30 pp.**

RS-94-16 **Lars Arge.** ***External-Storage Data Structures for Plane-Sweep Algorithms.*** **June 1994, 37 pp.**

RS-94-15 **Mogens Nielsen and Glynn Winskel.** ***Petri Nets and Bisimulations.*** **May 1994, 36 pp.**

RS-94-14 **Nils Klarlund.** ***The Limit View of Infinite Computations.*** **May 1994, 16 pp. To appear in the LNCS proceedings of Concur '94, LNCS, 1994.**

RS-94-13 **Glynn Winskel.** ***Stable Bistructure Models of PCF.*** **May 1994, 26 pp.** ***Preliminary draft.*** **Invited lecture for MFCS '94. To appear in the proceedings of MFCS '94, LNCS, 1994.**

RS-94-12 **Glynn Winskel and Mogens Nielsen.** ***Models for Concurrency.*** **May 1994, 144 pp. To appear as a chapter in the** ***Handbook of Logic and the Foundations of Computer Science*****, Oxford University Press.**

RS-94-11 **Nils Klarlund.** ***A Homomorphism Concept for ω-Regularity.*** **May 1994, 16 pp.**

RS-94-10 **Jakob Jensen, Michael Jørgensen, and Nils Klarlund.** ***Monadic Second-order Logic for Parameterized Verification.*** **May 1994, 14 pp.**

RS-94-9 **Gordon Plotkin and Glynn Winskel.** ***Bistructures, Bidomains and Linear Logic.*** **May 1994, 16 pp. To appear in the proceedings of ICALP '94, LNCS, 1994.**