Axiomatic Constructor Classes in Isabelle/HOLCF

Brian Huffman, John Matthews, Peter White
OGI School of Science and Engineering at OHSU

TPHOLs 2005, Oxford, 22–25 August

Motivation: Abstraction over Type Constructors

Map Functions

- Here is an ordinary list type constructor, defined in Haskell.

```
data List a = Nil | Cons a (List a)
```

- It is basically a container type, so we can easily define a map function for it.

```
mapList :: (a -> b) -> List a -> List b
mapList f Nil = Nil
mapList f (Cons x xs) = Cons (f x) (mapList f xs)
```

- We can also define this function in Isabelle, and prove properties about it.

Map Functions (2)

- Here is a tree datatype, defined in Haskell. Each node contains a List of subtrees.

```
data Tree a = Leaf a | Node (List (Tree a))
```

- We can define a map function for this type too.

```
mapTree :: (a -> b) -> Tree a -> Tree b
mapTree f (Leaf x) = Leaf (f x)
mapTree f (Node xs) = Node (mapList (mapTree f) xs)
```

- There is no fundamental reason why we couldn't define these in Isabelle either.

Functor Class

- Similar map functions may be defined for many type constructors.
- The types of map functions all fit the same pattern—only the type constructor is different.
- In Haskell, we can use type classes to define an overloaded map function:

```
class Functor f where
  fmap :: (a -> b) -> (f a -> f b)
```

- This *constructor class* permits abstraction over type constructors.

More Abstraction over Type Constructors

- Instead of hardwiring the List constructor into the tree nodes, maybe we would like to allow any type constructor.

```
data Tree2 f a = Leaf2 a | Node2 (f (Tree2 f a))

mapTree2 :: Functor f =>
    (a -> b) -> Tree2 f a -> Tree2 f b
mapTree2 f (Leaf2 x) = Leaf2 (f x)
mapTree2 f (Node2 y) = Node2 (fmap (mapTree2 f) y)

instance Functor f => Functor (Tree2 f)
    where fmap = mapTree2
```

Haskell/Isabelle

- Haskell treats types and type constructors uniformly:
  - Either may be used
    - in a polymorphic function type
    - as an argument of a datatype
    - as a parameter of a type class
- In Isabelle, only types have this status:
  - Only types may be used as type constructor arguments.
  - Type classes may only quantify over types.
  - Polymorphic function types may have type variables, but not type constructor variables.

Modeling Haskell Features in Isabelle

To help Isabelle catch up with Haskell, we propose a simple solution:

- Use **types** to represent **type constructors**.
- Use a **binary type constructor** to represent **type application**.

Representing Types and Type Constructors

Of course, we need some infrastructure to make this work.

- First of all, we need a way to represent types as values.
- Then we can represent a type constructor as a function from types to types.
- Finally we need a way to associate such a function to a type variable.

Representing Types with Universal Domains

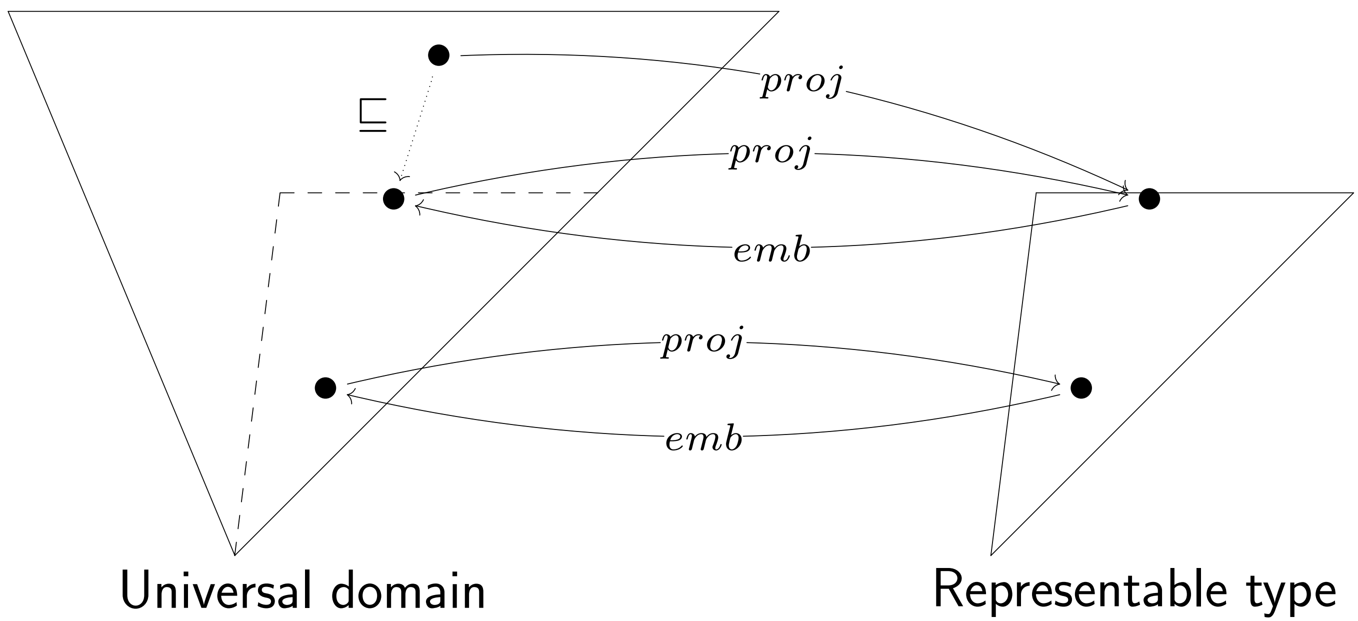
- The following Haskell datatype is an example of a *universal domain*:

```
data U = UInt Int | UProd U U | UFun (U -> U)
```

- It is possible to encode a value of any Haskell datatype as a value of type `U`.

- We can define a similar universal domain type in Isabelle/HOLCF.
  - Using the continuous function space avoids cardinality problems

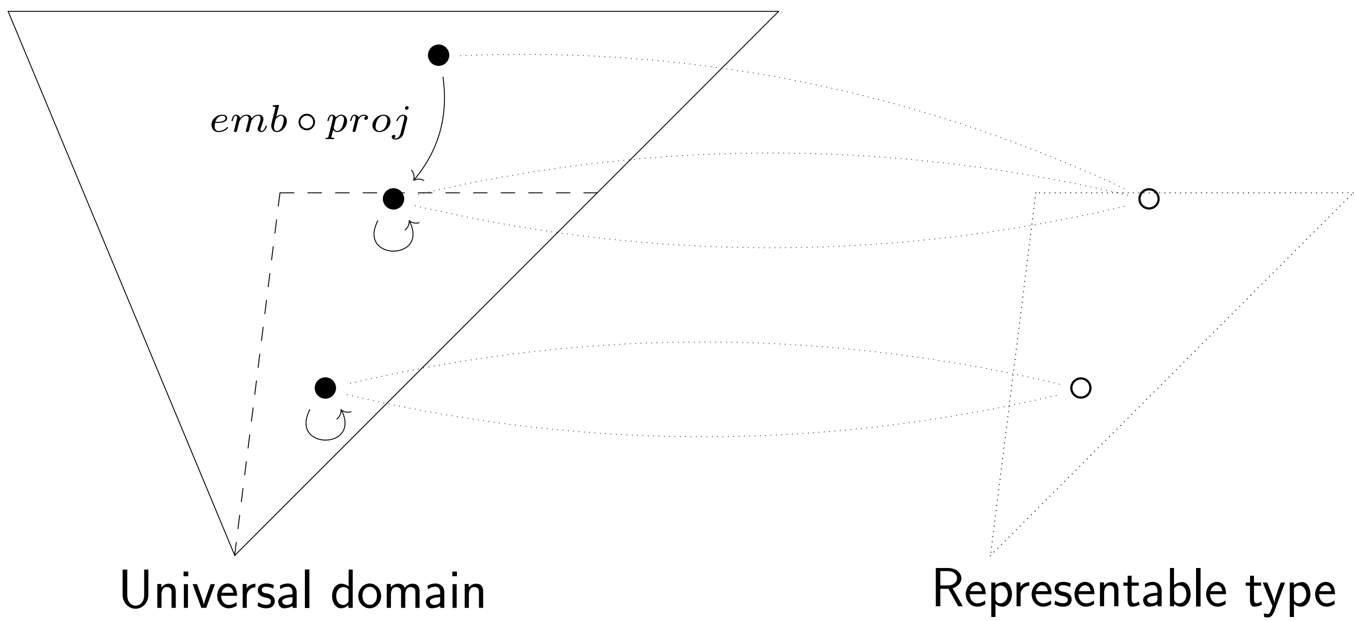
EP-Pairs and Representable Types



Overloaded functions `emb::'a -> U` and `proj::U -> 'a`

Class constraint `'a::rep` means `emb`, `proj` are an ep-pair.

Representing Types with Projections



Composing `emb` and `proj` gives a projection over `U`.

Type `U projection` is a predicate subtype of `U -> U`.

Overloading: Mapping from Types to Values

- For each type `'a::rep` (representable type)
  - Associate a value `REP('a)::U projection` that represents `'a`
  - Defined in terms of `emb` and `proj`
  - `REP('a)` is actually sugar for something like `REP(arbitrary::'a)`, a function applied to a dummy argument
- For each type `'f::tycon` (type constructor)
  - Associate a value `TC('f)::U projection -> U projection`
  - User-supplied definition for `TC` at each type instance
  - No class axioms—in fact, we never use values of type `'f`.

Type Constructor for Type Application

- Definition of type constructor `App` for explicit type application
  - Here `x:::A` means "`x` is a fixed-point of projection `A`"

```
typedef ('a,'f)App (infixl "$")
    = "{x. x ::: TC('f::tycon) (REP('a::rep))}"
```

- This type definition satisfies the property

```
REP('a$'f) = TC('f) (REP('a))
```

Summary: Modeling Type Constructors

Language feature	**Isabelle representation**
value	value of type `U`
type	value of type `U projection`
type variable (kind $*$)	type variable (class `rep`)
constructor variable (kind $* \longrightarrow *$)	type variable (class `tycon`)
type application	binary type constructor `App`
type class	subclass of class `rep`
type constructor class	subclass of class `tycon`

Adding Axioms to Constructor Classes

- The development so far allows some abstraction over type constructors:
  - Overloaded constants may have type constructor variables in their types.
  - Type constructors may take tycons as arguments.
  - We can declare subclasses of `tycon`: i.e. type constructor classes.
- But so far we can not declare very useful class axioms for type constructor classes...

Example: Functor Class

- Constant and theorems we would like to have for an axiomatic class of functors:

```
fmap :: ('a -> 'b) -> 'a$'f -> 'b$'f

theorem functor_id:
  fmap (id::'a -> 'a) = (id::'a$'f -> 'a$'f)

theorem functor_comp:
  ∀(f::'b -> 'c) (g::'a -> 'b).
    fmap (f ∘ g) = (fmap f) ∘ (fmap g)
```

- Problem: Isabelle class axioms can only mention *one* type variable.

Functor Class Axioms

- To get rid of extra type variables, use "untyped" setting
  - Replace extra type variables with `U`
  - Define polymorphic `fmap` by coercion from "untyped" version

```
fmap :: ('a -> 'b) -> 'a$'f -> 'b$'f ==
  proj (emb (rep_fmap :: (U -> U) -> U$'f -> U$'f))
```

Functor Class Axioms (2)

- Class axioms defined in terms of "untyped" `rep_fmap`
  - One class axiom for each functor law
  - Additional class axiom to specify type of `rep_fmap`
- Functor laws are polymorphic, and should hold in *all* type instances.
  - Can model quantification over *types* using quantification over *values*
- Original laws about `fmap` are easily derived from the class axioms.

Summary

- We have extended Isabelle with a type constructor for explicit type application
- Many notions of domain theory have been formalized, to support representations of types
- Can reason abstractly about type constructor classes like functors and monads
  - Functor and monad laws are real theorems in Isabelle
  - Purely definitional—no new axioms!
- Isabelle provides full type inference, with type constructor variables and classes

Further Directions

- We have also implemented:
  - Monad type constructor class
  - Several class instances (Maybe, List, etc.)
  - Monad transformers (State, Error, Resumption)
  - Higher-order type constructors (kinds besides $* \to *$)
  - Constructor classes for Isabelle/HOL

- Future work:
  - Datatype package support for defining `emb`, `proj` automatically
  - More automation for defining `tycons`
  - Systematic way of adding new constructor classes