HOLCF: Higher Order Logic of Computable Functions

Franz Regensburger
regensbu@informatik.tu-muenchen.de

Technische Universität München

Abstract. This paper presents a survey of HOLCF, a higher order logic of computable functions. The logic HOLCF is based on HOLC, a variant of the well known higher order logic HOL, which offers the additional concept of type classes.
HOLCF extends HOLC with concepts of domain theory such as complete partial orders, continuous functions and a fixed point operator. With the help of type classes the extension can be formulated in a way such that the logic LCF constitutes a proper sublanguage of HOLCF. Therefore techniques from higher order logic and LCF can be combined in a fruitful manner avoiding drawbacks of both logics. The development of HOLCF was entirely conducted within the Isabelle system.

1 Introduction

This paper presents a survey of HOLCF, a higher order logic of computable functions. The logic HOLCF is based on HOLC, a variant of the well known higher order logic HOL [GM93], which offers the additional concept of type classes.

HOLCF extends HOLC with concepts of domain theory such as complete partial orders, continuous functions and a fixed point operator. With the help of type classes the extension can be formulated in a way such that the logic LCF [GMW79, Pau87] constitutes a proper sublanguage of HOLCF. Therefore techniques from higher order logic and LCF can be combined in a fruitful manner avoiding drawbacks of both logics.

The logic HOLC is implemented in the logical framework Isabelle [Pau94] and the development of HOLCF was conducted within the Isabelle system, too. The syntax, semantics and proof rules of HOLC together with the development of HOLCF are described in full detail in my thesis [Reg94].

In parallel with my development of HOLCF Sten Agerholm developed the HOL-CPO [Age94] system on the basis of Gordon's HOL System. Although the overall aim of the two theses is the same, namely the combination of HOL and LCF, the techniques used in the two approaches differ in many aspects. The availability of type classes had significant impact onto the development of HOLCF. Some problems Sten Agerholm had to deal with could be avoided[1] but

[1] See the discussion after the introduction of theory Cfun1 on page 12.

on the other hand new mechanisms for conservative (safe) theory extensions with respect to type classes had to be established first.

This paper is organized as follows. In section 2 I will give a brief survey of HOLC the higher order logic with type classes. The main focus of this survey is on the differences between Gordon's HOL [GM93] and HOLC which is implemented in the Isabelle system [Pau94]. Section 3 addresses the central issue of the paper, namely the development of the logic HOLCF. Finally section 4 draws a conclusion together with a survey of topics that have been formalized in HOLCF too but could not be presented in this paper due to space limitation. There is also a discussion of current and future work.

2 Higher order logic with type classes

The logic HOLC is a variant of Gordon's HOL [GM93]. It is formalized within the Isabelle system [Pau94] which is not only a logical framework but also a generic tactical theorem prover. In Isabelle terms HOLC is called an *object logic* which is formalized using Isabelle's *meta logic*, namely intuitionistic higher order logic. The meta logic is the formal language of the logical framework Isabelle [Pau89]. In Isabelle the logic HOLC is just called HOL but in this paper I use the name HOLC to avoid confusion with Gordon's HOL.

Besides some minor syntactic differences the main difference between Gordon's HOL and Isabelle's HOLC is the availability of *type classes* in HOLC. The concept of type classes is not specific to the object logic HOLC; it is derived from Isabelle's meta logic. In the beginning type classes were introduced in Isabelle by Nipkow [Nip91, NP93] as a purely syntactic device. They admit a fine grained use of polymorphism for the description of object logics. Since type classes are available in the meta logic they can be used in object logics, too. However, this is only sensible if the semantics of the object logic gives meaning to the concept of polymorphism with type classes.

2.1 What are type classes in HOLC?

This question is answered best by using some examples. As a basis for the following examples some knowledge of polymorphism in Gordon's HOL, which is Hindley/Milner polymorphism, is assumed. A detailed description of polymorphism in HOL, especially its semantics, can be found in [GM93].

In Gordon's HOL types are interpreted as inhabitants of a universe of sets which exhibits certain closure properties sufficient for the interpretation of types and type constructors. Polymorphic constants, such as the identity function $=::\alpha\Rightarrow\alpha\Rightarrow\mathsf{bool}$ or Hilbert's choice function $\varepsilon::(\alpha\Rightarrow\mathsf{bool})\Rightarrow\alpha$, are interpreted as families of interpretations (generalized cartesian products indexed by the sets of the universe). The syntax of HOL provides type variables, usually denoted by small Greek letters, e.g. in the type $\alpha\Rightarrow\alpha\Rightarrow\mathsf{bool}$ of the polymorphic equality $=$. Type terms are interpreted in an environment which maps every type variable to a member of the above mentioned universe of types.

The interpretation of types in HOLC is slightly more involved. In HOLC there may be different kinds of types or *classes* of types respectively in order to use Isabelle's terminologies. In the semantics of HOLC every type class is associated with its private universe of type interpretations. The most important type class in HOLC is the class term the semantics of which directly corresponds to the HOL universe of sets. Besides the type class term, which is mandatory, theories in HOLC may depend on additional type classes. The issue of additional type classes is discussed later on in this paper. For the moment, let us assume that there is just the class term. The semantics of a HOLC theory which respects this restriction corresponds to the semantics in Gordon's HOL.

Now I will present some examples which demonstrate the use of type classes. Suppose we want to formalize partially ordered sets (po's) so that we can address the ordering relation of the ordered sets via the polymorphic constant $\sqsubseteq :: \alpha \Rightarrow \alpha \Rightarrow \mathsf{bool}$ like in LCF. A first attempt for a formalization, in which the power of type classes is not used, would be the following (the syntax is explained below):

Porder0_first = HOL +
default term
consts
$\sqsubseteq :: \alpha \Rightarrow \alpha \Rightarrow \mathsf{bool}$
rules
refl_less $x \sqsubseteq x$
antisym_less $x \sqsubseteq y \wedge y \sqsubseteq x \longrightarrow x = y$
trans_less $x \sqsubseteq y \wedge y \sqsubseteq z \longrightarrow x \sqsubseteq z$
end

The example shows a typical theory extension in Isabelle. The new theory is called Porder0_first. It extends the theory HOL with a new polymorphic constant $\sqsubseteq$ of type $\alpha \Rightarrow \alpha \Rightarrow \mathsf{bool}$. The properties of the new constant are specified using the three well known axioms refl_less, antisym_less and trans_less. The phrase default term tells Isabelle's type inference mechanism that every type variable which occurs without an explicit qualification of the type class should be treated as a type variable of class term. In the above example we used this default mechanism to simply write $\sqsubseteq :: \alpha \Rightarrow \alpha \Rightarrow \mathsf{bool}$ instead of the more verbose phrase $\sqsubseteq :: \alpha :: \mathsf{term} \Rightarrow \alpha \Rightarrow \mathsf{bool}$.

In the three axioms of the example above it is not necessary to mention any types since the type inference mechanism can deduce all of the needed information. The technique of type inference for type class polymorphism is addressed in full detail in [NP93].

The theory Porder0_first is problematic for two reasons. First of all the theory constitutes an extension which is not *safe* in the sense of HOL.[2] We used three axioms to specify the notion of a partial order instead of using definitions which is prefered in HOL since definitions preserve models and therefore consistency. The second problem is that the above formalization is too strong. It means

[2] See page 5 for an explanation of *safe extension*.

that every type τ in the class term must be equipped with an ordering relation $\sqsubseteq :: \tau \Rightarrow \tau \Rightarrow$ bool. One could argue that there is always at least one trivial partial ordering for every type, namely the identity relation. However, this rather crude patch is ruled out immediately, once we add the additional constant $\bot :: \alpha$ and a fourth axiom

minimal $\quad \bot \sqsubseteq x$

With the help of type classes we can find an elegant way out of the second problem and with a little more effort we can make this way a safe one too, which also solves the first problem of conservativity. A discussion of solutions to this problem which stay in the framework of Gordon's HOL can be found in Agerholm's thesis [Age94]. We reformulate our first attempt in the following way:

```
Porder0_second = HOL +
default term
classes  po < term
consts
        ⊑:: (α::po)⇒α⇒bool
rules
refl_less        x ⊑ x
antisym_less     x ⊑ y ∧ y ⊑ x ⟶ x = y
trans_less       x ⊑ y ∧ y ⊑ z ⟶ x ⊑ z
end
```

The phrase classes po $<$ term declares the new class identifier po. By convention a theory which mentions a class identifier in a context like classes po $<$ term is interpreted as the definition of the class on the left hand side of the $<$ symbol.[3] The properties of the new type class are entirely specified in the body of the class definition. First of all, the phrase po $<$ term means that po is supposed to be a *subclass* of term.

In order to explain the semantics of the subclass relation $<$, we have to look at the constants and the axioms which are specified in the sections consts and rules respectively. The constants in the section consts of a class definition are called the *characteristic constants* of the new class. The axioms in the section rules are called the *characteristic axioms* of the new class. The semantics of the class po is now as follows.

It consists of a universe of mathematical structures (algebras if you like) such that each of these structures can be obtained by the following construction. First we take a structure out of the universe for the *superclass*. In the example the superclass is term and the structures in the universe for term solely consist of a carrier set. Then we add an interpretation for the characteristic constant $\sqsubseteq$, such that the characteristic axioms for this constant, here the axioms of a partial ordering, are fulfilled. In summary the interpretation of the class po is the

[3] This implies that there must be exactly one such occurrence per class identifier. The class term is the only exception to this rule.

universe of all partial orderings[4] that can be obtained by taking the carrier from the universe of the superclass term and adding an arbitrary ordering function.

In general the semantics for a class h with $\mathsf{h} < \mathsf{k}$ is the universe of all structures that can be obtained from the structures in k by adding interpretations for the characteristic constants of h such that the characteristic axioms of h are fulfilled. This way, every structure in the universe carries along the particular interpretations for the characteristic constants of that class and all its superclasses.

This is sufficient since the characteristic constants and axioms[5] of a class must be polymorphic exactly in one type variable of the class.[6] As usual the polymorphic constants are interpreted as generalized cartesian products, but this time the products are indexed by the entire structures of the universe for the type class. This way it is easy to select always the right instance for the polymorphic constant, which is by construction the one carried along as part of the indexing structure. See [Reg94] for a formal treatment.

Now we come back to the first problem, namely the conservativity of the theory extension. In HOL an extension of a theory is called conservative (save, definitional) if and only if an arbitrary model of this theory can be extended in a *strongly persistent* [EGL89, GM93] way to a model of the extended theory. Therefore safe extensions also preserve consistency.

In our example we extended the theory HOL with a new class po together with a description of its characteristic constants and axioms. The only thing which can go wrong is that the new universe for the class po is empty. This means that there is no way to extend any structure of the universe term by an interpretation for $\sqsubseteq$ that fulfills the characteristic axioms. In order to prevent this failure we simply have to show in advance that there is at least one such extension.

In the example it is easy to find a witness. For the type of the witness we take bool and as ordering function we take the identity $=:\mathsf{bool}\Rightarrow\mathsf{bool}\Rightarrow\mathsf{bool}$. Then we prove the following theorems in the theory HOL:

$$\mathsf{x}\ (=::\mathsf{bool}\Rightarrow\mathsf{bool}\Rightarrow\mathsf{bool})\ \mathsf{x}$$
$$\mathsf{x}\ (=::\mathsf{bool}\Rightarrow\mathsf{bool}\Rightarrow\mathsf{bool})\ \mathsf{y} \wedge \mathsf{y} = \mathsf{x} \longrightarrow \mathsf{x} = \mathsf{y}$$
$$\mathsf{x}\ (=::\mathsf{bool}\Rightarrow\mathsf{bool}\Rightarrow\mathsf{bool})\ \mathsf{y} \wedge \mathsf{y} = \mathsf{z} \longrightarrow \mathsf{x} = \mathsf{z}$$

Of course this is a trivial task but it shows that there will be at least one structure in the universe for the class po, namely the interpretation of the type bool together with the identity function on type bool as the ordering function.

[4] Carrier plus ordering function. In higher order logic we use functions $\tau\Rightarrow\tau\Rightarrow\mathsf{bool}$ to model binary relations.

[5] These are precisely those constants and axioms which occur in the class definition.

[6] This restriction leads to a strictly weaker notion of class polymorphism than the one known e.g. from the functional programming languages Haskell or Gofer. However, the restriction simplifies the semantics of class polymorphism since the interpretation of the instance of a characteristic constant is always non-polymorphic. It therefore can and is supposed to be an element of some appropriate carrier set in the universe of class term.

Now we are ready to give the final version of the theory Porder0 which is an example of an *extension by a new class*. It is as follows:

```
Porder0 = HOL +
default term
classes   po < term
arities   bool::po
consts
          ⊑:: (α::po)⇒α⇒bool
rules
refl_less          x ⊑ x
antisym_less       x ⊑ y ∧ y ⊑ x ⟶ x = y
trans_less         x ⊑ y ∧ y ⊑ z ⟶ x ⊑ z

inst_bool_po       (⊑::bool⇒bool⇒bool) = (=::bool⇒bool⇒bool)
end
```

The only difference between the version Porder0_second and the final one is that we mentioned the witness type bool. In the section arities we specified that the type bool is a type in class po and the axiom inst_bool_po describes the *instance* of the characteristic constant $\sqsubseteq$ for the witness type bool. The new *arity* and the instance definition are validated by the theorems we proved before which also guarantee that the above theory extension is safe. A formal treatment of all these argumentations can be found in [Reg94].

In the examples above we saw how to introduce a new type class in a conservative way. Suppose now that we want to formalize the following. Given a type σ in class term and a type τ in class po the type of functions $\sigma \Rightarrow \tau$ can be partially ordered too using the pointwise extension of the ordering in τ. This time we make the theory extension safe from the beginning. First we define the pointwise ordering on the function space:[7]

```
Fun1 = Porder0 +
consts
          less_fun:: (α::term ⇒ β::po) ⇒ (α ⇒ β) ⇒ bool
rules
less_fun_def       less_fun = (λf1 f2.∀x. f1(x) ⊑ f2(x))
end
```

In the theory Fun1 we just introduced the new constant less_fun. Since the only axiom less_fun_def is a definition the theory extension Fun1 is obviously conservative. The theory Fun1 is an example for an *extension by a new constant* which corresponds to the same notion in Gordon's HOL. Constants which are introduced in this way are different from characteristic constants of classes. Therefore they are not restricted in the degree of their polymorphism. Characteristic constants can only be introduced within a class definition. However, for

[7] In the example Fun1 is based on Porder0. In the full development of HOLCF the theory Fun1 is based on additional theories Porder and Pcpo. See figure 1.

the definition of a new constant characteristic constants of an already defined class can be used like in the example above. Next we prove the following three theorems in the theory Fun1:

$$\begin{array}{l} \mathsf{less\_fun(x)(x)} \\ \mathsf{less\_fun(x)(y)} \wedge \mathsf{less\_fun(y)(x)} \longrightarrow \mathsf{x} = \mathsf{y} \\ \mathsf{less\_fun(x)(y)} \wedge \mathsf{less\_fun(y)(z)} \longrightarrow \mathsf{less\_fun(x)(z)} \end{array}$$

These theorems show that the function less_fun behaves like a partial ordering. Therefore we are allowed to formalize the following theory extension which is called an *extension by a new arity*.

```
Fun2 = Fun1 +
arities   ⇒ :: (term,po)po
rules
inst_fun_po       (⊑::(α::term ⇒ β::po) ⇒ (α ⇒ β) ⇒ bool) = less_fun
end
```

The phrase ⇒::(term,po)term tells Isabelle's type inference mechanism that given a type σ in class term and a type τ in class po the function space $\sigma \Rightarrow \tau$ is a type in class po. The axiom inst_fun_po fixes the instance of the characteristic constant $\sqsubseteq$ for the type $\sigma \Rightarrow \tau$. Due to the theorems we proved in advance the above extension is safe again.

This concludes the short survey on the logic HOLC. Besides the *extension by a new class* (theory Porder0), the *extension by a new arity* (theory Fun2) and the *extension by a new constant* (theory Fun1) there is the *extension by a new type* of class term. This last extension mechanism corresponds directly to the extension by type definition in Gordon's HOL and is therefore not discussed here. However, we will see an example for this extension mechanism in section 3.4.

3 Development of HOLCF

In this section I will present parts of the development of HOLCF using the higher order logic HOLC with type classes which was briefly described in the previous section. Figure 1 shows part of the hierarchy of theories which constitutes the logic HOLCF.

The theory Porder0 is known from section 2. In this theory the type class po of partial orders is introduced. In Porder the notions of upper bounds, least upper bounds and ω-chains are introduced. In theory Pcpo the class pcpo of pointed complete partial orders is introduced as a subclass of po. The characteristic constant of this class is the symbol $\bot$ for the least element. The main parts of the theories Fun1 and Fun2 were already presented in section 2. Theory Fun3 contains just the arity and instance declarations for the function type constructor $\Rightarrow$ with respect to the type class pcpo.

In theory Cont the notions of monotone and continuous functions are defined as predicates on the full function space $\alpha \Rightarrow \beta$ over pcpo's α and β. Since

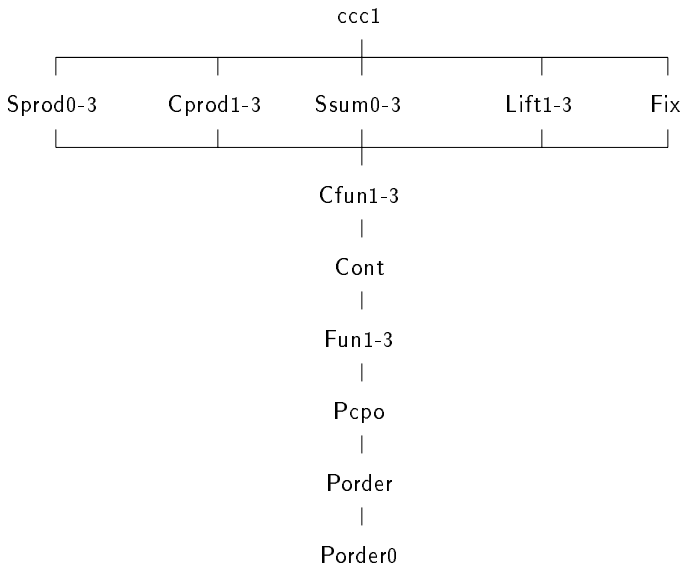

Fig. 1. The HOLCF theories

continuous functions play a central role in the logic LCF the theories Cfun1−3 introduce a special type constructor $\rightarrow$ for continuous functions. In order to avoid confusion elements of the type $\alpha \rightarrow \beta$ are called *operations*. The type $\alpha \rightarrow \beta$ is introduced via a type definition that fixes the interpretation of $\alpha \rightarrow \beta$ to be isomorphic to the subset of continuous functions of the type $\alpha \Rightarrow \beta$. We will discuss this further below.

The theories Sprod0−3, Cprod1−3, Ssum0−3 and Lift1−3 conservatively introduce the types of strict products, cartesian products, strict sums and lifting type over types in class pcpo. Since the extensions have to be safe there are always several steps needed for every type construction. In a first step the type construction itself is defined via an extension by a new type of the class term. In the next step, the ordering function is defined and it is proved that the function behaves like a partial ordering. The next step shows that there is a least element in the type under consideration and that there always exist least upper bounds for ω-chains. This validates the last step which specifies that the type construction yields types in class pcpo provided the argument types of the type construction are in class pcpo too.

The theory Fix contains the fixed point theory of LCF. Central to this theory are the definitions of the fixed point operator and the definition of admissibility. Amongst others Kleene's fixed point theorem, Scott induction and various propagations of admissibility are proved.

The theory ccc1 is a union of the theories enumerated above and defines in

addition the identity operation and the composition of operations. The name of the theory stems from the fact that the class pcpo together with operations as arrows forms a category. In addition this category can also be shown to be cartesian closed by taking the cartesian product as categorical product and the type of operations as exponential.[8] In the following we concentrate on the theories Porder, Pcpo, Cont, Cfun1–3 and Fix.

3.1 The theory Porder

In theory Porder the notions of upper bounds, least upper bounds (lub's) and ω-chains are introduced. Due to the use of the type class po the polymorphism of the various constants can be restricted to the class of partially ordered types. This leads to a very natural formalization of the concepts above. Note how the characteristic constant $\sqsubseteq$ of the class po is used in the axioms. The default class is still term. Therefore the type variable α is explicitly qualified with the class po. However, one qualification per type term is sufficient. Note the difference between the infix predicate $\ll|$ and the function lub. The former is a *relation* which means that x is a least upper bound of set S whereas the latter is a *function* which yields *some* x that is a least upper bound of S provided there exists one. The type constructor set is the polymorphic powerset constructor. Applied to a type τ it constructs the powerset of τ that is isomorphic to the type $\tau \Rightarrow$bool.

Porder = Porder0 +
consts

$<\|$	:: α set $\Rightarrow$ α::po $\Rightarrow$ bool	(infixl 55)
$\ll\|$	:: α set $\Rightarrow$ α::po $\Rightarrow$ bool	(infixl 55)
lub	:: α set $\Rightarrow$ α::po	
is_chain	:: (nat$\Rightarrow\alpha$::po) $\Rightarrow$ bool	

rules

is_ub	S $<\|$ x = $\forall$y.y$\in$S $\longrightarrow$ y $\sqsubseteq$ x
is_lub	S $\ll\|$ x = S $<\|$ x $\wedge$ ($\forall$u. S $<\|$ u $\longrightarrow$ x $\sqsubseteq$ u)
lub	lub(S) = (εx. S $\ll\|$ x)
is_chain	is_chain(Y) = ($\forall$i.Y(i) $\sqsubseteq$ Y(Suc(i)))

end

It is convenient to have both of these notions to talk about least upper bounds. For technical reasons an ω-chain is formalized as a function which enumerates the chain and not as the range of this enumeration.

3.2 The theory Pcpo

In this theory we introduce the new type class pcpo (*pointed complete partial orders*) as a subclass of po. The intention is that pcpo is inhabited by all types

[8] Together with suitable arrows.

which are not only partially ordered but in addition have a least element and are complete with respect to ω-chains. This is the kind of types which is needed to formalize the logic LCF.

```
Pcpo = Porder +
classes  pcpo < po
arities  void :: pcpo
consts
         ⊥:: α::pcpo
rules
minimal          ⊥ ⊑ x
cpo              is_chain(Y) ⟶ ∃x. range(Y) ≪| x::(α::pcpo)

inst_void_pcpo   (⊥::void) = ⊥_void
end
```

The witness for the non-emptiness of the new class is the trivial type void which solely consists of one element ⊥_void. Clearly this type is partially ordered by the identity relation and has ⊥_void as least element. I did not provide the formalization of the type and its instance for the class po since it is trivial.

Note that due to the explicit qualification x::(α::pcpo) chain completeness is only required for types in class pcpo. Without this qualification the type inference mechanism would have computed α::po which would be too strong. The function range is the function which yields the range of its argument function. Here we get the range of the chain Y.

3.3 The theory Cont

We skip the theories Fun1, Fun2 and Fun3. In these theories it is shown that the full function space α⇒β over types α::term and β::pcpo can be partially ordered by the pointwise ordering and has pcpo structure. We immediately skip to the theory Cont that introduces the notions of monotone and continuous functions.

```
Cont = Fun3 +
default pcpo
consts
         monofun :: (α::po ⇒ β::po) ⇒ bool
         contlub  :: (α ⇒ β) ⇒ bool
         contX    :: (α ⇒ β) ⇒ bool
rules
monofun          monofun(f) = ∀x y. x ⊑ y ⟶ f(x) ⊑ f(y)

contlub          contlub(f) = ∀Y. is_chain(Y) ⟶
                                  f(lub(range(Y))) = lub(range(λi.f(Y(i))))

contX            contX(f)   = ∀Y. is_chain(Y) ⟶
                                  range(λi.f(Y(i))) ≪| f(lub(range(Y)))
end
```

First of all note that we changed the default class to be pcpo. Therefore we need the explicit qualification po for the type of the predicate monotone. The first two axioms of the theory directly correspond to those which can be found in every text book.[9] This is due to the use of type classes which allows us to hide a lot of details behind the scenes. In a higher order logic without type classes the axioms would be cluttered with premises about the ordering relation which needs to be passed as an explicit argument to all of the predicates above. See [Age94] for more details.

Perhaps the definition of the third axiom contX is surprising. However, it can be proved, and indeed it was proved in this theory that the following holds.

$$\mathsf{contX(f)} = (\mathsf{monofun(f)} \wedge \mathsf{contlub(f)})$$

3.4 Theories Cfun1 - Cfun3

The theory Cfun1 is central to the development of HOLCF. Here the expressive power of higher order logic with type classes is apparent in several places. In theory Cfun1 we introduce the type of operations such that its semantics is isomorphic to the subset of continuous functions. The theory is as follows:

```
Cfun1 = Cont +
types    →  2    (infixr 5)
arities  →  ::   (pcpo,pcpo)term
consts
  Cfun   :: (α ⇒ β)set

  fapp   :: (α → β)⇒(α ⇒ β)          ( (_[_])  [1000,0] 1000)
  fabs   :: (α ⇒ β)⇒(α → β)          (binder Λ  10)

  less_cfun :: (α → β)⇒(α → β)⇒bool
rules
  Cfun_def              Cfun = {f. contX(f)}

  Rep_Cfun              fapp(g) ∈ Cfun
  Rep_Cfun_inverse      fabs(fapp(g)) = g
  Abs_Cfun_inverse      f ∈ Cfun ⟶ fapp(fabs(f))=f

  less_cfun_def         less_cfun(g1,g2) = ( fapp(g1) ⊑ fapp(g2) )
end
```

The theory Cfun1 is an example for a *conservative extension by a new type*. The constructor → is introduced as an infix type constructor. The three axioms Rep_Cfun, Rep_Cfun_inverse and Abs_Cfun_inverse state that the type $\alpha \rightarrow \beta$ is

[9] In a text book you probably will find f(⨆i.Y(i)) = ⨆i.f(Y(i)) instead of f(lub(range(Y))) = lub(range(λi.f(Y(i)))).

isomorphic to the set Cfun which is the set of all continuous functions of type $\alpha \Rightarrow \beta$. Of course it has been shown in advance that this subset is not empty.[10]

The interesting thing about this theory is that the new constructor is restricted to argument types which are in class pcpo. This restriction is vital since without a pcpo structure the 'subset of all continuous functions' is without any meaning. Due to the use of type classes the new constructor is 'total' on its argument classes. The same situation arises during the formalization of strict products and strict sums (theories Sprod0−3, Ssum0−3).

The technique used is similar to the one which can be found in languages with subtypes. There suitable subtypes are used to model partial functions. In a higher order logic without type classes there is no way to introduce a type constructor for continuous functions, strict products and strict sums since it would have to be partial. See [Age94] for a detailed discussion of the problem.

The mysterious phrases ((_[_]) [1000,0] 1000) and (binder Λ 10) introduce mixfix syntax for the new type. Instead of writing the less readable fapp(f)(x) for application and fabs(λx.t(x)) for abstraction of an operation, the user simply writes f[x] and Λx.t(x). This syntactic sugaring yields a smooth embedding of LCF terms. A term is part of this LCF sublanguage if it is just built of variables, continuous constants, Λ-abstractions and _[_]-applications.

As a result of the above type definition β-reduction for operations is subject to a restriction which concerns the continuity of the abstraction. It can be shown that the following theorem about β-reduction of operations holds:

$$\mathsf{contX(t)} \longrightarrow (\Lambda \mathsf{x.t(x)})[\mathsf{u}] = \mathsf{t(u)}$$

This means that in order to do a β-reduction the continuity of the body has to be proved first. Fortunately this continuity proof can be done automatically if the body t(x) is a term in the LCF sublanguage.

The last axiom less_cfun_def defines the ordering relation for operations. Of course the ordering is inherited from the full function space. In the theories Cfun2 and Cfun3 it is shown that the ordering defined above really yields a pcpo-structure which finally is used to validate the instances

inst_cfun_po $(\sqsubseteq :: (\alpha \rightarrow \beta) \Rightarrow (\alpha \rightarrow \beta) \Rightarrow \mathsf{bool}) = \mathsf{less\_cfun}$
inst_cfun_pcpo $\bot :: \alpha \rightarrow \beta = \Lambda \mathsf{x}.\bot$

and the arity definitions

arities $\rightarrow$ :: (pcpo,pcpo)po
arities $\rightarrow$ :: (pcpo,pcpo)pcpo

[10] The new Isabelle version provides a subtype package in the style of Gordon's HOL that produces this axiomatization behind the scenes. The package also checks whether the user supplied a theorem about the non-emptiness of the representing set.

3.5 The theory Fix

This theory introduces the fixed point theory of LCF. The main parts are shown below. The iterator iterate which iterates an operation n-times starting with value c is defined by primitive recursion. The parameter n in the third argument of primitive recursion is not really needed for the definition of the iterator. However, we have to supply it in order to confirm to the type of primitive recursion nat_rec.

```
Fix = Cfun3 +
consts
  iterate      :: nat⇒(α→α)⇒α⇒α
  lfix         :: (α→α)⇒α
  fix          :: (α→α)→α
  adm          :: (α⇒bool)⇒bool
rules
  iterate_def  iterate(n,F,c) = nat_rec(n,c,λn x.F[x])

  lfix_def     lfix(F) = lub(range(λi.iterate(i,F,⊥)))
  fix_def      fix = (Λf. lfix(f))

  adm_def      adm(P) = ∀Y. is_chain(Y) ⟶
                              (∀i.P(Y(i))) ⟶ P(lub(range(Y)))
end
```

The *function* lfix of type $(\alpha\to\alpha)\Rightarrow\alpha$ is just introduced as intermediate constant to ease the technical treatment. The interesting constant is the fixed point operator fix which has type $(\alpha\to\alpha)\to\alpha$ of an operation. In definition adm_def the notion of admissibility is defined. Some of the main theorems of the theory Fix are the following fixed point properties:

```
fix_eq       fix[F]=F[fix[F]]
fix_least    F[x]=x ⟶ fix[F] ⊑ x
fix_def2     fix[F] = lub(range(λi. iterate(i,F,⊥)))
```

The first two of them are well known from LCF. Note that the notation corresponds to the one used in LCF. Clearly the third theorem is beyond the expressive power of LCF. It is Kleene's constructive characterization of the least fixed point of a continuous function. It is already a theorem since only *functions* may be defined using an application context. Extensionality of functions guarantees the conservativity of such 'definitions'. In order to derive the theorems above, the continuity of the *function* lfix had to be proved first. The proof follows the argumentation that can be found in the literature about LCF. See [Win93] and [Gun92] for two different approaches.

Two other prominent theorems of domain theory are the principle of Scott-Induction and computational induction.

```
fix_ind      adm(P) ∧ P(⊥) ∧ (∀x. P(x) ⟶ P(F[x])) ⟶ P(fix[F])
comp_ind     adm(P) ∧ (∀n. P(iterate(n,F,⊥))) ⟶ P(fix[F])
```

They both immediately follow from the definition of admissibility. In addition various propagations of admissibility were derived. Some of these are listed below:

adm_⊥ess	$\mathsf{contX}(u) \wedge \mathsf{contX}(v) \longrightarrow \mathsf{adm}(\lambda x.u(x) \sqsubseteq v(x))$
adm_subst	$\mathsf{contX}(t) \wedge \mathsf{adm}(P) \longrightarrow \mathsf{adm}(\lambda x.P(t(x)))$
adm_conj	$\mathsf{adm}(P) \wedge \mathsf{adm}(Q) \longrightarrow \mathsf{adm}(\lambda x.P(x) \wedge Q(x))$
adm_disj	$\mathsf{adm}(P) \wedge \mathsf{adm}(Q) \longrightarrow \mathsf{adm}(\lambda x.P(x) \vee Q(x))$

In LCF these theorems are hard-wired as syntactic tests in the system since they cannot even be expressed inside the logic. See [Pau84] for a discussion of the drawbacks of this lack of expressive power. HOLCF is much more flexible since the admissibility of a predicate can often be derived by a special argumentation although the predicate does not directly fit into the syntactic schemes like the ones listed above. This is due to the fact that admissibility is definable in HOLCF.

4 Conclusion

In section 2 the central ideas of higher order logic with type classes were presented. In particular mechanisms for theory extensions with respect to type classes and their conservativity were illustrated using some simple examples. In section 3 the main steps of the development of HOLCF, a higher order version of LCF, were described.

Only a few theories were presented and almost no theorems. However, summing up it took more than 30 steps of conservative theory extensions and about 600 theorems to formalize and derive all the logical concepts that constitute LCF. The full formalization of HOLC, its syntax, semantics and proof rules together with a detailed description of the development of HOLCF and some applications can be found in my thesis [Reg94].

Due to the use of type classes and Isabelle's advanced syntactic capabilities the resulting formalization of LCF is smoothly integrated into HOLC. Higher order logic and logic of computable functions can freely be mixed which yields a higher order version of LCF, namely HOLCF. The advantage of this combination was briefly discussed during the presentation of fixed point theory. The concept of admissibility can be formalized inside the logic which remedies some drawbacks of LCF.

There are other advantages of the combination that could not be discussed due to a lack of space. In [Reg94] some recursive data types like strict lists or streams were formalized in HOLCF. For types with strict constructors (e.g. strict lists) structural induction principles can be derived that are not restricted by any admissibility considerations. In LCF only for strict types over chain-finite argument types can the admissibility proviso be eliminated [Pau87]. In addition, for all tree-like types a co-induction principle [Pit92] can be derived in HOLCF.[11]

[11] Usually this is only interesting for types with infinite elements, e.g. streams.

Currently HOLCF is tuned for use as the kernel language of a specification language for distributed systems in the style of [BDD+93]. A type definition package in the style of LCF that produces exclusively conservative axiomatizations is in preparation.

5 Acknowledgment

I am grateful for the constructive suggestions received from the referees. I would also like to thank Tobias Nipkow for his advice and many discussions about HOLCF.

References

[Age94] Sten Agerholm. *A HOL Basis for Reasoning about Functional Programs.* PhD thesis, University of Aarhus, BRICS Departement of Computer Science, 1994. BRICS Report Series RS-94-44.

[BDD+93] Manfred Broy, Frank Dederichs, Claus Dendorfer, Max Fuchs, Thomas Gritzner, and Rainer Weber. The Design of Distributed Systems: An Introduction to FOCUS. Technical Report TUM-I9202-2, Institut für Informatik, Technische Universität München, 1993.

[EGL89] H.D. Ehrich, M. Gogolla, and U.W. Lippeck. *Algebraische Spezifikation abstrakter Datentypen.* Teubner, 1989.

[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. Gordon, R. Milner, and C. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation*, volume 78 of *LNCS*. Springer, 1979.

[Gun92] Carl A. Gunter. *Semantics of Programming Languages: Structures and Techniques.* The MIT Press, 1992.

[Nip91] Tobias Nipkow. Order-Sorted Polymorphism in Isabelle. In G. Huet, G. Plotkin, and C. Jones, editors, *Proc. 2nd Workshop on Logical Frameworks*, pages 307–321, 1991.

[NP93] Tobias Nipkow and Christian Prehofer. Type checking type classes. In *Proc. 20th ACM Symp. Principles of Programming Languages*, pages 409–418, 1993.

[Pau84] L.C. Paulson. *Deriving Structural Induction in LCF*, volume 173 of *LNCS*, pages 197–214. Springer, 1984.

[Pau87] L.C. Paulson. *Logic and Computation, Interactive Proof with Cambridge LCF*, volume 2 of *Cambridge Tracts in Theoretical Computer Science.* Cambridge University Press, 1987.

[Pau89] L.C. Paulson. The foundation of a generic theorem prover. *Journal of Automated Reasoning*, 5(3):363–397, 1989.

[Pau94] L.C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *LNCS.* Springer, 1994.

[Pit92] Andrew Pitts. A co-induction principle for recursively defined domains. Technical Report 252, University of Cambridge, Computer Laboratory, 1992.

[Reg94] Franz Regensburger. HOLCF: Eine konservative Erweiterung von HOL um LCF, 1994. Dissertation, Technische Universität München.

[Win93] G. Winskel. *The Formal Semantics of Programming Languages.* The MIT Press, 1993.

This article was processed using the LaTeX macro package with LLNCS style