J. Functional Programming **1** (1): 1–000, January 1998. Printed in the United Kingdom
© 1998 Cambridge University Press

HOLCF = HOL + LCF

Olaf Müller, Tobias Nipkow, David von Oheimb, Oscar Slotosch

http://www.in.tum.de/~{mueller|nipkow|oheimb|slotosch}

Institut für Informatik, Technische Universität München,
D-80290 München, Germany

Abstract

HOLCF is the definitional extension of Church's Higher-Order Logic with Scott's Logic for Computable Functions that has been implemented in the theorem prover Isabelle. This results in a flexible setup for reasoning about functional programs. HOLCF supports standard domain theory (in particular fixpoint reasoning and recursive domain equations) but also coinductive arguments about lazy datatypes. This paper describes in detail how domain theory is embedded in HOL and presents applications from functional programming, concurrency and denotational semantics.

1 Introduction

HOLCF is a logic for reasoning about functional programs. It provides arbitrary forms of recursion (via a fixpoint operator) and a package for defining datatypes. The latter caters for infinite objects, induction and coinduction. HOLCF is a synthesis of two logical systems, HOL and LCF, combining the best of both worlds. Before we go into technicalities (of which there is no shortage), we sketch the historical and logical roots of HOLCF.

The development of tactic-based interactive theorem provers started with LCF (Gordon *et al.*, 1979; Paulson, 1987), a system to support reasoning in Scott's *Logic for Computable Functions*. Aart from its many technical innovations, it was the first theorem prover to take the notion of partial computable functions seriously. Unfortunately, this commitment does not come cheap, as the users of LCF were to discover over time. Every type in LCF is a domain and every function is continuous. This reflects the denotational semantics view of programming, but rules out or at least complicates many other specification and verification tasks. A typical example is that in LCF every function is potentially partial, which complicates reasoning about total functions.

Partly as a result of his LCF experience, Gordon developed the HOL system (Gordon & Melham, 1993), where all functions are total. This goes a long way and has made HOL very popular, but breaks down as soon as truly partial functions, e.g. a programming language interpreter, enter the scene. A typical way out is to use a relation instead of a function, but this is neither natural nor simple. There is also the grey area of total functions whose totality is not easy to establish. The

standard HOL system only supports primitive recursive function definitions and it took a long while before more sophisticated definition mechanisms became available (Slind, 1996). In contrast, LCF allows arbitrary recursive function definitions.

Therefore it is natural to aim for a synthesis of HOL and LCF which allows both partial and total functions. This paper describes HOLCF, an extension of HOL with the notions of LCF, first designed by Regensburger (Regensburger, 1994; Regensburger, 1995) and further developed by the authors of this paper. HOLCF is based on Nipkow and Paulson's implementation of HOL within the generic Isabelle system (Paulson, 1994).

One of the key features of HOLCF is that it is a *definitional* extension of HOL with domain theory: instead of axiomatizing/hardwiring domain theory (as in LCF), its semantics is defined by means of conservative extensions mechanisms and the usual axioms are derived as theorems. Thus a large part of the paper is concerned with formalizing the semantics of domain theory in HOL and deriving the standard axiomatization. The flexibility of this approach enables us to compensate for incompletenesses of LCF by resorting to (fully formal) semantic arguments. The only exception to the definition principle is our datatype package, which asserts three axioms (while still being conservative!).

Despite the title of this paper, HOLCF is more than the sum of its parts. HOLCF supports not just two separate worlds — HOL's set theoretic one of total functions and LCF's domain theoretic one of continuous functions — but also the transition between them. This means that the LCF part is not a carbon copy of the original LCF but offers additional concepts designed to ease the transition to and from HOL. Although HOLCF cannot overcome the complications introduced by domains and partial functions, it can delay the point where they rear their ugly head. The underlying philosophy is to express as much as possible in the HOL basis and as much as necessary in the LCF extension (see (Agerholm, 1994b) for a similar philosopy).

The paper is structured as follows. After a brief introduction of Isabelle/HOL (§2) we describe the overall structure of HOLCF (§3). The core of the paper (§4) is a detailed presentation of the definition of the basic concepts of domain theory in HOL. This is followed by a user-oriented description of the package for the definition of recursive datatypes (§5). Finally (§6) we present a number of applications: functional programming with lazy lists, a model for Input/Output Automata, and the denotational semantics of a simple imperative language.

The paper assumes that the reader is familiar with notions from domain theory (see, for example, (Winskel, 1993)).

1.1 Related work

HOLCF in its current state shows several significant improvements on the initial version by Regensburger (Regensburger, 1994; Regensburger, 1995), which are all documented in this paper. The first improvement concerns the frequent applicability conditions attached to proof rules in domain theory: properties like being a cpo, a continuous function or an admissible predicate. HOLCF solves this problem

by coding as much as possible into the type which can then be handled automatically by type checking: partial orders are introduced as type classes and continuous functions constitute an independent type. Here we have achieved some significant improvements. *Axiomatic type classes* due to Wenzel (Wenzel, 1997) have replaced ordinary type classes. Their key advantage is that Isabelle checks the proposition that some type is a member of some axiomatic type class by insisting that the type satisfies the axioms of the type class; Regensburger did not have this device at his disposal and this left a potential source of unsoundness. Axiomatic type classes also replace Regensburger's complex semantic account of type classes by Wenzel's purely syntactic account. We have also added several new classes (for example cpos), thus making the class hierarchy more expressive. The automatic admissibility check has been enhanced significantly. The second improvement concerns a new datatype package which solves recursive domain equations automatically, handling also infinite and even mutually recursive datatypes. Regensburger (Regensburger, 1994) merely laid the logical foundation for such a package by providing an extra-logical argument that allows to construct datatypes in a conservative way. The third improvement concerns the methodological treatment of HOLCF. As mentioned above, HOLCF's philosophy is to stay in HOL as long as possible before moving to its more powerful, but also more complicated LCF extension. Such an approach is only possible due to the new interface between HOL and HOLCF. The importance of this interface has emerged during some major case studies, which are sketched in this paper as well.

Closely related to our work are the approaches by Agerholm (Agerholm, 1994b) and Bartels et al. (Bartels *et al.*, 1996) who extend Gordon's HOL and PVS, respectively, by notions of domain theory. However, there are significant differences. Neither system provides a datatype package and they deal differently with the large number of applicability conditions. Agerholm encodes partial orders as a pair of a carrier set and a relation and tries to cope with the applicability conditions by specialized proof tactics. This turned out to be distinctly more complex than our elegant solution using type classes. Furthermore, our admissibility test is stronger, as it employs a larger set of inference rules. The PVS approach employs predicate subtyping, type judgments, and theory parameterizations instead. As the authors admit, this has shortcomings as well, because the first is not powerful enough, the second does not allow for free variables, and the last results in cumbersome theory dependencies. Last but not least, the PVS version does not provide tactics for proving admissibility automatically.

There is also work on extending type theory with partial functions that employs notions of domain theory (Constable & Smith, 1987; Audebaud, 1991; Crary, 1998). This work is still largely concerned with overcoming theoretical problems arising from the use of type theory. On the other hand, the problems with formalizing and automating continuity mentioned above point out deficiencies in the type systems of both PVS and Isabelle/HOL: the former's lack of polymorphism, and the latter's lack of subtyping. There are type theories without these deficiencies, e.g. Nuprl (**?**).

2 Isabelle/HOL

Isabelle/HOL (Nipkow, 1998a) is the instantiation of the generic interactive theorem prover Isabelle (Paulson, 1994) with Church's formulation of Higher-Order Logic and is very close to Gordon's HOL system (Gordon & Melham, 1993). In this paper HOL is short for Isabelle/HOL. This section introduces just enough of HOL to make the paper self-contained. Below you find a short introduction to HOL's surface syntax:

Formulae The syntax is standard, except that there are two implications ($\longrightarrow$ and $\Longrightarrow$), two universal quantifiers ($\forall$ and $\bigwedge$), and two equalities (= and $\equiv$) which stem from the object and meta-logic, respectively. The distinction can be ignored while reading this paper. The notation $[\![A_1; \ldots; A_n]\!] \Longrightarrow A$ is short for the nested implication $A_1 \Longrightarrow \ldots \Longrightarrow A_n \Longrightarrow A$.

Types follow the syntax for ML-types, except that the function arrow is $\Rightarrow$.

Theories introduce constants with the keyword `consts`, non-recursive definitions with `defs`, primitive recursive definitions with `primrec`, new axioms with `rules`, and syntactic shorthands (macros) with `translations`. Further constructs are explained as we encounter them.

2.1 Axiomatic type classes

Axiomatic type classes were first suggested by Nipkow (Nipkow, 1993) and were turned into an integral feature of Isabelle by Wenzel (Wenzel, 1997). As the name indicates, axiomatic type classes can be viewed (as a first approximation) as an extension of Haskell-style type classes (Hudak *et al.*, 1992) by axioms. In a nutshell, an axiomatic type class can be viewed as a set of types satisfying certain axioms. Below we give an informal introduction. For a thorough treatment see the paper by Wenzel.

Type classes classify types just as types classify values. Given a type τ and a class C, the notation $\tau{::}C$ means that τ is of class C. Classes are partially ordered. The class of all HOL-types is called `term`; it is the greatest element in the class hierarchy.

In the simplest case, type classes merely provide an overloading mechanism. For example, in domain theory the symbol $\sqsubseteq$ represents different orderings on different domains. For this purpose we write

axclass `sq_ord < term` *"square ordering"*

which introduces a new subclass `sq_ord` of `term` (without any axioms). We can now use `sq_ord` to constrain type variables to range only over types of this class. Thus we can declare

consts $\sqsubseteq$ :: α::`sq_ord` $\Rightarrow$ α $\Rightarrow$ `bool` `(infix)`

which introduces the polymorphic predicate $\sqsubseteq$ and restricts its argument types to those of class `sq_ord`. The type checker will reject any term $s \sqsubseteq t$ unless s and t are both of some type τ such that $\tau{::}$`sq_ord` holds.

Note that in contrast to Haskell, where the declaration of a type class comprises

the declaration of its methods (i.e. functions), the declaration of $\sqsubseteq$ is separate from the declaration of class `sq_ord`. The reason is that membership of a type in an axiomatic class does not depend on the existence of certain functions but on the provability of certain axioms. Therefore we now turn to an example of a class involving axioms, the class of partially ordered types, which plays a crucial role in this paper. Their definition consists of a class `po`, a subclass of `sq_ord`, which imposes the usual axioms for a partial order:

axclass `po < sq_ord`

```
                        x⊑x
[| x⊑y; y⊑x |] ⟹ x = y
[| x⊑y; y⊑z |] ⟹ x⊑z
```

Starting from these axioms, we may derive further theorems about $\sqsubseteq$ which hold in any type of class `po`. In general, a type class declaration is of the form

axclass C `<` S_1`, ...,` S_n
$axiom_1$
...
$axiom_k$

Just like in Haskell, there is an `instance` declaration which tells Isabelle that some type is of a certain class. In Haskell, this comes with the opportunity to define the methods of the class. In Isabelle, this comes with the obligation to prove that the axioms of the class hold in that type. As an example, let us show that the set of functions from an arbitrary type into a partial order is again a partial order. For a start, we need to make $\sqsubseteq$ available on functions. We simply write

instance $\Rightarrow$ :: `(term,sq_ord)sq_ord`

which claims a certain "functionality" for the type constructor $\Rightarrow$, namely that $\sigma \Rightarrow \tau$ is of class `sq_ord` provided σ is of class `term` (any HOL type is) and τ is of class `sq_ord`. (The "functionality" of an n-ary type constructor is given as $(C_1, \ldots, C_n)C$ for suitable classes C_i and C.) Because the result class `sq_ord` has no axioms, there is nothing to prove. Thus, any type can be declared to be a member of a class without axioms. Such classes only serve to overload function symbols.

Now we can define the meaning of $\sqsubseteq$ on functions as the pointwise extension of $\sqsubseteq$ on the range type of the function:

defs `f ⊑ g ≡ ∀x. f x ⊑ g x`

Given this definition, it is easy to derive the above axioms for `po` as theorems about pointwise ordered functions; call those theorems `refl_less_fun`, `antisym_less_fun`, and `trans_less_fun`. Now we can convince Isabelle that the pointwise extension of a partial order is again a partial order by declaring

instance $\Rightarrow$ :: `(term,po)po (refl_less_fun,antisym_less_fun,trans_less_fun)`

From this point onwards all axioms and theorems involving the generic $\sqsubseteq$ can be used for functions whose range type is of class `po` because the type checker now knows that the function space itself is of class `po`.

Note that the definition of $\sqsubseteq$ is separate from the instance declaration. In fact, it must precede the instance declaration because in order to derive the necessary theorems for the instance declaration the definition needs to have been made already. Note further that the definition of $\sqsubseteq$ above is nontrivial because it involves

primitive recursion on types: $\sqsubseteq$ appears on both sides of the definitional equality, but its type on the right-hand side is strictly smaller than its type on the left-hand side. Note finally that it is quite legal to define $\sqsubseteq$ again, for example on some base type. As long as the multiple definitions do not overlap, Isabelle will accept them and keep them apart via their types. This is the reason why overloading works.

In the sequel we usually omit the presentation of the theorems needed for an `instance` declaration and write `(...)`.

An interesting refinement of the above method for introducing axiomatic type classes is due to Slotosch (Slotosch, 1997b; Slotosch, 1997a). He uses an additional constant to enforce that $s \sqsubseteq t$ is well-typed only if the type of s and t is of class `po`.

2.2 Defining types

The logic HOL is strongly typed. In order to avoid inconsistencies, every type has to be non-empty. There are three ways to define new types:

1. `types`, for example `types tr = bool lift`. This introduces an abbreviation for the user's convenience.
2. `datatype`, for example `datatype` α `lift = Undef | Def` α. This defines a free datatype together with theorems for induction etc. on the new type.
3. `typedef` is used to introduce types that are isomorphic to a non-empty subset of an existing type.

The most general way is the third one. Since we used it for the introduction of several types in HOLCF, we explain it here by an example:

typedef `pnat = {p::nat. 0<p}  (PosNE)`

In this example we define the type of positive natural numbers (`pnat`) to be (isomorphic to) the set of elements `p` of type `nat` (written `p::nat`) that fulfil the predicate `0<p`. The witness that the new type is non-empty, the theorem $\exists$`x. x:{p::nat. 0<p}` called `PosNE` is proved over natural numbers before the type `pnat` can be defined in this way. The `typedef` construct introduces the type only if the representing subset can be proved to be non-empty. HOL has no "real" subtyping, but subtypes may be introduced with coercion functions *abs* and *rep*. The `typedef` construct in the example automatically defines the subset of the representing values (called `pnat`, too) and the coercion functions:

```
consts  pnat      :: nat set
        Abs_pnat  :: nat ⇒ pnat
        Rep_pnat  :: pnat ⇒ nat
defs    pnat ≡ {p . 0<p}              definition of representing subset
rules   Rep_pnat x ∈ pnat
        y ∈ pnat ⟹ Rep_pnat(Abs_pnat y) = y
                   Abs_pnat(Rep_pnat x) = x
```

These coercion functions can be used to define functions on the subtype, for example `plus_pnat = λx y. Abs_pnat (Rep_pnat x + Rep_pnat y)`.

2.3 Proof procedures

Isabelle provides a 'simplifier' and a 'classical reasoner'. The simplifier performs conditional and unconditional rewriting and uses contextual information. The classical reasoner provides automation for logical formulas by doing some tableau search for proofs. See (Paulson, 1994; Paulson, 1997) for more details.

We only sketch the most important feature for HOLCF: *simplification sets*. A simplification set is a collection of (conditional) rewriting rules applicable by the simplification tactics. Every theory has a default simplification set, to which new theorems can be added as appropriate.

3 Structure of HOLCF

Isabelle theories are named and hierarchically structured (via the `+` operation on theories). The theory structure of HOLCF is shown in Fig. 1. The contents of each theory is summarized below.

`Porder`	Partial orders (§2.1).
`Pcpo`	Various classes of cpos (§4.1.2 and §4.1.3).
`Fun`	Function spaces as cpos (§4.2.1).
`Cont`	Continuous function spaces as cpos (§4.2.2).
`Cfun`	The (sub)type of continuous functions (§4.2.2).
`Discrete`	Discrete cpos (§4.3.1).
`Ssum`	Strict sums (§4.3.5).
`Sprod`	Strict products (§4.3.6).
`Cprod`	Cartesian products (§4.3.6).
`Up`	Lifting for cpos (§4.3.7).
`Fix`	The fixpoint operator (§4.2.4).
`Lift`	Lifting arbitrary HOL types to (flat) domains (§4.3.2).
`One`	Domain with a single defined element (§4.3.3).
`Tr`	Domain of truth values (§4.3.4).

A secondary structuring principle is the hierarchy of orders (partial orders, cpos, pcpos, etc.). These orders are represented as type classes and influence the exact theory structure: declaring a type to be an instance of a type class requires theorems about that type and requires the type to be an instance of all superclasses. For example, for continuous functions we have the following theory structure:

`Cfun1 = Cont` + *definition of* $\rightarrow$ *and* $\sqsubseteq$, *proofs for* `po`
`Cfun2 = Cfun1` + *declaration of* $\rightarrow$ *as instance of* `po`, *proofs for* `cpo` *and* `pcpo`
`Cfun3 = Cfun2` + *declaration of* $\rightarrow$ *as instance of* `cpo` *and* `pcpo`

The fine structure of individual theories is ignored in Fig. 1.[1]

[1] The full structure can be found at `www.in.tum.de/~isabelle/library/HOLCF/`.

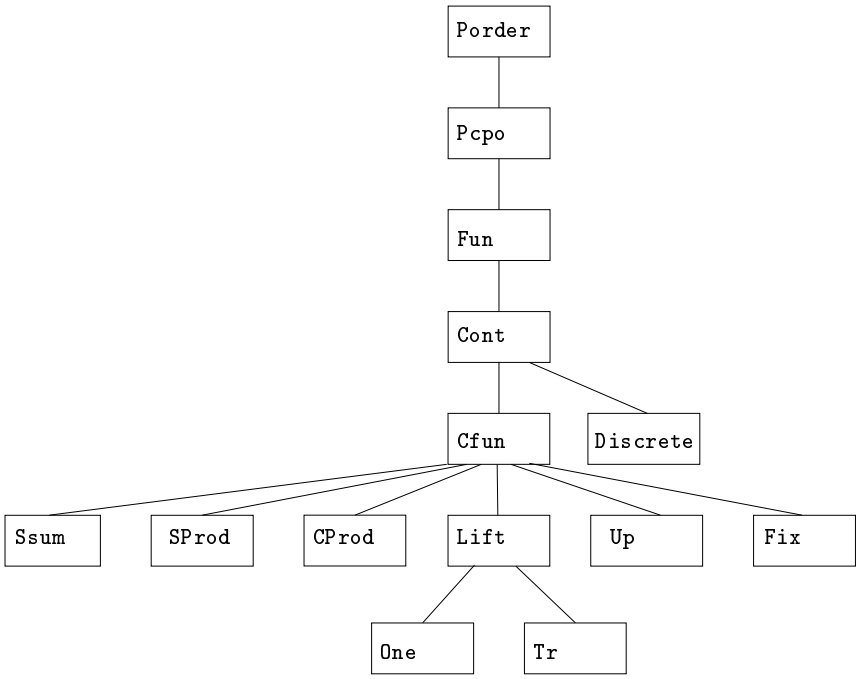

Fig. 1. Theory structure of HOLCF

4 Domain theory

4.1 Partially ordered sets

Probably the most unique aspect of our work is the use of axiomatic type classes to define the order-theoretic basis of domain theory. In §2.1 we have already introduced the predicate $\sqsubseteq$ and the class `po` of partial orders. We now continue to develop the class hierarchy.

4.1.1 Chains and upper bounds

Domain theory is based on *complete partial orders*, a refinement of orders where certain sets must have least upper bounds. HOLCF is based on the popular notion of ω-*chains*, which are formalized as certain functions from `nat` into a partial order:

consts `chain :: (nat ⇒ α::po) ⇒ bool`
defs `chain Y ≡ ∀i. Y i ⊑ Y (Suc i)`

Upper bounds and least upper bounds are formalized as expected:

consts `<|, <<|  :: (α::po)set ⇒ α ⇒ bool  (infixl)`
`lub  :: (α::po)set ⇒ α`
defs `S <| x  ≡ ∀y∈S. y⊑x`
`S <<| x  ≡ S <| x ∧ (∀u. S <| u ⟶ x⊑u)`
`lub S  ≡ εx. S <<| x`

This is pretty much unchanged from Regensburger's original treatment. The main

point to note is that because least upper bounds may not exist, the functional notation `lub S`, though more convenient, is weaker than the relational `S` $\ll|$ `x`, which also asserts the existence of a least upper bound, namely `x`.

4.1.2 Complete partial orders (cpos)

Complete partial orders are defined as a subclass `cpo` of `po`:

axclass `cpo < po`
`chain Y` $\Longrightarrow$ `∃x. range Y` $\ll|$ `x`

The axiom says that all ω-chains must have least upper bounds. The HOL constant `range` :: $(\alpha \Rightarrow \beta) \Rightarrow \beta$ `set` returns the set of all possible results of a function:

defs `range f` $\equiv$ `{y. ∃n. y = f n}`

4.1.3 Pointed cpos (pcpos)

Some constructions of domain theory only require cpos, but many need in addition a least element, commonly denoted by $\bot$. Thus we introduce the class of *pointed complete partial orders*:

axclass `pcpo < cpo`
`∃x. ∀y. x`$\sqsubseteq$`y`

and give a name to the least element using Hilbert's description operator

consts $\bot$ :: α::`pcpo`
defs $\bot \equiv$ `εx. ∀y. x`$\sqsubseteq$`y`

The use of the ε-operator, which was suggested to us by Wenzel, may look a bit roundabout. Why did we not introduce $\bot$ before we declared `pcpo` and state the axiom as $\bot\sqsubseteq$`y`? The subtle reason is that if we declare $\bot$ first, it cannot have type α::`pcpo` because `pcpo` is not known yet. Declaring it to be of some known type like α::`cpo` causes no logical problems (Wenzel, 1997), but looks a bit odd and can confuse inexperienced users: it allows well-typed but meaningless terms (we cannot prove anything interesting about them) involving $\bot::\tau$ where τ is not of class `pcpo`.

Note that in LCF every type is a pcpo. Regensburger's HOLCF follows this lead and does not introduce a separate class of cpos. This makes things more uniform and simpler for the novice. The main drawback is that when turning a HOL type into a domain, one has no choice but to make it a pcpo, thus introducing a fictitious $\bot$-element. This in turn complicates reasoning about the type. Therefore HOLCF now follows Winskel (Winskel, 1993) in distinguishing the intermediate class `cpo`. See §4.3.1 for an application.

4.1.4 Flat and chain-finite cpos

Monotonicity of functions ensures the existence of a least fixpoint and continuity allows to calculate it as the limit of the Kleene chains. Due to the combination of HOL and LCF functions, continuity does not come for free and has to be proved

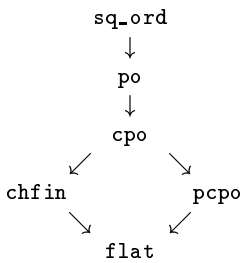

Fig. 2. Subclass structure of HOLCF

sometimes. Often, it is non-trivial to prove that a given function has these properties, but there are special subclasses of domains where monotonicity and continuity are guaranteed automatically. These important subclasses are the *flat* pcpos and the *chain-finite* cpos:

axclass `flat < pcpo`
`x`$\sqsubseteq$`y` $\Longrightarrow$ `x = ⊥ ∨ x = y`

axclass `chfin < cpo`
`chain Y` $\Longrightarrow$ $\exists$`n. max_in_chain n Y`

where `max_in_chain` is defined as:

consts `max_in_chain` :: `nat ⇒ (nat ⇒ α::po) ⇒ bool`
defs `max_in_chain i C ≡ ∀j. i≤j ⟶ C i = C j.`

In an earlier version of HOLCF, both `flat` and `chfin` were modelled as predicates taking a dummy argument that merely carries the type

`flat (x::α) ≡ ∀x::α y. x`$\sqsubseteq$`y ⟶ x = ⊥ ∨ x = y,`

and the well-known fact that any flat pcpo is chain-finite was given as

`flat x` $\Longrightarrow$ `chfin x`

Lifting the latter implication to the class level is interesting because it goes beyond Haskell's type system. We *prove* that `flat` is a subclass of `chfin` by proving that the axiom for `chfin` holds in all flat pcpos:

`chain (Y::nat⇒α::flat) ⟶ ∃n. max_in_chain n Y`

Calling this lemma `flat_subclass_chfin`, we can convince Isabelle of the subclass relationship `flat < chfin` using an extended `instance` declaration:

instance `flat < chfin (flat_subclass_chfin)`

As a result we have the subclass hierarchy depicted in Fig. 2. The key advantage of turning `flat` and `chfin` from predicates into type classes is that the subclass hierarchy is automatically taken into account during deductions (via unification).

Typical applications of these type classes include the fact that any strict function from a flat pcpo is monotone

`f (⊥::α::flat) = ⊥` $\Longrightarrow$ `monofun f`

and that any monotone function from a chain-finite cpo is continuous:

```
monofun (f::(α::chfin)⇒β) ⟹ cont f.
```

Further applications follow in the section on admissibility (§4.2.4).

4.2 Function spaces

Having defined abstract notions of partially ordered sets, we now need to populate these classes. We begin with the most important construction, the function spaces.

4.2.1 Full function spaces

It is easy to show that the type of all functions into a pcpo is again a pcpo. The first steps, culminating in the instance declaration ⇒ :: `(term,po)po` are explained in §2.1 above. In the same fashion we prove

instance ⇒ :: `(term, cpo) cpo` `(...)`
instance ⇒ :: `(term,pcpo)pcpo` `(...)`

4.2.2 Continuous functions

On top of the full function space domain we define a predicate `cont` to characterize continuous functions:

consts `cont` :: `(α::cpo ⇒ β::cpo) ⇒ bool`
defs `cont f ≡ ∀Y. chain Y ⟶ f``range Y ⋘ f (lub(range Y))`

where `f``A ≡ {y. ∃x∈A. y = f x}`. Continuity can alternatively be characterized using the following two predicates

defs `monofun f ≡ ∀x y. x ⊑ y ⟶ f x ⊑ f y`
`contlub f ≡ ∀Y. chain Y ⟶ f (lub(range Y)) = lub(f``range Y)`

The following theorem expresses the equivalence between the two notions:

```
cont f = (monofun f ∧ contlub f)
```

As one of HOLCF's main features we now define a new type of continuous functions:

typedef `(α → β) = {f. cont f}` `(...)`

Thus, the continuous function space constructor is →. As explained in §2.2, this type definition introduces the following abstraction and representation functions:

consts `Abs_CFun` :: `(α ⇒ β) ⇒ (α → β)`
`Rep_CFun` :: `(α → β) ⇒ (α ⇒ β)`

These two constants represent abstraction and application for type →. To increase readability, we add a bit of syntax exmphasizing this fact:

`Λx. t` stands for `Abs_Cfun (λx. t)` and `c`x` stands for `Rep_Cfun c x`.

Starting with the axiom `f∈{f. cont f} ⟹ Rep_CFun (Abs_CFun f) = f` generated by the definition of →, we immediately derive β-reduction of continuous functions:

```
cont f ⟹ (Λx. f x)`u = f u          β-reduction
```

Table 1. *Function spaces in HOLCF*

space	name	abstraction	application	β-reduction
full	$\Rightarrow$	`λx. t`	`f t`	`(λx. t) u = t[u/x]`
continuous	$\rightarrow$	`Λx. t`	`f‘t`	`cont t ⟹ (Λx. t)‘u = t[u/x]`

Cpos and pcpos are closed under $\rightarrow$:

instance $\rightarrow$ :: `(cpo, cpo) cpo (...)`
instance $\rightarrow$ :: `(cpo,pcpo)pcpo (...)`

For the proofs of the necessary witnesses see (Regensburger, 1994).

Of course, for continuous functions we also have the identity and composition operators, defined as `ID ≡ (Λx. x)` and `f oo g ≡ (Λx. f‘(g‘x))`.

4.2.3 Reasoning about continuity

Thus, we have two function spaces in HOLCF, denoted by $\Rightarrow$ and $\rightarrow$ (see Table 1). Terms of the continuous function space (Λ-abstractions and ‘-applications) constitute the so-called *LCF sublanguage* of HOLCF. The idea is to capture continuity implicitly by the type system as much as possible. Indeed, terms of the LCF sublanguage are automatically recognized to be continuous by the type checker. However, continuity cannot always be handled implicitly: the β-reduction rule in this framework generates an explicit proof obligation for continuity, as shown above. Therefore there is a special continuity tactic in HOLCF that discharges those proof obligations. This tactic reduces the continuity of an LCF term to the continuity of its basic components by structural rules that syntactically follow Λ-abstractions and ‘-applications. All possible basic components — constant functions, the identity, the conditional, and the composition operation — have been proved to be continuous once and for all. Thus, continuity of a term of the LCF sublanguage can always be determined automatically, either implicitly by type checking or explicitly by the continuity tactic.

4.2.4 Fixpoint induction and admissibility

One of the most interesting parts of domain theory are the fixpoint operator and the fixpoint induction rule. The fixpoint operator `fix` is defined as

consts `fix :: (α::pcpo → α) → α`
defs `fix ≡ (Λf. lub(range(λi. iterate i f ⊥)))`

where `iterate :: nat ⇒ (α → α) ⇒ α ⇒ α` is defined by primitive recursion such that `iterate i f` denotes f^i. It is proved that `fix` is indeed continuous, as the definition using the type constructor $\rightarrow$ suggests. As an important result we get the following characteristic property of the fixpoint operator:

```
fix‘f = f‘(fix‘f)
```

Note that the usual continuity assumption for this theorem is not explicitly needed in our setting, as it is already built into the type constructor $\rightarrow$ used for any argument `f` of `fix`. This considerably facilitates reasoning about fixpoint equalities.

Fixpoint induction. The fixpoint induction rule has been derived as usual:

```
⟦adm P; P ⊥; ⋀x. P x ⟹ P (f`x)⟧ ⟹ P (fix`f)
```

As known from the literature (e.g. (Paulson, 1987)), this rule includes the admissibility of `P` as an assumption. A predicate `P` is admissible iff it holds for the least upper bound of every chain satisfying `P`.

```
consts    adm    :: (α::cpo ⇒ bool) ⇒ bool
defs      adm P ≡ ∀Y. chain Y ⟶ (∀i. P (Y i)) ⟶ P (lub(range Y))
```

In practice, it is of vital importance that admissibility proof obligations are discharged automatically. For this reason, an admissibility check has been implemented in HOLCF.

Admissibility Check. From the literature (e.g. (Paulson, 1987)) a number of theorems are known that determine the admissibility of a predicate simply by exploiting its syntactic structure. Two of them are shown here:

```
⟦adm P;  adm Q ⟧ ⟹ adm (λx. P x ∨ Q x)
⟦cont u; cont v⟧ ⟹ adm (λx. u x⊑v x)
```

All these structural rules are known to HOLCF's simplifier. Thus, in a lot of cases Isabelle reduces the admissibility obligation for a predicate `P` to continuity obligations for the functions occurring in `P`. These continuity obligations are likewise automatically discharged by the simplifier according to §4.2.2, at least if we stay in the LCF sublanguage (i.e. in the continuous function space). For extensions of the continuity tactic to mixed HOL and LCF terms, see §4.3.2.

If this syntactic check fails, there is another arrow in our quiver: the substitution theorem

```
⟦cont t; adm P⟧ ⟹ adm (λx. P (t x))
```

Unfortunately, it produces a great number of higher-order unifiers when applied without explicit instantiation. In practice, however, it is only used in combination with the fact that every predicate with a chain-finite argument type is admissible:

```
adm (λx::α::chfin. P x)
```

Therefore, by combining these two rules, we get the rule

```
cont t ⟶ adm (λx. P ((t x)::α::chfin))
```

which has been implemented as a proof procedure that enumerates all such chain-finite subterms `(t x)` and checks if they are continuous in `x`. The proof procedure is automatically used if the standard admissibility check fails. The example `f`x ≠ TT`, where `f` is continuous and `TT` a value from a flat domain (here: the truth values defined in §4.3.4), may serve to illustrate the power of this test, as it is not covered by the structural rules described above. See the end of §6.3 for a further example.

The resulting admissibility test demonstrates the advantage of higher-order logic: Whereas in LCF the admissibility check was implemented as a hardwired and incomplete oracle, our test performs a real proof and, in addition, if it fails there is at least the possibility to prove admissibility manually by its semantic definition.

Weak admissibility. In HOLCF there is also a weaker version of admissibility with the same type as `adm`, called `admw` that is more closely related to the notion of Kleene chains.

defs `admw P` $\equiv$ `∀f. (∀n. P (iterate n f ⊥)) ⟶ P (lub(range(λi. iterate i f ⊥)))`

It is easily shown that `adm P` implies `admw P`. As before for `adm P`, it is possible to derive a corresponding fixpoint induction rule.

$[\![$`admw P; ∀n. P (iterate n f ⊥)`$]\!] \Longrightarrow$ `P (fix‘f)`

In rare cases this rule is helpful, as it requires only the weaker assumption `admw P`. This may be the case if `adm P` cannot be shown automatically using the admissibility test described above.

4.3 Other domain constructions

Once we are equipped with function spaces on pcpos, we can define other general domain constructions together with the typical functions acting on them. This subsection presents the most important basic domains provided in HOLCF. They are useful in their own right and also serve as the building blocks for composite datatypes. For example, the one-elements type, the strict sum, non-strict and strict products, and lifting of domains are heavily used by the datatype package for user-defined recursive domains (see §5).

All domains presented in this subsection are introduced definitionally, for example via a `datatype` or a `typedef`. For reasons of space we rarely present the full construction but only the functions offered and a few key properties. For details see the literature (Regensburger, 1994; Paulson, 1987).

4.3.1 Discrete cpos

A discrete cpo is one where the ordering is the identity. The *raison d'être* of discrete cpos is to provide a simple means of turning arbitrary types into cpos. The type constructor

datatype `(α)discr = Discr α`

turns any HOL type τ into the discrete cpo (τ)`discr` by defining

defs `(x::(α::term)discr)` $\sqsubseteq$ `y` $\equiv$ `x = y`

and declaring

instance `discr :: (term)cpo` `(...)`

where the theorems `(...)` are trivial to prove.

The advantage of the extra type constructor `discr` is that one does not need to turn each individual type by hand into a discrete cpo (as done for `discr`) but one simply wraps the type up in `discr`. To unwrap the wrapped up values there is a function `undiscr :: (α::term)discr ⇒ α` with the property `undiscr(Discr x) = x`.

For an application of discrete cpos see §6.3.

4.3.2 Lifting of HOL types

It is often useful to turn HOL types not only into cpos (using `discr`), but directly into pcpos. This is done with the type constructor `lift` that lifts HOL types to flat domains by defining

datatype α `lift = Undef | Def` α
defs `(x::(`α`::term lift)) ⊑ y ≡ (x = Undef ∨ x = y)`

and declaring a series of instance declarations (the result type class ranging from `po` to `pcpo`), culminating in

instance `lift :: (term)flat (...)`

In particular, it is proved that `Undef` plays the role of the least element. Accordingly, from now on $\bot$ is used in favour of `Undef`. Moreover, every theorem generated for `lift` by the datatype package of HOL is reformulated with $\bot$ instead of `Undef`, so that `Undef` is completely hidden from the user.

In order to stay in HOL as long as possible before switching to the LCF extension, we introduce the functionals `flift1` and `flift2`.

consts `flift1 :: (`α `⇒` β`::pcpo) ⇒` α `lift →` β
`flift2 :: (`α `⇒` β `) ⇒` α `lift →` β `lift`

The former lifts the argument type of a HOL function and expects the range type to be a pcpo, while the latter lifts both argument and range types. Basically, they extend HOL functions in a strict way:

```
consts flift1C :: (α ⇒ β::pcpo) ⇒ α lift ⇒ β
       flift2C :: (α ⇒ β)       ⇒ α lift ⇒ β lift
defs   flift1  f ≡ Λx. flift1C f x
       flift1C f ≡ λx. case x of ⊥ ⇒ ⊥ | Def y ⇒      f y
       flift2  f ≡ Λx. flift2C f x
       flift2C f ≡ λx. case x of ⊥ ⇒ ⊥ | Def y ⇒ Def (f y)
```

Two continuity theorems have been proved about `flift1C` and `flift2C`:

```
⟦⋀y. cont (λx. (f x) y); cont g⟧ ⟹ cont (λx. flift1C (f x) (g x))
                          cont g  ⟹ cont (λx. flift2C  f    (g x))
```

These lemmas are sufficient for proving every continuity obligation about a mixed HOL/LCF term that obeys a certain methodology, namely: At the inner level of the term there may be pure HOL terms, which are lifted to LCF by the "one-way" interface functions `Def`, `flift1` and `flift2`, while at the outer level there are terms of the LCF sublanguage only.

We sketch the proof idea for this completeness result here; see (Müller, 1998b) for a formal inductive argument over the structure of terms. Consider a continuity proof obligation for a lifted HOL term. It is immediately clear that the only continuity obligation that might not be captured by the two lemmas above could concern the first argument of `flift2C`. However, due to the "one-way" interface methodology, it should be a pure HOL term. Thus, a continuity obligation is not even expressible.

Therefore, by adding these two lemmas to the continuity lemmas described in §4.2.2, the simplifier is now able to discharge every continuity obligation of mixed HOL and LCF terms automatically, provided that the methodology is observed.

The result is a well-defined interface between HOL and its LCF extension which allows to integrate HOL terms without the drawback of manual continuity proofs. In several case studies this methodology turned out to be of great practical value (see (Müller, 1998b) for details).

A major application of the `lift` theory are argument types of recursive domain constructions. See §6.1 for the example of lazy lists. Another nice example for the use of type lifting are the operations on truth values and the one-element domain, which will be introduced next.

4.3.3 A one-element domain

The type `one` is a domain with one single defined element (`ONE`) and the bottom element. It is used for constants in the domain package (see §5). The example also shows how lifting of types works:

```
types one = unit lift
consts  ONE :: one
defs    ONE ≡ Def ()
```

The type `unit` is defined in HOL and contains the single element `()`.

4.3.4 Truth values

The truth values `TT, FF,` $\bot$ are defined by lifting the boolean values `True` and `False`.

```
types   tr = bool lift
defs    TT ≡ Def True
        FF ≡ Def False
```

The continuous conditional expression, written `If _ then _ else _`, is obtained by lifting HOL's `if _ then _ else _` in the first argument:

```
If b then e1 else e2 ≡ flift1 (λb. if b then e1 else e2)‘b
```

Then, the logical connectives are defined as follows:

```
defs neg     ≡  flift2 Not
     andalso ≡ (Λx y. If x then y else FF)
     orelse  ≡ (Λx y. If x then TT else y)
```

4.3.5 Strict sum

The strict sum of two pcpos, written $\alpha \oplus \beta$, behaves like the disjoint union but identifies the two $\bot$-elements. Its encoding is a variation of the usual encoding of disjoint unions in HOL (Gordon & Melham, 1993; Paulson, 1994), modified to take $\bot$ into account (Regensburger, 1994). It takes a certain amount of work to show that the strict sum of two pcpos is again a pcpo:

```
instance ⊕ :: (pcpo,pcpo)pcpo   (...)
```

There are two injections and a functional for case distinctions:

consts `sinl` :: $\alpha \to \alpha \oplus \beta$
`sinr` :: $\beta \to \alpha \oplus \beta$
`sscase` :: $(\alpha \to \gamma) \to (\beta \to \gamma) \to \alpha \oplus \beta \to \gamma$

On non-$\bot$ elements they behave like the disjoint union in HOL, e.g. the injections are injective and

```
x ≠ ⊥ ⟹ sscase`f`g`(sinl`x) = f`x
x ≠ ⊥ ⟹ sscase`f`g`(sinr`x) = g`x
```

All three functions are strict: `sinl`⊥ = ⊥`, `sinr`⊥ = ⊥` and `sscase`f`g`⊥ = ⊥`.

4.3.6 Products

The strict product of two pcpos (construction omitted), written $\alpha \otimes \beta$, is again a pcpo:

instance $\otimes$ :: `(pcpo,pcpo)pcpo` `(...)`

The basic constructors and destructors for strict products are

consts `spair` :: $\alpha \to \beta \to \alpha \otimes \beta$
`sfst` :: $\alpha \otimes \beta \to \alpha$
`ssnd` :: $\alpha \otimes \beta \to \beta$
`ssplit` :: $(\alpha \to \beta \to \gamma) \to \alpha \otimes \beta \to \gamma$
translations `(|x, y|)` ≡ `spair`x`y`

which satisfy, among others, the following important properties:

```
sfst`⊥ = ⊥                y ≠ ⊥ ⟹ sfst`(|x,y|) = x
ssnd`⊥ = ⊥                x ≠ ⊥ ⟹ ssnd`(|x,y|) = y
                    ssplit`f`⊥ = ⊥
⟦x ≠ ⊥; y ≠ ⊥⟧ ⟹ ssplit`f`(|x,y|) = f`x`y
```

For some purposes we need also non-strict products. They are obtained from the ordinary HOL products by adding the componentwise ordering and defining continuous constructors and destructors. For example, the constructor is `<x,y>`, which is distinct from ⊥ unless both `x` and `y` are ⊥, in contrast to `(|x,y|)`.

4.3.7 Lifting of domains

With the above strict constructions, one can only build strict datatypes. For the representation of *lazy* datatypes (e.g. streams) we provide a lifting type constructor `u` that adds a new ⊥-element. The representation of this type is a sum of the unit type and the argument type: `typedef` α `u = {x::(unit + `α`). True}`. The constructor `u` turns any cpo into a pcpo, the proof of which takes a bit of work.

We provide a lazy lifting function `up` :: $\alpha \to$ `(`α`)u` for lifting elements into the type, and functional `fup` of type `(`$\alpha \to \gamma$`) → (`α`)u` → γ for lifting functions. Their characteristic properties are

```
up`⊥ ≠ ⊥     (up`x ⊑ up`y) = (x⊑y)     fup`f`⊥ = ⊥     fup`f`(up`x) = f`x
```

The difference between this lifting and the lifting of HOL types in §4.3.2 is that it preserves the structure of the underlying domain, whereas the `lift` construction builds a flat domain. Structure preservation is essential, for example, for the prefix-ordering on the domain of streams (see §6.1).

5 Datatype package

As recursive datatypes are used ubiquitously in (functional) programming itself as well as in reasoning about functional programs, the HOLCF system provides a package (Oheimb, 1995) for their convenient definition and application.

The package, invoked with the keyword `domain`[2], can handle mutually recursive definitions of free datatypes, even infinite ones (with non-strict constructors). It determines and proves the characteristic properties of each datatype defined, including strictness, definedness, distinctness and injectivity of the constructors, as well as induction and coinduction principles.

Our package resembles the `gen_struct_axm` command of LCF (Paulson, 1987), except that it handles also mutual recursion. Furthermore, while `gen_struct_axm` simply asserts most characteristic properties of the datatypes as axioms, we construct all user-relevant entities by definitions and prove their properties from a minimal set of axioms. A general category-theoretic argument (Regensburger, 1994) confirms the consistency of these axioms, from which we can conclude that our datatype construction is conservative.

Agerholm (Agerholm, 1994b) takes a rather different approach. He formalizes recursive domains using the type definition package of HOL and providing them with a cpo structure manually. This enables reuse of HOL types and their properties, but the construction is non-automatic and therefore tedious, and it does not handle mutually recursive types or infinite elements. He also gives an ad-hoc formalization of lazy lists. On the other hand, he has formalized (Agerholm, 1994a) the inverse limit construction for solving general recursive domain equations in ZF. It is not clear whether this approach is useful in practice in particular since he used the logic HOL-ST, an unusual mixture of HOL and set theory.

The use of our package is similar to ML `datatype` declarations, except that also destructors and discriminators are defined and indirect recursion is not allowed. As a general formal description of the input format and the corresponding output of the package would be rather lengthy and a bit cumbersome, we explain the application of the package by typical examples.

5.1 Free datatypes

Like in functional programming languages, free datatypes are defined via their constructors. Here we define the well-known datatype of (polymorphic) lazy lists, i.e. possibly infinite sequences over elements of any pcpo type α. We have chosen this example because, while being non-trivial, it should be easy to understand and will be used extensively in §6.

domain α `llist = nil | # (hd::`α`) (lazy tl::`α `llist) (cinfixr)`

The empty list is denoted by `nil`, while the "cons" constructor is given by the

[2] its counterpart for plain HOL types is invoked by `datatype`.

right-associative infix symbol[3] `#`, with selector functions `hd` and `tl`. Appropriate discriminator functions, here `is_nil` and `is_#`, are derived automatically.

As, by definition, the binary constructor is strict in its first argument and lazy in the second, elements of type α `llist` come in three flavors:

- finite total sequences: $\mathtt{a}_1\mathtt{\#...\#a}_n\mathtt{\#nil}$
- finite partial sequences: $\mathtt{a}_1\mathtt{\#...\#a}_n\mathtt{\#}\bot$
- infinite sequences: $\mathtt{a}_1\mathtt{\#a}_2\mathtt{\#a}_3\mathtt{\#...}$

5.1.1 Syntax

The datatype package generates the following entities as syntactic representation of datatypes:

- the type(s) with their arities, in our example

  types `llist 1`
  instance[4] `llist :: (pcpo)pcpo (...)`

- the isomorphism pair between the folded (abstract) and unfolded (representing) version of the type

  consts `llist_abs` :: $\mathtt{one} \oplus (\alpha \otimes (\alpha\ \mathtt{llist})\mathtt{u}) \to \alpha\ \mathtt{llist}$
  `llist_rep` :: $\alpha\ \mathtt{llist} \to \mathtt{one} \oplus (\alpha \otimes (\alpha\ \mathtt{llist})\mathtt{u})$

- the `case`, `copy`, and `take` auxiliary functionals, which are described below

  consts `llist_case` :: $\tau \to (\alpha \to \alpha\ \mathtt{llist} \to \tau) \to \alpha\ \mathtt{llist} \to \tau$
  `llist_copy` :: $(\alpha\ \mathtt{llist} \to \alpha\ \mathtt{llist}) \to \alpha\ \mathtt{llist} \to \alpha\ \mathtt{llist}$
  `llist_take` :: $\mathtt{nat} \Rightarrow \alpha\ \mathtt{llist} \to \alpha\ \mathtt{llist}$

- predicates for finiteness and the characterization of bisimulations

  consts `llist_finite` :: $\alpha\ \mathtt{llist} \Rightarrow \mathtt{bool}$
  `llist_bisim` :: $(\alpha\ \mathtt{llist} \Rightarrow \alpha\ \mathtt{llist} \Rightarrow \mathtt{bool}) \Rightarrow \mathtt{bool}$

- and the constructors, discriminators and selectors of the datatype.

  consts

`nil`	::	α `llist`
`op #`	::	$\alpha \to \alpha\ \mathtt{llist} \to \alpha\ \mathtt{llist}$
`is_nil`	::	$\alpha\ \mathtt{llist} \to \mathtt{tr}$
`is_#`	::	$\alpha\ \mathtt{llist} \to \mathtt{tr}$
`hd`	::	$\alpha\ \mathtt{llist} \to \alpha$
`tl`	::	$\alpha\ \mathtt{llist} \to \alpha\ \mathtt{llist}$

  Additionally, macros allowing to formulate case distinctions on the datatype in a pleasant way are produced:

```
translations case l of nil ⇒ v | x#xs ⇒ w == llist_case‘v‘(Λx xs. w)‘l
```

Analogous types and constants are generated for other datatype definitions.

[3] For simplicity, we have chosen the same symbol as used for the HOL datatype `list`.
[4] To be exact, the **instance** declaration is replaced by an **arities** declaration, which is not described in this paper.

5.1.2 Semantics

Let us now reveal the details of how the above types and constants are defined. All definitions of this section are interna for the user of the package, he or she does not have to understand them.

The (`abs`, `rep`) pair is required to be an isomorphism between the (abstract) left and (representing) right hand side of the defining equation.

```
rules   abs_iso    llist_rep`(llist_abs`x) = x
        rep_iso    llist_abs`(llist_rep`x) = x
```

Together with the axiom *reach* given below, this yields an elegant characterization of the datatype as the least solution of its defining equation. The soundness of this construction, i.e. the existence and uniqueness of the semantic model, has been proved externally to HOLCF by category-theoretic means.

All other functions are based solely on the `abs` and `rep` functions and the functions defined for the general domain constructions described in §4.3. That is, one-element domains are used for dummy arguments of constant constructors, the truth values serve as results of the discriminators, the disjoint sum represents the case distinction between different constructors, the non-strict product handles mutually recursive datatypes, and lifted domains are employed in non-strict constructors.

The `case` functional applies one of its argument functions, depending on case analysis, to a datatype element. With this auxiliary functional and the basic domain constructions just mentioned, the definition of the constructors, discriminators, and selectors is rather straightforward.

```
defs  llist_case ≡ Λc f z. case llist_rep`z of sinl`x ⇒ (Λdummy. c)
                           | sinr`y ⇒ ssplit`(Λx xs. f`x`(fup`ID`xs))`y

      nil    ≡        llist_abs`(sinl`ONE)
      op #   ≡ Λx xs. llist_abs`(sinr`(|x,up`xs|))
      is_nil ≡ Λz. case z of nil ⇒ TT | x#xs ⇒ FF
      is_#   ≡ Λz. case z of nil ⇒ FF | x#xs ⇒ TT
      hd     ≡ Λz. case z of nil ⇒ ⊥  | x#xs ⇒ x
      tl     ≡ Λz. case z of nil ⇒ ⊥  | x#xs ⇒ xs
```

The `copy` functional copies a datatype element, except that it applies its argument function to each (just one here) occurence of recursion. The `take` functional denotes the n-times repeated application of the `copy` functional to the completely undefined function, yielding finite approximations of a datatype element up to depth n.

```
defs  llist_copy ≡ Λf z. case z of nil ⇒ nil | x#xs ⇒ x#(f`xs)
      llist_take ≡ λn. iterate n llist_copy ⊥
```

The least fixpoint of the `copy` functional can be understood as the limit of the `take` functional for increasing n, for which `fix`llist_copy` $\sqsubseteq$ `ID` holds. The axiom *reach* requires this fixpoint to be already as strongly defined as the identity on the datatype. In this way, the initial (i.e., least) solution of the defining domain equation is described.

```
rules   reach    fix`llist_copy`x = x
```

A datatype element is finite iff it can be reached by some finite approximation. Bisimulations on the datatype are characterized as binary relations describing identical behavior of a pair of elements.

```
defs  llist_finite ≡ λx. ∃n. llist_take n`x = x
      llist_bisim  ≡ λR. ∀x x'. R x x' ⟶ x = ⊥    ∧ x' = ⊥     ∨
                                         x = nil  ∧ x' = nil   ∨
            (∃y ys ys'. y ≠ ⊥ ∧ R ys ys' ∧ x = y#ys ∧ x' = y#ys')
```

5.1.3 Theorems

Based solely on the two isomorphism axioms given above, the characteristic properties of the `case`, `copy`, and `take` functionals are proved for internal use. Together with these properties (not shown here), the package proves a bunch of theorems exhibiting the properties of the user-relevant functions. They include the characteristic properties typically needed in proofs on the datatype:

- exhaustion and case distinction

```
 x = ⊥ ∨ x = nil ∨ (∃y ys. x = y#ys ∧ y ≠ ⊥)
⟦x = ⊥ ⟹ P; x = nil ⟹ P; ⋀y ys. ⟦x = y#ys; y ≠ ⊥⟧ ⟹ P⟧ ⟹ P
```

- strictness and definedness properties, together with the characteristic equations for selectors and discriminators,

```
⊥#xs = ⊥,   nil         ≠ ⊥,    x ≠ ⊥ ⟹ x#xs          ≠ ⊥
            hd` ⊥       = ⊥,             tl` ⊥         = ⊥
            hd` nil     = ⊥,             hd`(x#xs)     = x
            tl` nil     = ⊥,    x ≠ ⊥ ⟹ tl`(x#xs)     = xs
            is_nil` ⊥   = ⊥,             is_#  ` ⊥     = ⊥
  x ≠ ⊥ ⟹ is_nil` x   ≠ ⊥,    x ≠ ⊥ ⟹ is_#  ` x     ≠ ⊥
            is_nil` nil = TT,   x ≠ ⊥ ⟹ is_nil`(x#xs) = FF
            is_#  ` nil = FF,   x ≠ ⊥ ⟹ is_#  `(x#xs) = TT
```

- and the distinctness and injectivity of the constructors

```
          ¬ nil ⊑ x#xs,   x ≠ ⊥ ⟹ ¬ x#xs ⊑ nil
            nil  ≠ x#xs,              x#xs  ≠ nil
⟦x#xs ⊑ y#ys; x ≠ ⊥; y ≠ ⊥⟧ ⟹ x ⊑ y ∧ xs ⊑ ys
⟦x#xs  = y#ys; x ≠ ⊥; y ≠ ⊥⟧ ⟹ x  = y ∧ xs  = ys
```

In addition, exploiting the reach axiom, the package proves the (structural) induction and coinduction principles of the new datatype. This process takes some intermediate steps, e.g. an induction rule for finite elements, which may be useful for special applications. Here, as the `llist` datatype is infinite in general, just a trivial "finiteness" theorem is generated. For the same reason, the full induction rule requires an admissibility condition here.

```
take_lemma (⋀n. llist_take n`xs = llist_take n`ys) ⟹ xs = ys
finite     ¬ llist_finite xs ∨ llist_finite xs
finite_ind ⟦P ⊥;  P nil;
           ⋀y ys. ⟦y ≠ ⊥; P ys⟧ ⟹ P (y#ys)⟧ ⟹ P (llist_take n`xs)
ind        ⟦adm P;  P ⊥;  P nil;
           ⋀y ys. ⟦y ≠ ⊥; P ys⟧ ⟹ P (y#ys)⟧ ⟹ P xs
```

The induction rule is useful particularly in cases where the admissibility of `P` can be proven automatically. If this is not the case, then our experience suggests not to attempt to prove admissibility directly, as this often becomes the hardest part of the entire proof. If possible, one should instead resort to other proof principles that do not need admissibility. These are essentially the take lemma (as given above) and the coinduction principle:

```
coind    [|llist_bisim R; R x y|] ==> x = y
```

Examples demonstrating the typical use of the take lemma, the induction, and the coinduction principles are given in §6.1.

5.2 Mutual recursion

As an example of mutual recursion, we formalize the datatype of first-order terms with constructors for variables and function applications:

```
domain term  =  Var name | App  name terms
and    terms =  Nil      | Cons term terms
```

Ideally one would write something like `App name (term list)`, but since the package does not cater for indirect recursion, we have simulated this with mutual recursion, which is always possible. Note that this appears to be the first natural formalization of terms in domain theory: previous constructions (Paulson, 1985; Agerholm, 1995) were restricted to constants and binary functions in order to bypass the problem of mutual or indirect recursion.

The entities produced by the package are the same as those for non-mutually recursive datatypes, with some differences reflecting the recursion between terms and term lists. For example, the `copy` functional is constructed as a pair here:

```
term_terms_copy ≡ Λf. <term_copy`f, terms_copy`f>
```

Also some of the proof rules, like induction and coinduction, can only be given simultaneously for all concerned types:

```
coind [|term_terms_bisim R; (fst R) x1 x1'; (snd R) x2 x2'|]
      ==> x1 = x1' ∧ x2 = x2'
ind   [|                                        P1  ⊥;
        ⋀n  .  n ≠ ⊥                        ==> P1 (Var`n);
        ⋀n t. [|n ≠ ⊥; t ≠ ⊥;       P2 t|] ==> P1 (App`n`t);
                                               P2  ⊥;
                                               P2  Nil;
        ⋀t s. [|t ≠ ⊥; s ≠ ⊥; P1 t; P2 s|] ==> P2 (Cons`t`s)|]
      ==> P1 x1 ∧ P2 x2
```

6 Applications

In this section we sketch how HOLCF has been used for non-trivial applications. In §6.1 we give functional programming examples by defining functions on lazy lists and explaining proof support for them. In §6.2 this theory of sequences is used

to reason about a model of reactive, distributed systems, namely I/O automata (Lynch & Tuttle, 1989). The last subsection §6.3 presents a denotational semantics for a simple imperative, sequential programming language.

6.1 Functional programming with lazy lists

Typical recursive functions on the lazy list datatype defined in §5 include

```
consts  map    :: (α → β ) → α llist → β llist
        filter :: (α → tr) → α llist → α llist
        iter   :: (α → α ) → α       → α llist
```

which are defined by fixpoint constructions like

```
defs    map ≡ fix‘(Λh f l. case l of nil ⇒ nil | x#xs ⇒ f‘x#h‘f‘xs)
```

In the sequel, we derive some well-known properties of these functions in order to demonstrate the application of the most important proof principles.

We can obtain the (recursive) equations characterizing these functions from their fixpoint definitions in a rather generic way using a tactic which essentially needs the fixpoint theorem and the continuity of the body of the fixpoint. In the case of `map` the latter is trivially fulfilled because the function `f` and the case distinction are continuous by their definition using the type constructor →.

```
          map‘f‘ ⊥     = ⊥
          map‘f‘ nil   = nil
x ≠ ⊥ ⟹ map‘f‘(x#xs) = f‘x#map‘f‘xs
          filter‘P‘ ⊥     = ⊥
          filter‘P‘ nil   = nil
x ≠ ⊥ ⟹ filter‘P‘(x#xs) = If P‘x then x#filter‘P‘xs else filter‘P‘xs
          iter‘f‘x = x#iter‘f‘(f‘x))
          iter‘f‘⊥ = ⊥
```

6.1.1 A connection between `map` *and* `filter`

As our first example proof, consider the (kind of) commutation of `filter` and `map` as functions over lazy lists:

```
P‘⊥ = ⊥ ⟹ filter‘P oo map‘f = map‘f oo filter‘(P oo f)
```

The strictness premise is necessary because `#` is strict in its first argument.

The proof of this property is by structural induction on the list involved. So, after stating the goal and exploiting the extensionality of continuous functions in order to obtain an explicit argument `x` to induct over, we apply the induction rule, `llist.ind`, instantiated to this `x`. We now have to prove the four subgoals

```
1.  P‘⊥ = ⊥ ⟹ adm (λu. filter‘P‘(map‘f‘u) = map‘f‘(filter‘(P oo f)‘u))
2.  P‘⊥ = ⊥ ⟹      filter‘P‘(map‘f‘⊥)     = map‘f‘(filter‘(P oo f)‘⊥)
3.  P‘⊥ = ⊥ ⟹      filter‘P‘(map‘f‘nil)   = map‘f‘(filter‘(P oo f)‘nil)
4. ⟦P‘⊥ = ⊥; a ≠ ⊥;  filter‘P‘(map‘f‘l)     = map‘f‘(filter‘(P oo f)‘l)⟧
   ⟹                filter‘P‘(map‘f‘(a#l)) = map‘f‘(filter‘(P oo f)‘(a#l))
```

where the first three are proved automatically by the simplifier. This involves the admissibility check described in 4.2.4 and simplifications using the characteristic equations of `map` and `filter`. By further rewriting with these equations, the fourth subgoal becomes:

```
⟦P‘⊥ = ⊥; a ≠ ⊥; filter‘P‘(map‘f‘l) = map‘f‘(filter‘(P oo f)‘l)⟧ ⟹
(case f‘a#map‘f‘l of nil ⇒ nil
 | x#xs ⇒ If P‘x then x#filter‘P‘xs else filter‘P‘xs) =
 map‘f‘(If P‘(f‘a) then a#filter‘(P oo f)‘l else filter‘(P oo f)‘l)
```

Now we discriminate on `f‘a = ⊥` and solve the trivial case by simplification. With the new premise `f‘a ≠ ⊥`, and equipped with a case splitting tool for the condition of the `If` construct, the simplifier is able to finish the rest of the proof.

Within Isabelle/HOLCF, this is an easy six-lines proof where the only non-automatic steps are induction and case splitting.

6.1.2 A connection between `map` *and* `iter`

Our second example involves the equality of the infinite list generated by `iter‘f` out of the seed `f‘x` and the result of mapping `f` over the list `iter‘f‘x`, both of which produce the lazy list `f‘x#f‘(f‘x)#f‘(f‘(f‘x))#...`. With the assumption that `f` is strict, the equation should hold for any `x`.

```
f‘⊥ = ⊥ ⟹ iter‘f‘(f‘x) = map‘f‘(iter‘f‘x)
```

We conduct the proof first by coinduction, i.e. with the rule `llist.coind`. Thus we have to give a suitable bisimulation, where here it turns out that the canonical choice `R = λm n. ∃z. m = iter‘f‘(f‘z) ∧ n = map‘f‘(iter‘f‘z)` suffices. The main effort of the proof is to show that this is indeed a bisimulation:

After unfolding the definition of `llist_bisim` and after some simple predicate-calculus steps, we have to discriminate on `z = ⊥` and `f‘z = ⊥`. The remainder of the proof is straightforward but tedious, applying strictness properties and involving about eleven unfolding, simplifying, and pure predicate-calculus steps.

An alternative proof that turns out to be much shorter is to apply the take lemma for lazy lists, which gives the subgoal

```
llist_take n‘(iter‘f‘(f‘x)) = llist_take n‘(map‘f‘(iter‘f‘x))
```

In order to prove this formula by induction on `n`, we have to strengthen it by ∀-quantification over `x`. The base case is trivial, because `llist_take 0‘x = ⊥`. Then we unfold both occurrences of `iter` in the inductive step, and obtain

```
⟦f‘⊥ = ⊥; ∀x. llist_take n‘(iter‘f‘(f‘x)) = llist_take n‘(map‘f‘(iter‘f‘x))⟧
⟹ llist_take (Suc n)‘(f‘x#iter‘f‘(f‘(f‘x))) =
    llist_take (Suc n)‘(map‘f‘(x#iter‘f‘(f‘x)))
```

As above, we discriminate on `x = ⊥` in order to distribute `map` over `#`. Then the simplifier solves the subgoal applying strictness and the induction hypothesis.

6.2 Theory of I/O automata with lifted lazy lists

In this subsection we briefly describe a model of I/O automata in a mixed HOL and HOLCF formalization. For this model we construct an instance of the lazy lists

datatype introduced in 5.1 that provides a good integration of pure HOL terms into HOLCF. See (Devillers *et al.*, 1997) for a detailed analysis of these lists in comparison to alternative formalizations.

6.2.1 Lazy lists with lifted elements

The definition of α `llist` requires the element type α to be in type class `pcpo`. However, for our application it is more convenient to handle the elements in a total fashion, i.e. as types of class `term`. Therefore we define a new type of lazy lists that allows elements to be of any HOL type using the constructor `lift`:

types α `Llist = (`α`::term lift) llist`

Furthermore, a new "cons"-operator [5] for HOL elements is introduced:

consts `##` :: α`::term` $\Rightarrow$ α `Llist` $\rightarrow$ α `Llist` `(infix)`
defs `s ## a` $\equiv$ `Def a # s`

Operations on `Llist` profit from this integration of HOL and HOLCF types. For example, it is now possible to define filtering with a total predicate `P::`α $\Rightarrow$ `bool` as follows:

consts `Filter` :: (`P::`α`::term` $\Rightarrow$ `bool)` $\Rightarrow$ α `Llist` $\rightarrow$ α `Llist`
defs `Filter P` $\equiv$ `filter‘(flift2 P)`

The equalities for `Filter` follow from those for `filter` and from the equation `If Def b then A else B = if b then A else B`:

```
Filter P‘⊥       = ⊥
Filter P‘nil     = nil
Filter P‘(x##xs) = if P x then x##(Filter P‘xs) else Filter P‘xs
```

The `map` operation is modified analogously:

consts `Map` :: (`P::`α`::term` $\Rightarrow$ β`::term)` $\Rightarrow$ α `Llist` $\rightarrow$ β `Llist`
defs `Map f` $\equiv$ `map‘(flift2 f)`

These examples demonstrate the general advantages of lazy lists whose argument types are lifted (flat) domains:

- Elements of lazy lists that do not need support for infinity or undefinedness can be handled in the simpler logic HOL — see e.g. the total predicate `P` — and lifted to domains as late as possible. Theories and libraries about arbitrary HOL types can be reused.
- Not only *elements* of lazy lists but also the *operations* on the lazy lists themselves profit from pushing as much as possible into HOL. For example, the last equation for `Filter` uses the two-valued operator `if then else` instead of its three-valued counterpart `If then else`. In general, reasoning about lazy lists is much more efficient using the built-in tableaux calculus and the simplifier which are tailored for two-valued logic. An analogous calculus for a three-valued logic would require a completely new and different design.
- The `Def` tag in the definition of `##` ensures that all elements are defined. Therefore the nasty precondition `x` $\neq \bot$ for `filter` is no longer needed for `Filter`. In general, this saves a lot of $\bot$ case distinctions.

[5] As before for `#`, the symbol `##` does not reflect the actual code.

6.2.2 I/O automata

I/O-Automata (Lynch & Tuttle, 1989) are a model for reactive, distributed systems. Significant parts of the theory of these automata has been formalized in Isabelle/HOLCF (Nipkow & Slind, 1995; Müller & Nipkow, 1997; Müller, 1998a; Müller, 1998b). In the sequel we present only a small fragment of this formalization, focussing on the communication histories of I/O-automata which are described by lazy lists of type `Llist`. From now on α and σ describe types of class `term`.

An action signature models different types of actions and is described by the type

types α `signature = ` α `set * ` α `set * ` α `set`

where the first, second and third component may be extracted with the selectors `inputs`, `outputs` and `internals`. Furthermore, the externally visible interface of an action signature is denoted by

defs `externals S ≡ inputs S ∪ outputs S`

The three components of an action signature have to be disjoint.

An I/O automaton is a triple of an action signature, a set of start states, and a set of transition triples, described by the type

types `(`α`,`σ`)ioa = `α `signature * `σ `set * (`σ ` * `α ` * `σ`)set`

The members of this triple are extracted by `sig_of`, `starts_of` and `trans_of`. Isabelle's syntax mechanism is used to abbreviate `externals o sig_of` to `ext` and to write `s -a-A⟶ t` for a step `(s,a,t) ∈ trans_of A`.

The set of states reachable by an I/O automaton `A` is defined inductively as the least set of states satisfying the following two rules:

```
 s ∈ starts_of A                ⟹ s ∈ reachable A
⟦s ∈ reachable A; s -a-A⟶ t⟧ ⟹ t ∈ reachable A
```

6.2.3 Behaviours of I/O automata

In the sequel we focus on notions to describe the behavior of I/O automata over time, namely *executions* and *traces*. A finite or infinite alternating sequence of states and actions representing steps of an I/O automaton `A` is called an *execution fragment* of `A`. An *execution* is an execution fragment beginning with a start state. *Traces* are the subsequences of executions consisting of their external actions only, and therefore describe the visible behavior of an automaton.

Executions are modeled by a pair of a start state and a lazy list of action/state pairs, lifted to a flat domain:

types `(`α`,`σ`)execution = `σ ` * (`α ` * `σ`)Llist`

A predicate identifies those lists that represent an execution fragment:

```
consts is_exec_frag :: (α,σ)ioa ⇒ (α,σ)execution ⇒ bool
defs   is_exec_frag A ex ≡ is_exec_fragC A‘ (snd ex) (fst ex) ≠ FF
```

It is realized by a continuous function

consts `is_exec_fragC :: (`α`,`σ`)ioa ⇒ (`α ` * `σ`)Llist ⇒ `σ ` → tr`

which "runs down" the lazy list checking if all of its transitions are steps of `A`. The predicate `is_exec_frag` is true if `is_exec_fragC` terminates and returns `TT` (for finite executions) or if it does not terminate (for infinite executions). We define the operation `is_exec_fragC` as a fixpoint; the following rewrite rules have been derived from the definition automatically:

```
is_exec_fragC A‘⊥           s = ⊥
is_exec_fragC A‘nil         s = TT
is_exec_fragC A‘((a,t)##ex) s = Def s-a-A⟶t andalso is_exec_fragC A‘ex t
```

Using the lemmas `x ≠ ⊥ ⟹ (x andalso y ≠ FF) = (x ≠ FF ∧ y ≠ FF)` and `(Def x ≠ FF) = x` we obtain the corresponding rules for `is_exec_frag`:

```
is_exec_frag A (s,⊥)
is_exec_frag A (s,nil)
is_exec_frag A (s,(a,t)##ex) = s-a-A⟶t ∧ is_exec_frag A (t,ex)
```

Analogous to the `Filter` example in the previous section this shows how to handle as much as possible with two-valued logic in recursive operations (∧ instead of `andalso`).

Finally, executions and traces are defined as follows:

```
defs   executions A ≡ {ex. (fst ex) ∈ starts_of A ∧ is_exec_frag A ex}
       traces     A ≡ {Filter (λa. a ∈ ext A) ‘ (Map fst ex).
                       ∃s. (s,ex) ∈ executions A }
```

6.2.4 Refinement of I/O automata

An I/O automaton `C` *implements* another automaton `A` iff `traces C ⊆ traces A`. Such implementation relations are shown by the use of *refinement mappings*. In this paper we only consider *weak refinement mappings*. Such a mapping `f` is a function from the states of the concrete automaton `C` to the states of the abstract automaton `A` that has to fulfill two requirements: First, start states of `C` have to be mapped to start states of `A`. Second, for every reachable step `s -a-C⟶ t` of `C` the corresponding step `(f s) -a-A⟶ (f t)` of `A` has to exist if `a` is an external action, otherwise `A` has to stutter, i.e. `f s = f t`.

```
defs  is_weak_ref_map f C A ≡ (∀s ∈ starts_of C. f s ∈ starts_of A) ∧
        (∀s t a. s ∈ reachable C ∧ s -a-C⟶ t ⟶
                 if a ∈ ext A then (f s) -a-A⟶ (f t) else f s = f t)
```

The correctness of weak refinement mappings is established by the following theorem which has been proved in HOLCF:

```
⟦is_weak_ref_map f C A; ext C = ext A⟧ ⟹ traces C ⊆ traces A
```

Note the following important methodological point here: This theorem has been proved making heavy use of HOLCF because it involves recursively defined lazy lists. However, the predicate `is_weak_ref_map` is completely defined in the simpler logic HOL. Therefore refinement proofs in applications can be done in HOL, whereas the more powerful but also more complicated domain theory is utilized for the meta theory of I/O automata only. This is a remarkable advantage of the decision to use lazy lists with *lifted* elements.

6.3 Denotational semantics

Historically, the main motivation for developing domain theory and hence LCF was to provide a formal foundation for denotational semantics. We conclude our list of applications by returning to those historic roots. Below we give a denotational semantics of IMP, a simple imperative programming language with while loops (Winskel, 1993). For a full coverage of operational, denotational and axiomatic semantics of IMP in Isabelle see (Nipkow, 1996; Nipkow, 1998b).

IMP is based on two types which are not further specified: (storage) *locations* `loc` and *values* `val`. On top of these we define

```
types  state =  loc   ⇒ val
       aexp  =  state ⇒ val
       bexp  =  state ⇒ bool
```

The type `state` is the usual mapping from locations to values. The types `aexp` and `bexp` formalize arithmetic and boolean *expressions*. Note that we have taken a semantic short-cut here: rather than defining the syntax of expressions, we work directly in terms of their semantics. Bypassing syntax in favour of semantics means that concrete expressions look a bit unusual. For example, $x+1$ becomes `λs.s(x)+1`. It is routine to modify the parser and pretty printer to translate between the two forms automatically. We ignore these syntactic issues and focus on the semantic side of things.

The abstract syntax of *commands*, i.e. statements, of IMP is defined as a HOL datatype. The constructors represent assignment, sequential composition, conditional, and while loop:

```
datatype com = Skip
             | Assign loc aexp
             | Seq    com com
             | Cond   bexp com com
             | While  bexp com
```

The denotational semantics of a command is a partial function from states to states. For while loops, it is defined as a least fixpoint. Thus we would like it to be of type `state → (state)lift`, but this does not quite work: the domain of the function, i.e. `state`, is not a cpo, and hence `state → (state)lift` is not a pcpo, which means that `fix` is inapplicable. Of course we could define `state` as `loc → val`, but this is hardly natural because `state` represents the machine store, which is a total function. Therefore we explicitly turn `state` into a discrete cpo. This leads to the following function `D` that maps syntax to semantics and is defined by primitive recursion (a hallmark of denotational semantics):

```
consts  D :: com ⇒ (state)discr → (state)lift
primrec
  D(Skip)          =  (Λs. Def(undiscr s))
  D(Assign x a)    =  (Λs. Def((undiscr s)[a(undiscr s)/x]))
  D(Seq c1 c2)     =  (dlift(D c2) oo (D c1))
  D(Cond b c1 c2)  =  (Λs. if b(undiscr s) then (D c1)`s else (D c2)`s)
  D(While b c)     =  fix`(Λw s. if b(undiscr s) then (dlift w)`((D c)`s)
                                 else Def(undiscr s))
```

Note that `_[_/_]` is the pointwise update of functions. The auxiliary functional `dlift` :: `((`α::`term)discr` $\to$ β::`pcpo)` $\Rightarrow$ `((`α`)lift` $\to$ β`)` lifts a function from a discrete cpo to one from a lifted cpo:

defs `dlift f` $\equiv$ `Λx. case x of Undef` $\Rightarrow$ $\bot$ `| Def(y)` $\Rightarrow$ `f‘(Discr y)`

Thanks to the infrastructure for type `lift`, `dlift f` is automatically shown to be continuous for every `f`.

The above definition of `D` is pretty much what the textbook says, except that the explicit bijection `undiscr` spoils the view a little. An early version of `D` (Nipkow, 1996) had the type `com` $\Rightarrow$ `(state)lift` $\to$ `(state)lift`. At the time this was necessary because HOLCF was based only on class `pcpo` and discrete cpos were not available. This meant that the definition actually looked simpler, e.g. `(D c2) oo (D c1)` instead of `dlift(D c2) oo (D c1)`. However, this complicated some proofs considerably and we abandoned this design.

Let us now examine the proof of a typical theorem about IMP, namely the soundness of Hoare's proof rule for while loops:

$$\frac{\{A\}\ c\ \{A\}}{\{A\}\ \texttt{while}\ b\ \texttt{do}\ c\ \{A \wedge \neg b\}}$$

We model assertions just like boolean expressions:

types `assn = state` $\Rightarrow$ `bool`

The validity of a Hoare-triple $\{A\}\ c\ \{B\}$, where A and B are assertions and c a command, is defined in HOLCF by a constant `hoare_valid` of the obvious type `assn` $\Rightarrow$ `com` $\Rightarrow$ `assn` $\Rightarrow$ `bool`. We use Isabelle's flexible syntax facilities to write $\models$ `{A} c {B}` instead of `hoare_valid A c B`. The definition itself is classical:

defs $\models$ `{A} c {B}` $\equiv$ `∀s t. A s ∧ (D c)‘(Discr s) = Def t ⟶ B t`

Soundness of the while rule now becomes:

$\models$ `{A} c {A}` $\Longrightarrow$ $\models$ `{A} While b c {λs. A s ∧ ¬b s}`

where `λs. A s ∧ ¬b s` is the functional encoding of $A \wedge \neg b$. Unfolding the definitions of $\models$ and `D` means we have to prove:

```
∀s t. A s ∧ D c‘(Discr s) = Def t ⟶ A t
⟹ ∀s t. A s ∧ fix‘(Λw s. if b (undiscr s) then dlift w‘(D c‘s)
                                          else Def (undiscr s))‘(Discr s) = Def t
          ⟶ A t ∧ ¬b t
```

The conclusion is proved by fixpoint induction (see §4.2.4), which leaves us with three subgoals. We examine them one by one.

Admissibility is proved automatically:

`adm(λu. ∀s t. A s ∧ u‘(Discr s) = Def t ⟶ A t ∧ ¬b t)`

It should be noted that this relies on the fact that `u‘(Discr s)`, the only occurrence of `u`, is of type `state lift`, which is flat and hence chain-finite. This is one of the not-so-rare examples that require the extended admissibility test described at the end of §4.2.4.

The base case

```
∀s t. A s ∧ ⊥‘(Discr s) = Def t ⟶ A t ∧ ¬b t
```

is proved by simplification because `⊥ = Def t` is a contradiction.
In the induction step we have to prove

```
∀s t. A s ∧ (if b s then dlift x‘(D c‘(Discr s)) else Def s) = Def t
       ⟶ A t ∧ ¬b t
```

from the induction hypothesis

```
∀s t. A s ∧ x‘(Discr s) = Def t ⟶ A t ∧ ¬b t
```

Simplification combined with case distinction and predicate calculus reasoning solves the induction step.

7 Conclusion

HOLCF started life with Regensburger's Ph.D. and has been enhanced ever since. By now it comprises about 1000 lines of theories and 9000 lines of proofs. Apart from conducting the proofs themselves, the main challenges were: getting the structure right, hiding as much of domain theory as possible, and facilitating the transition between HOL and HOLCF. Hiding domain theory traditionally means automating trivial or awkward proof steps that arise from domain theory, e.g. $\bot$-cases and admissibility requirements. In the case of HOLCF, it additionally means hiding the encoding of domain theory in HOL, in particular explicit continuity checks (§4.2.3). The transition between HOL and HOLCF (§4.3.2) is crucial for larger developments that are conducted both in HOL and in HOLCF.

We have successfully met those challenges and HOLCF has become a logic for domain theory that is more expressive than LCF and suitable for large applications (Müller, 1998b). In fact, most of the facilities that go beyond LCF, e.g. the extended admissibility check (§4.2.4), were prompted by applications. HOLCF is now in a stable state and provides an excellent platform for formal developments involving partial functions or infinite objects.

Acknowledgments. We sincerely thank Franz Regensburger for developing the initial version of HOLCF and commenting on our further developments. Two anonymous referees helped us with their critical and insightful reports to turn a draft into a paper.

References

Agerholm, Sten. (1994a). *Formalising a model of the λ-calculus in HOL-ST.* Tech. rept. 354. University of Cambridge Computer Laboratory.

Agerholm, Sten. (1994b). *A HOL basis for reasoning about functional programs.* Ph.D. thesis, Department of Computer Science, University of Aarhus. BRICS report RS-94-44.

Agerholm, Sten. (1995). LCF examples in HOL. *The computer journal*, **38**, 121–130.

Audebaud, Philippe. (1991). Partial objects in the calculus of constructions. *Pages 86–95 of: 6th IEEE Symp. Logic in Computer Science.* IEEE Computer Society Press.

Bartels, Falk, Dold, Axel, von Henke, Friedrich, Pfeifer, Holger, & Rueß, Harald. (1996). *Formalizing fixed-point theory in PVS.* Tech. rept. Universität Ulm.

Constable, Robert, & Smith, Scott. (1987). Partial objects in constructive type theory. *Pages 183–193 of: 2nd IEEE Symp. Logic in Computer Science.* IEEE Computer Society Press.

Crary, Karl. (1998). Admissibility of fixpoint induction over partial types. *Pages 270–285 of:* Kirchner, C, & Kirchner, H. (eds), *Automated deduction — CADE-15.* Lect. Notes in Comp. Sci., vol. 1421. Springer-Verlag.

Devillers, Marco, Griffioen, David, & Müller, Olaf. (1997). Possibly infinite sequences in theorem provers: A comparative study. *Pages 89–104 of:* Gunter, E.L., & Felty, A. (eds), *Theorem proving in higher order logics.* Lect. Notes in Comp. Sci., vol. 1275. Springer-Verlag.

Gordon, M.C.J., Milner, Robin, & Wadsworth, C.P. (1979). *Edinburgh LCF: a mechanised logic of computation.* Lect. Notes in Comp. Sci., vol. 78. Springer-Verlag.

Gordon, M.J.C., & Melham, T.F. (eds). (1993). *Introduction to HOL: a theorem-proving environment for higher order logic.* Cambridge University Press.

Hudak, Paul, Peyton Jones, Simon, & Wadler, Philip. (1992). Report on the programming language Haskell: A non-strict, purely functional language. *ACM SIGPLAN Notices*, **27**(5). Version 1.2.

Lynch, Nancy, & Tuttle, Mark. (1989). An introduction to Input/Output automata. *Cwi quarterly*, **2**(3), 219–246.

Müller, Olaf. (1998a). I/O automata and beyond — temporal logic and abstraction in Isabelle. *Pages 1–18 of:* Grundy, J., & Newey, M. (eds), *Theorem proving in higher order logics (TPHOLs'98).* Lect. Notes in Comp. Sci., vol. 1479. Springer-Verlag.

Müller, Olaf. (1998b). *A verification environment for I/O-automata based on formalized meta-theory.* Ph.D. thesis, Institut für Informatik, Techn. Univ. München.

Müller, Olaf, & Nipkow, Tobias. (1997). Traces of I/O automata in Isabelle/HOLCF. *Pages 580–594 of:* Bidoit, M., & Dauchet, M. (eds), *Tapsoft'97: Theory and practice of software development.* Lect. Notes in Comp. Sci., vol. 1214. Springer-Verlag.

Nipkow, Tobias. (1993). Order-sorted polymorphism in Isabelle. *Pages 164–188 of:* Huet, G., & Plotkin, G. (eds), *Logical environments.* Cambridge University Press.

Nipkow, Tobias. (1996). Winskel is (almost) right: Towards a mechanized semantics textbook. *Pages 180–192 of:* Chandru, V., & Vinay, V. (eds), *Foundations of software technology and theoretical computer science.* Lect. Notes in Comp. Sci., vol. 1180. Springer-Verlag.

Nipkow, Tobias. (1998a). *Isabelle/HOL. The Tutorial.* Unpublished Manuscript. Available at `www.in.tum.de/~nipkow/pubs/HOL.html`.

Nipkow, Tobias. (1998b). Winskel is (almost) right: Towards a mechanized semantics textbook. *Formal aspects of computing*, **?** To appear.

Nipkow, Tobias, & Slind, Konrad. (1995). I/O automata in Isabelle/HOL. *Pages 101–119 of:* Dybjer, P., Nordström, B., & Smith, J. (eds), *Types for proofs and programs.* Lect. Notes in Comp. Sci., vol. 996. Springer-Verlag.

Oheimb, David von. (1995). *Datentypspezifikationen in Higher-Order LCF.* M.Phil. thesis, Technische Universität München.

Paulson, Lawrence C. (1985). Verifying the unification algorithm in LCF. *Science of computer programming*, **5**, 143–169.

Paulson, Lawrence C. (1987). *Logic and computation.* Cambridge University Press.

Paulson, Lawrence C. (1994). *Isabelle: A generic theorem prover.* Lect. Notes in Comp. Sci., vol. 828. Springer-Verlag.

Paulson, Lawrence C. (1997). Generic automatic proof tools. Veroff, R. (ed), *Automated reasoning and its applications.* MIT Press. Also Report 396, Computer Laboratory, University of Cambridge.

Regensburger, Franz. (1994). *HOLCF: Eine konservative Erweiterung von HOL um LCF.* Ph.D. thesis, Technische Universität München.

Regensburger, Franz. (1995). HOLCF: Higher Order Logic of Computable Functions. *Pages 293–307 of:* Schubert, E.T., Windley, P.J., & Alves-Foss, J. (eds), *Higher order logic theorem proving and its applications.* Lect. Notes in Comp. Sci., vol. 971. Springer-Verlag.

Slind, Konrad. (1996). Function definition in higher order logic. *Pages 381–397 of:* von Wright, J., Grundy, J., & Harrison, J. (eds), *Theorem proving in higher order logics.* Lect. Notes in Comp. Sci., vol. 1125. Springer-Verlag.

Slotosch, Oscar. (1997a). Higher order quotients and their implementation in isabelle hol. *Pages 291–306 of:* Gunter, E.L., & Felty, A. (eds), *Theorem proving in higher order logics.* Lect. Notes in Comp. Sci., vol. 1275. Springer-Verlag.

Slotosch, Oskar. (1997b). *Refinements in HOLCF: Implementation of interactive systems.* Ph.D. thesis, Institut für Informatik, TU München.

Wenzel, Markus. (1997). Type classes and overloading in higher-order logic. *Pages 307–322 of:* Gunter, E.L., & Felty, A. (eds), *Theorem proving in higher order logics.* Lect. Notes in Comp. Sci., vol. 1275. Springer-Verlag.

Winskel, Glynn. (1993). *The formal semantics of programming languages.* MIT Press.