A Deflation Model for Haskell Types

Brian Huffman

Portland State University

October 10, 2008

Semi-formal reasoning with Haskell

Theorem

For all xs, `map id xs = xs`

Proof.

By induction on `xs`.

- Base case ($\perp$): `map id` $\perp$ `=` $\perp$
- Base case (`[]`): `map id [] = []`
- Inductive step (`x : xs`):
  Assume `map id xs = xs`.
  Then `map id (x : xs)`
  `= id x : map id xs = x : xs`

□

Semi-formal reasoning with Haskell

- We want to make these kinds of proofs more rigorous
- Not just for lists, but for all datatypes
- What about induction over types like this:

```
data ResT m a = Done a
              | Resume (m (ResT m a))
```

- We need to have a precise semantics for Haskell datatypes!

In this talk, I will construct a model for types using *deflations*, a certain kind of idempotent function over a universal domain.

Class of Representable Types

A class of “representable” types

```
class Rep a where
  emb :: a -> U
  prj :: U -> a
```

- `U` is some “universal datatype”
- Intention: Every datatype definable in Haskell should be representable

Universal datatype

A universal datatype

```
data U = Con Int [U]
       | Fun (U -> U)
```

- Ordinary algebraic datatypes (sums of products) only need to use `Con`
  - Integer tag identifies which constructor
  - Constructor arguments in a list
- Embedding function space requires `Fun`

Example instance: Bool

```
instance Rep Bool where

  emb True  = Con 1 []
  emb False = Con 2 []

  prj (Con 1 []) = True
  prj (Con 2 []) = False
  prj _          = undefined
```

- Each constructor is assigned a number
- `prj` is inverse of `emb`
- `prj` is as "undefined" as possible

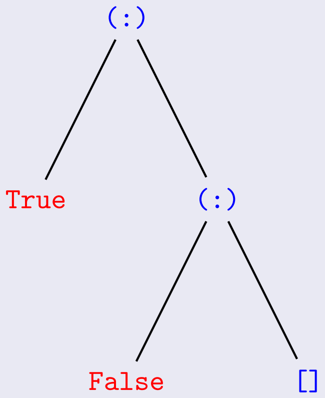
Example instance: Lists

Recursive types can use recursive emb/prj functions

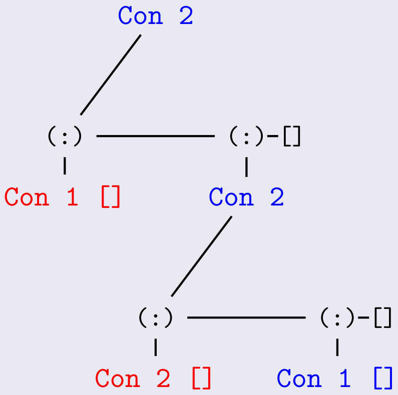
```
instance Rep a => Rep [a] where

  emb[a] []         = Con 1 []
  emb[a] (x : xs)   = Con 2 [emba x, emb[a] xs]

  prj[a] (Con 1 [])         = []
  prj[a] (Con 2 [x, xs])    = prja x : prj[a] xs
  prj[a] _                  = undefined
```

What does the embedding look like?

Before



After



Example instance: Trees

Indirect recursion: Haskell system implicitly constructs dictionaries

```
data Tree a = Node a [Tree a]

instance Rep a => Rep (Tree a) where

  emb_(Tree a) (Node x ts) =
                Con 1 [emb_a x, emb_[Tree a] ts]

  prj_(Tree a) (Con 1 [x,ts]) =
                Node (prj_a x) (prj_[Tree a] ts)
  prj_(Tree a) _ = undefined
```

Example instance: Functions

Negative recursion:

```
instance (Rep a, Rep b) => Rep (a -> b) where

  emb(a→b) f = Fun (emb_b . f . prj_a)

  prj(a→b) (Fun f) = prj_b . f . emb_a
  prj(a→b) _       = undefined
```

- In previous examples, emb calls only emb, prj calls only prj
- The type "a -> b" has variable "a" in a negative position
- Thus $\mathtt{emb}_{(a \to b)}$ calls $\mathtt{prj}_a$, and $\mathtt{prj}_{(a \to b)}$ calls $\mathtt{emb}_a$

Representing types

- So far, we know how to represent values of any type as values of type `U`
- How can we represent Haskell *types* themselves as values of some other type?

First try: Types as subsets of U

The full powerset of U is a cpo, has fixed points of monotone operators

- Problems with function space!

$$A \rightarrow B = \{\mathtt{Fun}\, f \,.\, \forall x \in A.\, f(x) \in B\}$$

- Contravariant in A: as the set A gets larger, the set $A \rightarrow B$ gets smaller
- Limited to positive recursion (like Coq)
- Also, this model is not extensional

Representing subsets as idempotent functions

- Function t representing a set A:
  - For $x \in A$, $t(x) = x$
  - For $x \notin A$, $t(x)$ returns something that is in A
  - Think: "Is this value in the set? If so, then OK. If not, then give me something else that is."
- For idempotent functions, range = set of fixed-points [Proof left to the reader]
- Note: Not all subsets can be represented by a *continuous* idempotent function

Idempotents for representable types

- For each (representable) Haskell type `a`, can we define an idempotent function `t :: U -> U` whose range is equal to the range of $\texttt{emb}_{\texttt{a}}$?
- Given type `a`, a suitable function is $(\texttt{emb}_{\texttt{a}}\ .\ \texttt{prj}_{\texttt{a}})$
- Easily verify that $(\texttt{emb}_{\texttt{a}}\ .\ \texttt{prj}_{\texttt{a}})$ is idempotent

Example idempotent: Bool

Recall definitions of $\texttt{emb}_{\texttt{Bool}}$ and $\texttt{prj}_{\texttt{Bool}}$:

```
emb_Bool True  = Con 1 []
emb_Bool False = Con 2 []
prj_Bool (Con 1 []) = True
prj_Bool (Con 2 []) = False
prj_Bool _          = undefined
```

Idempotent function tBool $=$ ($\texttt{emb}_{\texttt{Bool}}$. $\texttt{prj}_{\texttt{Bool}}$)

```
tBool :: U -> U
tBool (Con 1 []) = Con 1 []
tBool (Con 2 []) = Con 2 []
tBool _          = undefined
```

Example idempotent: Lists

Type abbreviation for idempotents:

```
type T = U -> U
```

Function `tList` represents `[]` type constructor

```
tList :: T -> T
tList a (Con 1 [])       = Con 1 []
tList a (Con 2 [x, xs]) = Con 2 [a x, tList a xs]
tList a _                = undefined
```

Satisfies $\texttt{tList (emb}_{\texttt{a}}\ \texttt{. prj}_{\texttt{a}}\texttt{) = (emb}_{\texttt{[a]}}\ \texttt{. prj}_{\texttt{[a]}}\texttt{)}$

What does behavior of tList look like?

Applied to the result of (emb [True, False]):

```
tList tBool $
  Con 2 [Con 1 [], Con 2 [Con 2 [], Con 1 []]]
= Con 2 [Con 1 [], Con 2 [Con 2 [], Con 1 []]]
```

Applied to an ill-formed argument:

```
tList tBool $
  Con 2 [Con 3 [], Con 2 [Con 2 [], Con 4 []]]
= Con 2 [⊥, Con 2 [Con 2 [], ⊥]]
```

Definition of deflations

A *deflation* (aka projection) is a function t such that for all x,

- $t(t(x)) = t(x)$, i.e. t is idempotent
- $t(x) \sqsubseteq x$, i.e. output is an approximation of the input

Function `tBool` is a deflation; `tList` maps deflations to deflations

Theorem

Subset ordering on deflations $\Longleftrightarrow$ domain ordering on functions

Corollary

The set determines the deflation—there is at most one deflation corresponding to any given set.

Example deflation: Trees

Function `tTree` represents `Tree` type constructor

```
tTree :: T -> T
tTree a (Con 1 [x, ts])
          = Con 1 [a x, tList (tTree a) ts]
tTree a _ = undefined
```

- Implicit dictionary construction stuff that we had with `emb` and `prj` is made explicit here
- `tTree` is just an ordinary recursively-defined function
- Definition is simple enough for even Isabelle/HOLCF to handle

Example deflation: Functions

Function `tFun` represents `(->)` type constructor

```
tFun :: T -> T -> T
tFun a b (Fun f) = Fun (b . f . a)
tFun a b _       = undefined
```

- No distinction needed between positive/negative occurrences of type variables
- `tFun` is continuous and monotone in both arguments
- Fixed points for all recursive types exist—no restriction to positive recursion!

Using deflations in HOLCF

- In Haskell, we defined types first, and deflations later
- In HOLCF, we will define deflations first, then types

Modeling in HOLCF

1. We define a subtype T of $U \rightarrow U$ which contains only deflations
2. Define T constructors representing sums, products, function space, etc.
3. Define recursive datatypes in terms of those, using `fixrec` package
4. Define an Isabelle type corresponding to any deflation, using Isabelle's sub-type definition mechanism

- Note that the range of a deflation is automatically a cpo!
- Now we should be able to construct any recursive Haskell datatype in Isabelle/HOLCF!

Bootstrapping Problem

Wait a minute—where does type U come from?

- Above, we used Haskell's recursive datatype facility to define type `U`
- But Haskell's recursive datatype facility is what we are trying to give a semantics for! This is circular reasoning!
- It is possible to build a universal domain another way.
- (I have done this as well, but it is the subject of another talk!)

Conclusions

Benefits of deflation model:

- Handles function space no problem! (->) is mono in both arguments.
- Can model all kinds of wacky recursive datatypes

Drawbacks of deflation model:

- Only does pointed types
- (Possibility: powerdomains for modeling unpointed types)

The End

Thank you

Appendix: Existential types

The deflation model can even handle datatypes with existential type quantification, if the universal domain U meets one more condition: The type T of deflations over U must be representable.

```
data Expr a
  = Val a
  | forall b . Apply (Expr (b -> a)) (Expr b)

tExpr :: T -> T
tExpr a (Con 1 [x]) = Con 1 [a x]
tExpr a (Con 2 [x, y, z])
    = Con 2 [tT x, tExpr (tFun b a) y, tExpr b z]
  where b = prj_T x
```