Axiomatic Constructor Classes in Isabelle/HOLCF

Brian Huffman, John Matthews, and Peter White

OGI School of Science and Engineering at OHSU, Beaverton, OR 97006
{brianh,johnm,peterw}@cse.ogi.edu

Abstract. We have definitionally extended Isabelle/HOLCF to support axiomatic Haskell-style constructor classes. We have subsequently defined the functor and monad classes, together with their laws, and implemented state and resumption monad transformers as generic constructor class instances. This is a step towards our goal of giving modular denotational semantics for concurrent lazy functional programming languages, such as GHC Haskell.

1 Introduction

The Isabelle generic theorem prover is organized as a modular collection of tools for reasoning about a variety of logics. This allows common theorem proving tasks such as parsing, pretty printing, simplification of formulas, and proof search tactics to be reused in each object logic. We are similarly interested in using Isabelle to modularly reason about programs written in a wide spectrum of programming languages. In particular, we want to verify programs written in lazy functional programming languages, such as Haskell [11].

Denotational semantics is one attractive approach for this, owing to its high level of abstraction and the ease in which both recursive datatypes and functions can be modeled. However, language features change over time. Modeling new language datatypes and primitives usually only requires local changes to the language semantics. On the other hand, the introduction of a new computational effect, such as exceptions, often requires global modifications. Furthermore, these effects must be formalized anew for each programming language under consideration.

In the last decade *monads* have become an increasingly popular way to mitigate this problem. A monad M is a single-argument type constructor together with a particular set of operations and equational laws (listed in Section 4.4) that can be used for expressing kinds of computational effects, such as imperative state update, exception handling, and concurrency. For more information, we recommend consulting one of the Haskell-oriented monad tutorials at [`http://www.haskell.org/bookshelf/#monads`].

When formalizing the semantics of a language, a monad can be defined that models the language's computational effects. The rest of the semantics is then specified abstractly in terms of the monad, allowing the language's effects to be modified in isolation.

Even greater modularity can be achieved by composing complex monads through a series of *monad transformers*. A monad transformer takes an existing monad and extends it with a specific new computational effect, such as imperative state, or exception handling. In this way an effect can be specified once, as a monad tranformer, and then reused in other language semantics.

1.1 Axiomatic Constructor Classes

Isabelle supports overloaded constant definitions [21]. Polymorphic constants usually have a single definition that covers all type instances, but multiple definitions are allowed if they apply to separate types. For example, unary negation in Isabelle/HOL has the polymorphic type $'a \Rightarrow 'a$. It may be applied to sets or to integers, with a different meaning in each case.

Axiomatic type classes [21] are another important feature of Isabelle. Each class has a set of *class axioms*, each of which has a single free type variable, and specifies properties of overloaded constants. An axiomatic type class is then a set of types: those types for which the class axioms have been proven to hold. Theorems can express assumptions about types using class constraints on type variables. Axiomatic type classes are used extensively in Isabelle's implementation of domain theory (see Section 2.1).

Unfortunately, Isabelle's type class system is not powerful enough to specify classes for monads or monad transformers. The problem is that Isabelle supports abstraction over types, using type variables; but there is no such abstraction for type constructors—they can only be used explicitly. This means that we can prove the monad laws hold for a particular type constructor, but we can not reason abstractly about monads in general. Furthermore, monad transformers cannot even be defined, since Isabelle does not allow type constructors to take other type constructors as arguments.

A key observation is that we can represent types themselves as values, and we can represent continuous type constructors as continuous functions over those values. We can then use Isabelle's existing type definition and axiomatic type class packages to represent such type constructors as new types. This now allows us to use type variables to reason abstractly about type constructors, and thus we can specify constructor classes simply as type classes.

This representation is carried out definitionally, and we have gone on to encode several monads and monad transformers in Isabelle/HOLCF. The most interesting of these is the resumption monad transformer, which models interleaving of computations.

2 Background

Isabelle is a generic interactive theorem prover, which can be instantiated with various kinds of object-logics. Isabelle/HOL is an instantiation of higher order logic. We will now summarize the basic syntax and keywords of Isabelle/HOL that will be used in the paper.

The formula syntax in Isabelle/HOL includes standard logical notation for connectives and quantifiers. In addition, Isabelle has separate syntax for the meta-level logic: $\bigwedge$, $\Longrightarrow$, and $\equiv$ represent meta-level universal quantification, implication, and equality. There is also notation for nested meta-level implication: $[\![P_1; \ldots; P_n]\!] \Longrightarrow R$ is short for $P_1 \Longrightarrow \cdots \Longrightarrow P_n \Longrightarrow R$.

The syntax of types is similar to the language ML, except that Isabelle uses a double arrow ($\Rightarrow$) for function types. Some binary type constructors are written infix, as in the product type *nat* $\times$ *bool*; other type constructors are written postfix, as in *bool list* or *nat set*. Finally, $'a$ and $'b$ denote free type variables.

Isabelle theories declare new constants with the **consts** keyword. Definitions may be supplied later using **defs**; alternatively, constants may be declared and defined at once using **constdefs**. Theories introduce new types with the **typedef** command, which defines a type isomorphic to a given non-empty set. The keywords **lemma** and **theorem** introduce theorems.

2.1 Isabelle/HOLCF

HOLCF [14,20] is an object logic for Isabelle designed for reasoning about functional programs. It is implemented as a layer on top of Isabelle/HOL, so it includes all the theories and syntax of the HOL object logic. In addition, HOLCF defines a family of new axiomatic type classes, several new type constructors, and associated syntax, which we will summarize here.

HOLCF introduces an overloaded binary relation $\sqsubseteq$, which is used to define information orderings for types: The proposition $x \sqsubseteq y$ means that x is an approximation to y. HOLCF then defines a sequence of axiomatic type classes *po* $\supseteq$ *cpo* $\supseteq$ *pcpo* to assert properties of the $\sqsubseteq$ relation. The class *po* contains types where $\sqsubseteq$ defines a partial order. The subclass *cpo* is for ω-complete partial orders, which means that there exists a least upper bound for each ω-chain. An ω-chain Y is a countable sequence where $Y_n \sqsubseteq Y_{n+1}$ for all n. The expression $\bigsqcup_n Y_n$ denotes the least upper bound of the chain Y. Finally, the class *pcpo* is for ω-cpos that additionally have a least element, written $\bot$. HOLCF declares *pcpo* to be the default class, so free type variables are assumed to be in class *pcpo* unless otherwise specified.

HOLCF defines a standard set of type constructors from domain theory. Given types $'a$ and $'b$ in class *pcpo*, and $'c$ in class *cpo*, the following are all in class *pcpo*: the cartesian product $'a \times 'b$, the strict product $'a \otimes 'b$, the strict sum $'a \oplus 'b$, the lifted type $'a\ u$, and the continuous function space $'c \to 'a$. Recall that a continuous function is a monotone function that preserves limits: $f(\bigsqcup_n Y_n) = \bigsqcup_n f(Y_n)$. HOLCF also defines a type constructor *lift* that can turn any type into a flat pcpo by adding a new bottom element.

HOLCF defines special syntax for operations involving the continuous function space. Continuous function application is written with an infix dot, as in $f{\cdot}x$. Continuous lambda abstraction is written $\Lambda x.\ P$. Composition of continuous functions f and g is written $f\ oo\ g$.

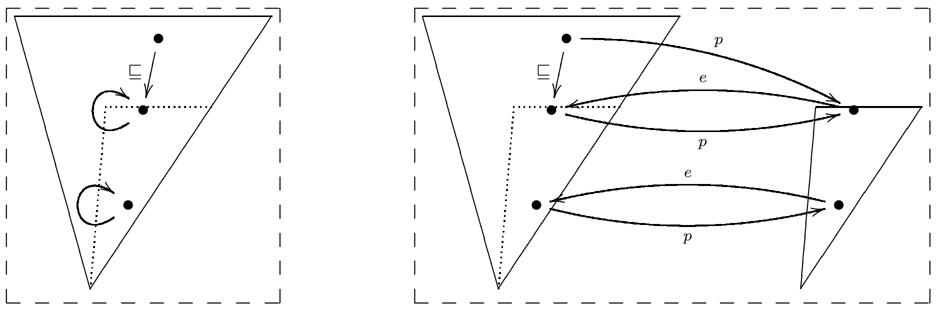

Fig. 1. Projections (*left*) and ep-pairs (*right*). The range of a projection defines a subset of a pcpo. An ep-pair defines an isomorphism between a subset and another type

3 Representing Types and Type Constructors

3.1 Embedding-Projection Pairs

To do formal reasoning about types, we need to be able to talk about what it means to embed one type into another. An appropriate concept in domain theory is the embedding-projection pair, or ep-pair [1,2,7]. Let $'a$ and $'b$ be types in class *pcpo*. A pair of continuous functions $e{::}'a \rightarrow 'b$ and $p{::}'b \rightarrow 'a$ is an ep-pair if $p\ oo\ e = ID{::}'a \rightarrow 'a$ and $e\ oo\ p \sqsubseteq ID{::}'b \rightarrow 'b$. The existence of such a pair shows that the type $'a$ can be embedded into type $'b$. An illustration of a simple ep-pair is shown in Fig. 1.

constdefs
  $\mathit{is\text{-}ep\text{-}pair} :: ('a \rightarrow 'b) \Rightarrow ('b \rightarrow 'a) \Rightarrow \mathit{bool}$
  $\mathit{is\text{-}ep\text{-}pair}\ e\ p \equiv (\forall x{::}'a.\ p{\cdot}(e{\cdot}x) = x) \wedge (\forall y{::}'b.\ e{\cdot}(p{\cdot}y) \sqsubseteq y)$

Ep-pairs have many useful properties: e is injective, p is surjective, both are strict, each function uniquely determines the other, and the range of e is a sub-pcpo of $'b$. Ep-pairs are also compositional, and they can be lifted over many type constructors, including cartesian product and continuous function space.

If we identify $'a$ with a subset of $'b$, so that e is just subset inclusion, then it is natural to consider just the composition $e\ oo\ p{::}'b \rightarrow 'b$. This gives a continuous function that is below the identity, and also idempotent, since $e\ oo\ p\ oo\ e\ oo\ p = e\ oo\ ID\ oo\ p = e\ oo\ p$. In domain theory, a function with these properties is called a projection (not to be confused with the second half of an ep-pair). Our definitions of ep-pairs and projections follow Amadio and Curien's [2, Defn. 7.1.6].

constdefs
  $\mathit{is\text{-}projection} :: ('a \rightarrow 'a) \Rightarrow \mathit{bool}$
  $\mathit{is\text{-}projection}\ p \equiv (\forall x.\ p{\cdot}(p{\cdot}x) = p{\cdot}x) \wedge (\forall x.\ p{\cdot}x \sqsubseteq x)$

A projection is a function, but it can also be viewed as a set: Just take the range of the function, or equivalently, its set of fixed points—for idempotent functions they are the same. A projection along with the set it defines are shown in Fig. 1. Every projection gives a set that is a sub-pcpo, and contains $\bot$. Not all sub-pcpos have a corresponding projection, but if one exists then it is unique. The set-oriented and function-oriented views of projections even give the same ordering: For any projections p and q, $p \sqsubseteq q$ if and only if *range* $p \subseteq$ *range* q.

We define a type constructor in Isabelle for the space of projections over any pcpo. Since *is-projection* is an admissible predicate, the set of projections is closed with respect to limits of ω-chains. Since $\Lambda x.\ \bot$ is a projection, the set also contains a least element—thus the resulting type is a pcpo.

typedef $'a$ *projection* $= \{p{::}'a \to 'a.\ \textit{is-projection}\ p\}$

Isabelle's type definition package provides *Rep* and *Abs* functions to convert between type $'a$ *projection* and type $'a \to 'a$. *Rep-projection* is injective and its range equals the set of all projection functions; *Abs-projection* is a left-inverse to *Rep-projection*. (See the Isabelle Tutorial [16, §8.5.2] for more details.)

We can define some operations on projections that are useful for reasoning about projections as sets. The *in-projection* relation is like the $\in$ relation for sets; the triple-colon notation is meant to be reminiscent of type annotations in Isabelle or Haskell. The function *cast* is implemented by simply applying a projection as a function—in the set-oriented view of projections, the intuition is that it casts values into a set, preserving those values that are already in the set.

constdefs
  $\textit{cast} :: 'a\ \textit{projection} \to 'a \to 'a$
  $\textit{cast} \equiv \Lambda D.\ \textit{Rep-projection}\ D$

  $\textit{in-projection} :: 'a \Rightarrow 'a\ \textit{projection} \Rightarrow \textit{bool}$ (**infixl** ::: *50*)
  $x ::: D \equiv \textit{cast}{\cdot}D{\cdot}x = x$

lemma *cast-in-projection*: $\textit{cast}{\cdot}D{\cdot}x ::: D$
lemma *subprojectionD*: $[\![D \sqsubseteq E;\ x ::: D]\!] \Longrightarrow x ::: E$

3.2 Representable Types

Using the domain package of Isabelle/HOLCF, we can define a universal domain U that is isomorphic to the lifted naturals plus its own continuous function space:

domain $U = \textit{UNat}\ (\textit{fromUNat} :: \textit{nat lift}) \mid \textit{UFun}\ (\textit{fromUFun} :: U \to U)$

The domain package defines continuous constructor and accessor functions for the type U, and proves a collection of theorems about them. Then we can easily show that *UNat* and *fromUNat* form an ep-pair from *nat lift* to U; similarly, *UFun* and *fromUFun* form an ep-pair from $U \to U$ to U. Using these ep-pairs as a starting point, we can then construct ep-pairs to U for several other types: *unit lift*, *bool lift*, $U \times U$, $U \otimes U$, $U \oplus U$, and $U_\bot$.

We say that a type $'a$ is *representable* if we can define an ep-pair between $'a$ and the universal domain U. We declare overloaded constants *emb* and *proj* to convert values to and from the universal domain, and encode the notion of representability with a new axiomatic type class *rep*:

consts
  $emb :: {'a} \rightarrow U$
  $proj :: U \rightarrow {'a}$

axclass $rep \subseteq pcpo$
  ep-emb-proj: *is-ep-pair emb proj*

Given two representable types $'a$ and $'b$, we can lift the ep-pairs over the cartesian product to construct a new ep-pair from $'a \times {'b}$ to $U \times U$. Then we can compose this with the standard ep-pair from $U \times U$ to U, and get an ep-pair from $'a \times {'b}$ to U. Thus $'a \times {'b}$ is representable if both $'a$ and $'b$ are, and we say that $\times$ is a *representable type constructor*. Similarly, we can show that $\rightarrow$, $\otimes$, $\oplus$, and $(-)_\perp$ are all representable type constructors as well. Our proofs of representability are analogous to those given by Gunter and Scott [7, §7.1, §7.3]. Note that U is also trivially representable, since we can embed any type into itself.

Mapping from Types to Values. We encode *values* of representable types as *values* of type U, but we encode *types* themselves as *projections* over U. We can construct a projection for any representable type by simply composing *emb* and *proj*.

constdefs
  rep-of :: $({'a}{::}rep)$ *itself* $\Rightarrow$ U *projection*
  rep-of $(t{::}{'a}$ *itself*$)$ $\equiv$ *Abs-projection* $((emb{::}{'a} \rightarrow U)$ *oo* $(proj{::}U \rightarrow {'a}))$

The special type $'a$ *itself* has only one value, which is written with the special syntax $TYPE('a)$. As the argument type of *rep-of*, it allows us to effectively take types as arguments. We could have used simply $'a$ as the argument type, but this does not accurately reflect what the function does—the result value does not depend on any actual values of type $'a$, it only depends on the type itself. This Isabelle-specific feature is not actually necessary: We could also have defined a type constructor that produces singleton types, using the datatype package; instead of $TYPE('a)$ we would just write *Myself*$::{'a}$ *itself*.

datatype $'a$ *itself* = *Myself* — *sample singleton type constructor*

The type U *projection* is large enough to encode all the programming language datatypes that we are interested in. We have already shown that the unit type, sums, products, and continuous function spaces are representable, and since U *projection* is a pcpo, it contains least fixed-points for all general recursive datatypes as well.

From now on, all free type variables are assumed to be in class *rep*, unless specified otherwise. We will not concern ourselves with non-representable types.

3.3 Representing Type Constructors

One way to think of a type constructor is as a function from types to types. Just as we can represent a type with a projection, we can represent a type constructor using a projection constructor, i.e. a continuous function of type *U projection* $\to$ *U projection*.

We declare an Isabelle type class *tycon* for type constructors. It is a syntactic class which has no axioms; the class only serves to restrict the possible argument types for overloaded functions. While instances of class *tycon* are actually types, we will never use them as such, or construct any values having those types. We only use them to define an overloaded constant *tc*, which returns a projection constructor. As before, we use *'f itself* as the argument type because the result should only depend on the type of the argument, and not its value.

axclass $\mathit{tycon} \subseteq \mathit{type}$
consts $\mathit{tc} :: ('f{::}\mathit{tycon})\ \mathit{itself} \Rightarrow U\ \mathit{projection} \to U\ \mathit{projection}$

Now we can define an Isabelle type constructor to model explicit type application. *App* takes two type arguments: *'a* of class *rep*, and *'f* of class *tycon*. By applying the projection constructor of *'f* to the representation of *'a*, we get a new projection. We define the resulting type to be isomorphic to the subset of *U* that corresponds to this projection.[1]

typedef (**open**) $('a,'f)\ \mathit{App}$ (**infixl** $\odot$ *65*)
$= \{x.\ x ::: \mathit{tc}\ \mathit{TYPE}('f{::}\mathit{tycon}) \cdot (\mathit{rep\text{-}of}\ \mathit{TYPE}('a{::}\mathit{rep}))\}$

defs
$\mathit{emb} :: 'a \odot 'f \to U \equiv \Lambda\ x.\ \mathit{Rep\text{-}App}\ x$
$\mathit{proj} :: U \to 'a \odot 'f \equiv \Lambda\ x.\ \mathit{Abs\text{-}App}\ (\mathit{cast} \cdot (\mathit{tc}\ \mathit{TYPE}('f) \cdot (\mathit{rep\text{-}of}\ \mathit{TYPE}('a))) \cdot x)$

lemma *rep-of-App*:
$\mathit{rep\text{-}of}\ \mathit{TYPE}('a \odot 'f) = \mathit{tc}\ \mathit{TYPE}('f) \cdot (\mathit{rep\text{-}of}\ \mathit{TYPE}('a))$

Since $\lambda x.\ x ::: D$ is an admissible predicate, and holds for $\bot$, it is easy to show that the resulting type is a pcpo. Also notice the infix syntax for the *App* type constructor: We use postfix application order to be consistent with Isabelle's type syntax.

4 Axiomatic Constructor Classes

4.1 Coercion

We define a function *coerce* to convert values between any two representable types. The *coerce* function will be useful in the next section for defining polymorphic constants in axiomatic type classes.

[1] For convenience, Isabelle's typedef package would normally try to define a constant *App* to be equal to the set specified in the type definition. That would cause an error in this case, because while the value of the set depends on the type variables *'a* and *'f*, those type variables do not appear in the type of the set itself. The keyword **open** overrides this behavior, so that the typedef package does not define such a constant.

constdefs
  coerce :: $('a::rep) \to ('b::rep)$
  coerce $\equiv$ *proj oo emb*

Now we establish some properties of *coerce*. Coercing from a smaller type to a larger type is always invertible, while coercing from a larger type to a smaller type is only invertible under some conditions. Applying a coerced function is equivalent to coercing before and after applying the original function. Finally, the *coerce* function may degenerate into *emb*, *proj*, or *ID*, depending on the type at which it is applied.

lemma *coerce-inv1*: $\mathit{rep\text{-}of}\ \mathit{TYPE}('a) \sqsubseteq \mathit{rep\text{-}of}\ \mathit{TYPE}('b)$
$\Longrightarrow \mathit{coerce}\cdot(\mathit{coerce}\cdot x :: 'b) = (x::'a)$
lemma *coerce-inv2*: $\mathit{emb}\cdot x ::: \mathit{rep\text{-}of}\ \mathit{TYPE}('b) \Longrightarrow \mathit{coerce}\cdot(\mathit{coerce}\cdot x :: 'b) = (x::'a)$
lemma *coerce-cfun*: $\mathit{coerce}\cdot f = \mathit{coerce}\ \mathit{oo}\ f\ \mathit{oo}\ \mathit{coerce}$
lemma *coerce-ID*: $\mathit{coerce}\cdot(\mathit{ID}::'a \to 'a) = \mathit{cast}\cdot\mathit{rep\text{-}of}\ \mathit{TYPE}('a)$

4.2 Functor Class

The first axiomatic type class that we define is the *functor* class. Instances of this class should have a polymorphic function *fmap* (similar to *map* for lists) that preserves the identity and function composition. Here are the type signature and theorems that we would like to have for *fmap*:

consts *fmap* :: $('a \to 'b) \to 'a\odot'f \to 'b\odot('f::\mathit{functor})$
theorem *fmap-ID*: $\mathit{fmap}\cdot\mathit{ID} = \mathit{ID}$
theorem *fmap-comp*: $\mathit{fmap}\cdot(f\ \mathit{oo}\ g) = \mathit{fmap}\cdot f\ \mathit{oo}\ \mathit{fmap}\cdot g$

The above theorems are not suitable for use as class axioms, because they each have multiple free type variables: *fmap-ID* has two and *fmap-comp* has four, counting *'f*. We could try to emulate a 4-parameter type class using predicates, but when reasoning about functors in general the number of extra assumptions would quickly get out of hand. We really need a set of class axioms with only *one* free type variable, the type constructor *'f*. Furthermore, the class axioms should ensure that the above laws hold at all instances of the other type variables. If Isabelle supported nested universal quantification of type variables [13] this would be simple to express, but we must find another way.

Our solution is to express the functor laws in an untyped setting, by replacing universally-quantified type variables with U, the universal domain type. In this setting we can model type quantification using quantification over the *U projection* type. Along these lines, we declare an "untyped" version of *fmap* called *rep-fmap*—the polymorphic *fmap* is defined by coercing it to more specific types.

consts *rep-fmap* :: $(U \to U) \to U\odot'f \to U\odot'f$
defs *fmap* $\equiv$ $\mathit{coerce}\cdot\mathit{rep\text{-}fmap}$

The *functor* class axioms are all in terms of *rep-fmap*. We need three altogether: one for each of the two functor laws, and one to assert that *rep-fmap* has an appropriate polymorphic type. We just need to decide what forms the laws should take. First consider the composition law *fmap-comp*: If we convert it to the untyped setting and make all quantification explicit, we get something like the following (abusing notation slightly):

theorem *rep-fmap-comp*:
$\forall\, a\ b\ c{::}U\ projection.$
$\quad \forall f{:::}b{\to}c.\ \forall g{:::}a{\to}b.\ rep\text{-}fmap{\cdot}(f\ oo\ g) = rep\text{-}fmap{\cdot}f\ oo\ rep\text{-}fmap{\cdot}g$

Because the *U projection* variables are only used to restrict the quantification of f and g, we can simplify this rule by removing the *U projection* quantifications, and allowing f and g to range over all functions of type $U \to U$. We end up with something that looks exactly like the original *fmap-comp* law, but at a more specific type.

The identity law *fmap-ID* works out differently, because it mentions a specific function value *ID* instead of using universal quantification. When we convert this rule to the untyped setting, we obtain terms of the form *coerce·ID*, which simplify to applications of *cast*.

axclass *functor* $\subseteq$ *tycon*
rep-fmap-type:
$\quad [\![\bigwedge x.\ x\ {:::}\ D \Longrightarrow f{\cdot}x\ {:::}\ E;\ emb{\cdot}xs\ {:::}\ tc\ TYPE('f){\cdot}D]\!]$
$\quad\quad \Longrightarrow emb{\cdot}(rep\text{-}fmap{\cdot}f{\cdot}xs)\ {:::}\ tc\ TYPE('f){\cdot}E$

rep-fmap-ID:
$\quad rep\text{-}fmap{\cdot}(cast{\cdot}D) = proj\ oo\ cast{\cdot}(tc\ TYPE('f){\cdot}D)\ oo\ emb$

rep-fmap-comp:
$\quad rep\text{-}fmap{\cdot}(f\ oo\ g) = rep\text{-}fmap{\cdot}f\ oo\ rep\text{-}fmap{\cdot}g$

Deriving the polymorphic versions of the functor laws is quite straightforward. It basically consists of unfolding the definition of *fmap*, and then using the *coerce* lemmas together with the *functor* class axioms to finish the proofs.

4.3 Functor Class Instances

In this section we describe the recommended method for establishing instances of the functor class. In typical usage, we expect the user to have defined an ordinary Isabelle type constructor—such as *llist*, for lazy lists—which is representable and has a function *map* that satisfies the functor laws. Our remaining task is to define a *tycon* instance that models this type constructor, and prove that it is an instance of the functor class.

To obtain the projection constructor that models *llist*, we need to derive a formula for *rep-of TYPE('a llist)* in terms of *rep-of TYPE('a)*. This is straightforward as long as *emb* and *proj* are defined in a reasonable way for *llist*: Embedding a value of type *'a llist* should be the same as first mapping *emb* over the list, and then embedding the resulting *U llist*. Similarly, projecting a value of type *'a llist* should be the same as projecting from U to *U llist*, and then applying *map·proj*.

```
rep-of TYPE('a llist)
  = Abs-projection (emb oo proj)
  = Abs-projection (emb oo map·emb oo map·proj oo proj)
  = Abs-projection (emb oo map·(emb oo proj) oo proj)
  = Abs-projection (emb oo map·(cast·(rep-of TYPE('a))) oo proj)
```

Accordingly, we define a function *functor-tc* that produces a projection constructor from any map function. We will typically instantiate the type variable *'l* to some type constructor applied to *U*, for example *U llist*.

```
constdefs
  functor-tc :: ((U → U) → 'l → 'l) ⇒ U projection → U projection
  functor-tc map ≡ ΛD. Abs-projection (emb oo map·(cast·D) oo proj)
```

Next we declare a type which will be an instance of class *tycon*. Since values of types in class *tycon* are never used for anything, it does not matter how we actually define the type—a singleton type works fine. By our convention, we use a capitalized version of the name of the original type constructor.

```
datatype LList = DummyLList  — exact definition does not matter
instance LList :: tycon ..
defs
  tc (t::LList itself) ≡ functor-tc (map::(U → U) → U llist → U llist)
  rep-fmap::(U → U) → U⊙LList → U⊙LList ≡ coerce·map
```

The above two definitions, together with proofs of the functor laws for *map* at type $(U \to U) \to U\ llist \to U\ llist$, are sufficient to prove the functor class axioms for type *LList*. To facilitate proof reuse, we define a predicate to encode this set of assumptions.

```
constdefs
  functor-locale :: ('f::tycon) itself ⇒ ((U → U) → 'l → 'l) ⇒ bool
  functor-locale (t::'f itself) map ≡
      (tc TYPE('f) = functor-tc map) ∧
      ((rep-fmap :: (U → U) → U⊙'f → U⊙'f) = coerce·map) ∧
      (map·ID = ID) ∧
      (∀f g. map·(f oo g) = map·f oo map·g)
```

Using only this predicate as an assumption, we can prove each of the functor class axioms. There are a couple of important intermediate theorems: First, that the argument to *Abs-projection* in the definition of *functor-tc* is actually a projection. Second, that the types $U \odot 'f$ and $'l$ are represented by the same projection—this means *coerce* is an isomorphism between the two. Using these lemmas together with the properties of *coerce*, it is then relatively straightforward to prove the three functor class axioms for type *'f*. This means that to establish a functor class instance, the user only needs to define the *tycon* and *rep-fmap* in the standard way, and then prove that the functor laws hold at a single type instance.

To show that *LList* is an instance of class *functor*, our only proof obligation is to show *functor-locale TYPE*$(LList)$ *map*. Once we have shown that *LList* is in class functor, we can now use *fmap* at type $('a \rightarrow 'b) \rightarrow 'a{\odot}LList \rightarrow 'b{\odot}LList$. We can also instantiate the functor laws *fmap-ID* and *fmap-comp* (or any other theorem proved about functors in general) to the *LList* type. Besides *fmap*, there are no constants or operations defined on the new *LList* type; however, we can always create values or operations on the type $'a{\odot}LList$ by coercion from the type $'a\ llist$, since the two types are isomorphic.

4.4 Monad Class

The *monad* class specifies two polymorphic constants, *return* and *bind*, with a set of three monad laws that they should satisfy. In addition, a fourth law should be satisfied by monads that are also instances of the *functor* class. Here are the type signature and theorems that we would like to have for class *monad*.

consts $return :: 'a \rightarrow 'a{\odot}('m{::}monad)$
consts $bind :: 'a{\odot}('m{::}monad) \rightarrow ('a \rightarrow 'b{\odot}'m) \rightarrow 'b{\odot}'m$ (**infixl** $\triangleright$ *55*)
theorem *monad-left-unit*: $(return{\cdot}x \triangleright f) = (f{\cdot}x)$
theorem *monad-right-unit*: $(m \triangleright return) = m$
theorem *monad-bind-assoc*: $((m \triangleright f) \triangleright g) = (m \triangleright (\Lambda\ x.\ f{\cdot}x \triangleright g))$
theorem *monad-fmap*: $fmap{\cdot}f{\cdot}xs = xs \triangleright (\Lambda\ x.\ return{\cdot}(f{\cdot}x))$

The functor and monad laws are closely connected. If we use the *monad-fmap* law as a definition for *fmap*, then the functor laws can be proved from the monad laws. Alternatively, we can define *monad* as a subclass of *functor*; in this case, the right unit law is redundant. We have chosen the subclass method, because it allows us to reuse some proofs from the *functor* class.

The definition of the *monad* class follows the same basic pattern as the *functor* class. We start by declaring overloaded representative versions of *return* and *bind*, where all polymorphic type variables are replaced with U. As with the functor class, the real *return* and *bind* are defined by coercion from these.

The *monad* class has five axioms in total: one each for the types of return and bind, and three more for the *monad-left-unit*, *monad-bind-assoc*, and *monad-fmap* rules. All three of these class axioms look exactly like the original rules, because (as with *fmap-comp*) all free type variables are attached to universally quantified values.

Our standard method for establishing instances of class *monad* is similar to the method for class *functor*. We define a predicate *monad-locale* that encodes all of the assumptions necessary to prove the *monad* class axioms. This predicate includes all the assumptions from *functor-locale*, plus additional assumptions stating that *rep-bind* and *rep-return* are defined by coercion, and that the monad laws hold for the underlying type.

4.5 Monad Transformers

In addition to simple type constructors like *LList*, our framework can also be used to define type constructors that take additional type arguments, some of which may be type constructors themselves. The trick is to use a type constructor with one or more type arguments as an instance of class *tycon*.

A good example is the state monad transformer, which has a total of three type parameters: The state type $'s$, the inner monad $'m$, and the result type $'a$. We declare $('a,'m,'s)$ *stateT* as a type abbreviation for $'s \to ('a \times 's)\odot 'm$, and define operations *map-stateT*, *return-stateT* and *bind-stateT* on this type in terms of the monad operations of $'m$. We can then use the monad laws for $'m$ to prove that these new operations satisfy the monad laws.

datatype $('m,'s)$ *StateT* = *DummyStateT* — *exact definition does not matter*
instance *StateT* :: (*monad*, *rep*) *tycon* ..
defs *tc* (t::$('m,'s)$ *StateT itself*)
  ≡ *functor-tc* (*map-stateT*::$(U \to U) \to (U,'m,'s)$ *stateT* $\to (U,'m,'s)$ *stateT*)

The above instance declaration says that $('m,'s)$ *StateT* is in class *tycon* if $'m$ is in class *monad* and $'s$ is in class *rep*. After defining *tc*, *rep-fmap*, *rep-return*, and *rep-bind* in the standard way, we can then use the *monad-locale* theorems to establish that $('m,'s)$ *StateT* is also in class *monad*, for any monad $'m$.

instance *StateT* :: (*monad*, *rep*) *monad*

A more complex example is the resumption monad transformer: It is particularly interesting because the datatype is defined with indirect recursion through a monad parameter. The domain package of HOLCF can not define such datatypes; we defined it manually by taking the least fixed-point of a projection constructor. In the definition of *rep-resT*, $(-)_\perp$ and $\oplus$ refer to the projection constructors associated with their respective type constructors.

constdefs
  rep-resT :: (U *projection* $\to$ U *projection*) $\to$ U *projection* $\to$ U *projection*
  rep-resT $\equiv \Lambda M\ A.\ \mathit{fix}\cdot(\Lambda R.\ A_\perp \oplus (M\cdot R)_\perp)$

typedef $('a,'m)$ *resT* = $\{x.\ x$ ::: *rep-resT*$\cdot$(*tc TYPE*$('m)$)$\cdot$(*rep-of TYPE*$('a)$)$\}$

The resulting type satisfies the isomorphism $('a,'m)$ *resT* $\cong$ $'a_\perp \oplus (('a,'m)$ *resT*$\odot 'm)_\perp$. Similarly to the previous example, we define a type $'m$ *ResT* as a member of class *tycon*, and we prove that if $'m$ is in class *monad*, then so is $'m$ *ResT*.

5 Axiomatic Constructor Classes in HOL

All of the framework described so far has been implemented in Isabelle/HOLCF, where every representable type is a pcpo. However, we have also explored the possibility of porting the theories to Isabelle/HOL. Most of it appears to work, albeit with some important restrictions. The major differences between the two versions are summarized in Table 1.

The essential characteristics of the class of representable types are that: (1) The class is closed with respect to several type constructors, and (2) the class

Table 1. Translation between HOLCF and HOL versions of axiomatic constructor classes

Concept	HOLCF	HOL
Representable type constructors	$\rightarrow, \times, \otimes, \oplus, \cdot_\perp$	$\times, +$, restricted $\Rightarrow$
Embedding between types	ep-pair	function with inverse
Encoding of representable type	projection over U	idempotent function
Ordering of type encodings	$\sqsubseteq$ on projections	$\subseteq$ on range sets
Encoding of type constructor	continuous function	monotone function

has an ordering, with a maximal representable type U. Having a maximal representable type is necessary for our method of reasoning about polymorphic constants in constructor classes.

Remember that in HOLCF we can define a universal domain U into which we can embed its continuous function space $U \rightarrow U$. However, this is not possible in HOL, since for any non-trivial type U the full function space $U \Rightarrow U$ has a strictly larger cardinality than U itself. Essentially this means that the full function space type constructor $\Rightarrow$ can not be representable.

It is possible to make $\Rightarrow$ representable in a limited way, by placing extra restrictions on the left type argument. For example, $'a \Rightarrow 'b$ could be representable for all representable types $'b$ and *countable* types $'a$. Isabelle's datatype package can define infinitely-branching trees, which would be a good candidate for a universal type that could represent these function spaces.

6 Related Work

The authors are members of the Programatica project [10], which is building a high assurance software development environment for Haskell98 [11]. Programatica allows users to embed desired correctness assertions and environmental assumptions in Haskell program elements. Assertions can also be annotated with *certificates* that provide evidence of validity, at differing levels of assurance. Example certificates can range from code inspection sign-offs, manually or randomly generated test cases, all the way up to theorem proving invocations. The Programatica environment tracks assertions and the Haskell definitions they depend on, and can re-invoke certificate servers automatically as needed. We are using Isabelle/HOLCF and axiomatic constructor classes to build a certificate server that provides a high level of assurance for validating Programatica assertions, even in the presence of Haskell functions that terminate on only a subset of their inputs.

Papaspyrou and Macos [19] illustrate how monads and monad transformers can provide a modular denotational semantics for a range of programming language features. They define a simple eager language of expressions with side effects (ELSE), and gradually extend the semantics to include side effects, non-deterministic evaluation of side-effects, concurrent execution of side-effects, and culminates in ANSI C style sequence points. The semantics of the language is parameterized on an underlying monad; this allows the desired computational effects to be reconfigured without globally rewriting the language semantics. The desired monad is also constructed modularly, by applying a sequence of monad transformers. The same framework can also be adapted to model structural language features such as procedures, non-local control-flow, and reference values. Papaspyrou has given a denotational semantics for a significant subset of ANSI C using the same methods [18].

Proof assistants such as Coq [4] and MetaPRL [8] whose type systems are based on dependent type theory can encode instances of axiomatic type and constructor classes as records whose fields contain implementations of the class methods, as well as proofs that the methods satisfy the class axioms. Overloaded functions can then be defined as functions that take these records as parameters. This is often called the *dictionary-passing* approach to implementing type classes. It is unclear to us whether the function parameter hiding and type inference heuristics of these proof assistants are sufficient to hide such dictionary passing from the user, as is the case with our implementation of constructor classes in Isabelle/HOLCF.

Theory morphisms are an alternative to axiomatic type classes for allowing theorems to be reused across families of types, and have been implemented in theorem provers such as IMPS [6,5] and PVS [17]. A key advantage of theory morphisms is that multiple morphisms can be defined that target the same type.

Axiomatic constructor classes can be simulated by theory morphisms, provided that the morphism is allowed to instantiate type constructors of arity greater than zero. However, instantiation of morphisms is an operation on theories, rather than terms, and therefore cannot be applied anonymously to subterms. Also, most-general class instantiations for a well-typed term can always be inferred by Isabelle's order-sorted type unification algorithm [15]. Larger Haskell programs rely on this heavily, and it prevents type annotations from swamping the actual code. To our knowledge, no similar capability is available for current theory morphism implementations. However, they could be implemented in principle, if one were willing to specify a "default" morphism for any given type scheme in the same way that class instances are defined.

Isabelle has a lightweight implementation of theory morphisms, called *locales* [3,12]. However, locales can not not instantiate type constructors, so they are unsuitable for modeling constructor classes. A more general theory morphism mechanism has recently been implemented for Isabelle by Johnsen and Lüth [9], that relies on the theorem prover's ability to attach proof objects to theorems. This allows theorems to be safely instantiated, without needing to modify Isabelle's kernel.

7 Conclusion

Using purely definitional means, we have developed a framework within Isabelle/HOLCF that permits abstract reasoning about type constructors. We have formalized the functor and monad type classes and proved several monad instances, including the maybe monad, lazy lists, the error monad, and the state monad. We have also formalized monad transformers for error handling, persistent state, and resumptions.

We have found that our framework works quite well for abstract reasoning about functors and monads in general. Isabelle's type class system neatly encapsulates all the assumptions related to the functor and monad laws.

Our framework still has much room for improvement, though. Even with a library of combinators available, it turns out that constructing *emb* and *proj* functions takes a bit of work for recursive types; this is something that would benefit from automation. It is also unfortunate that in our framework, we end up with two versions of each type constructor, for example *llist* and *LList*. This means that constants and theorems about *llist* must all be transferred over to *LList* one by one. This would benefit from automation as well. Alternatively, it would be nice to have a datatype package that generates *tycon* instances in the first place.

Other directions for future work aim to automate the translation from Haskell-style code into Isabelle definitions. Mechanizing the process of producing Isabelle code for new type classes is one possibility. With a more sophisticated universal domain, it may also be possible to model datatypes that use features like higher-rank polymorphism, which would be valuable for deep embeddings of Haskell semantics.

References

1. S. Abramsky and A. Jung. Domain theory. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 3, pages 1–168. Clarendon Press, 1994.
2. Roberto M. Amadio and Pierre-Louis Curien. *Domains and Lambda Calculi*, volume 46 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1998.
3. Clemens Ballarin. Locales and locale expressions in Isabelle/Isar. In *Types for Proof and Programs, Intl. Workshop TYPES 2003*, volume 3085 of *Lecture Notes in Computer Science*, pages 34–50. Springer, 2003.
4. Yves Bertot and Pierre Castéran. Interactive theorem proving and program development. Coq'Art : The calculus of inductive constructions. Texts in Theoretical Computer Science, page 492, 2004.
5. W. M. Farmer. An infrastructure for intertheory reasoning. In D. McAllester, editor, *Automated Deduction—CADE-17*, volume 1831 of *Lecture Notes in Computer Science*, pages 115–131. Springer-Verlag, 2000.
6. W. M. Farmer, J. D. Guttman, and F. J. Thayer Fábrega. IMPS: An updated system description. In M. McRobbie and J. Slaney, editors, *Automated Deduction—CADE-13*, volume 1104 of *Lecture Notes in Computer Science*, pages 298–302. Springer-Verlag, 1996.
7. C. A. Gunter and D. S. Scott. Semantic domains. In *Handbook of theoretical computer science (vol. B): formal models and semantics*, pages 633–674. MIT Press, 1990.

8. Jason Hickey, Aleksey Nogin, Robert L. Constable, Brian E. Aydemir, Eli Barzilay, Yegor Bryukhov, Richard Eaton, Adam Granicz, Alexei Kopylov, Christoph Kreitz, Vladimir N. Krupski, Lori Lorigo, Stephan Schmitt, Carl Witty, and Xin Yu. MetaPRL — A modular logical environment. pages 287–303.
9. Einar Broch Johnsen and Christoph Lüth. Theorem reuse by proof term transformation. In Konrad Slind, Annette Bunker, and Ganesh Gopalakrishnan, editors, *International Conference on Theorem Proving in Higher-Order Logics TPHOLs 2004*, volume 3223 of *Lecture Notes in Computer Science*, pages 152–167. Springer, September 2004.
10. Mark P. Jones. Programatica web page, 2005. `http://www.cse.ogi.edu/PacSoft/projects/programatica`.
11. Simon Peyton Jones. *Haskell98 Language and Libraries, Revised Report*, December 2002.
12. Florian Kammüller, Markus Wenzel, and Lawrence C. Paulson. Locales - A sectioning concept for Isabelle. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Thery, editors, *12th International Conference on Theorem Proving in Higher-Order Logics TPHOLs'99*, volume 1690 of *Lecture Notes in Computer Science*. Springer, September 1999.
13. Thomas F. Melham. The HOL logic extended with quantification over type variables. In *HOL'92: Proceedings of the IFIP TC10/WG10.2 Workshop on Higher Order Logic Theorem Proving and its Applications*, pages 3–17. North-Holland/Elsevier, 1993.
14. Olaf Müller, Tobias Nipkow, David Von Oheimb, and Oscar Slotosch. HOLCF = HOL + LCF. *Journal of Functional Programming*, 9(2):191–223, 1999.
15. Tobias Nipkow. Order-sorted polymorphism in Isabelle. In Gérard Huet and Gordon Plotkin, editors, *Logical Environments*, pages 164–188. Cambridge University Press, 1993.
16. Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle / HOL, A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.
17. S. Owre and N. Shankar. Theory interpretations in PVS. Technical Report SRI-CSL-01-01, Computer Science Laboratory, SRI International, Menlo Park, CA, April 2001.
18. N.S. Papaspyrou. *A formal semantics for the C programming language*. PhD thesis, National Technical University of Athens, Department of Electrical and Computer Engineering, Software Engineering Laboratory, February 1998.
19. N.S. Papaspyrou and D. Macos. A study of evaluation order semantics in expressions with side effects. *Journal of Functional Programming*, 10(3):227–244, May 2000.
20. Franz Regensburger. HOLCF: Higher order logic of computable functions. In *Proceedings of the 8th International Workshop on Higher Order Logic Theorem Proving and Its Applications*, pages 293–307. Springer-Verlag, 1995.
21. Markus Wenzel. Type classes and overloading in higher-order logic. In *TPHOLs '97: Proceedings of the 10th International Conference on Theorem Proving in Higher Order Logics*, pages 307–322. Springer-Verlag, 1997.