Deflation Chains and Induction Rules for Haskell Types

Brian Huffman

Portland State University

November 7, 2008

In this talk, I will show how to derive induction rules for recursive datatypes constructed with the deflation model.

Semi-formal reasoning with Haskell

Theorem

For all xs, `map id xs = xs`

Proof.

By induction on `xs`.

- Base case ($\bot$): `map id` $\bot$ `=` $\bot$
- 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`

$\square$

Semi-formal reasoning with Haskell

We want to make these kinds of proofs more rigorous

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

In last month's talk I showed how to define datatypes in HOLCF

- using deflations over a universal domain
- finding solutions to domain equations

But I have not yet shown how to derive induction rules

- domain equations may have multiple solutions
- we want to avoid "junk" elements

Review: Class of Representable Types

A “universal” Haskell datatype

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

A class of “representable” types

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

Review: Embedding-Projection Pairs

```
instance Rep Bool where

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

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

emb and prj satisfy the definition of an *embedding-projection pair*

- ∀x::Bool, prj (emb x) = x
- ∀y::U, emb (prj y) ⊑ y

Review: Deflations

```
type Defl a = a -> a

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

`tBool` satisfies the definition of a *deflation*

- $\forall$`x::U, tBool (tBool x) = tBool x`
- $\forall$`x::U, tBool x` $\sqsubseteq$ `x`

`tBool` $=$ `emb . prj` for type Bool

Chains

In domain theory, a *chain*

- is a sequence mapping $\mathbb{N}$ to a domain D
- is monotone: $\forall m \leq n,\ x_m \sqsubseteq x_n$
- has a least upper bound in D, written $\bigsqcup_n x_n$

We can model chains in Haskell using a special datatype for natural numbers.

Lazy Naturals

A type of lazy natural numbers

```
data Nat = S Nat

zero :: Nat
zero = undefined

one :: Nat
one = S zero

omega :: Nat
omega = S omega
```

Structure of type Nat

```
      omega
        :
        :
S (S (S ⊥))
        |
   S (S ⊥)
        |
      S ⊥
        |
        ⊥
```

Chains in Haskell

We can model chains in Haskell by functions on `Nat`

```
type Chain a = Nat -> a
```

The least upper bound of a chain is just the "infinitieth" element

```
lub :: Chain a -> a
lub f = f omega
```

Note that any chain we can define in Haskell is monotone by construction.

Chains of Deflations

A *take function* is a chain of deflations over a datatype

```
takeList :: Chain (Defl [a])
takeList (S n) [] = []
takeList (S n) (x : xs) = x : takeList n xs
```

Additional property of a take function

- least upper bound = identity deflation
- for example, note that `lub takeList` = `id`

Induction rule $\Longrightarrow$ take function

How can we prove that `takeList` is really a take function?

```
takeList (S n) [] = []
takeList (S n) (x : xs) = x : takeList n xs
```

Proof of `takeList omega xs` $=$ `xs` by induction on `xs`:

- `takeList omega` $\perp = \perp$
- `takeList omega []` $=$ `[]`
- `takeList omega (x : xs)`
  $=$ `x : takeList omega xs` $=$ `x : xs`

Take function $\Longrightarrow$ induction rule

The converse direction also holds

- `lub takeList` = `id` implies an induction rule

Assume, for some admissible predicate P

- P($\perp$), P(`[]`)
- $\forall$`xs::[a]`, P(`xs`) $\Longrightarrow$ P(`x:xs`)

Proof sketch that $\forall$`xs::[a]`, P(`xs`):

- Show that $\forall$`n::Nat`, `xs::[a]`, P(`takeList n xs`) by induction on `n`
- Hence $\forall$`xs::[a]`, P(`takeList omega xs`)
- Using assumption `lub takeList` = `id`, we have $\forall$`xs::[a]`, P(`xs`)

Take functions and Induction rules

Take-home message for this section:

$$\text{Take functions} \iff \text{Induction rules}$$

Constructing take functions

For defining types in HOLCF, we need to generate take functions for newly-defined types.

- We will define chains of deflations on new types in terms of deflations on type `U`
- Then we will transfer those deflations from `U` onto the new type

Next we will explore the connection between deflations on different types.

Embedding deflations

Given a deflation on type `a`, we can make a deflation on type `U`

```
emb_defl :: (Rep a) => Defl a -> Defl U
emb_defl d = emb . d . prj
```

We can verify `emb_defl d` satisfies the deflation axioms

- `emb_defl d (emb_defl d x) = emb_defl d x`
- `emb_defl d x` $\sqsubseteq$ `x`

Also note that `emb_defl d` $\sqsubseteq$ $\mathtt{emb}_{\mathtt{a}}$ `.` $\mathtt{prj}_{\mathtt{a}}$

Projecting deflations

Given a deflation on type `U`, can we make a deflation on type `a`?

```
prj_defl :: (Rep a) => Defl U -> Defl a
prj_defl t = prj . t . emb
```

prj_defl is the left-inverse of emb_defl

- `prj_defl (emb_defl d) = d`

But does `prj_defl` give back a deflation?

Projecting deflations

Is `prj_defl t` idempotent?

- `prj_defl t (prj_defl t x)`
- = `prj (t (emb (prj (t (emb x)))))`
- = ???

It seems that we are stuck...
unless we make additional assumptions about `t`.

Projecting deflations

We require that `t` $\sqsubseteq$ `emb`$_a$ `. prj`$_a$

- Thus range(`t`) $\subseteq$ range(`emb`$_a$ `. prj`$_a$)
- Thus $\forall$`x::U`, `emb (prj (t x))` $=$ `t x`

Now we can complete the earlier proof

- `prj_defl t (prj_defl t x)`
- $=$ `prj (t (emb (prj (t (emb x)))))`
- $=$ `prj (t (t (emb x)))`
- $=$ `prj (t (emb x))`
- $=$ `prj_defl t x`

Projecting deflations

One final property of prj_defl:

- Assume that `t::Defl U` $=$ $\texttt{emb}_{\texttt{a}}$ `.` $\texttt{prj}_{\texttt{a}}$
- Then `prj_defl t` $=$ `id::a -> a`

Modeling in HOLCF

We will use `prj_defl` to create take functions for new types

1. Define `ct :: Chain (Defl U)`
2. Define `t :: Defl U` = `lub ct`
3. Define type a isomorphic to range(`t`), with `t` = $\texttt{emb}_\texttt{a}$ `.` $\texttt{prj}_\texttt{a}$
4. Define `take_a :: Chain (Defl a)` = `\n -> prj_defl (ct n)`
5. Use take function `take_a` to derive an induction rule

Example: Lists

A deflation representing the list type constructor

```
tList :: Defl U -> Defl U
tList a (Con 1 []) = Con 1 []
tList a (Con 2 [x, xs]) = Con 2 [a x, tList a xs]
```

Instead of defining `tList` directly, we need to define a chain first:

```
ctList :: Defl U -> Chain (Defl U)
ctList a (S n) (Con 1 []) = Con 1 []
ctList a (S n) (Con 2 [x, xs]) =
        Con 2 [a x, ctList a n xs]
```

Note that `ctList` and `takeList` are related by `emb_defl` and `prj_defl`

Fancier example: Trees

A datatype of trees

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

There are some interesting design choices related to indirect recursion

- One approach: treat indirect recursion like mutual recursion
- Alternative: only use functor properties of list constructor

Fancier example: Trees

Treating indirect recursion as mutual recursion

```
takeTree :: Chain (Defl (Tree a))
takeTree (S n) (Node x ts) =
  Node x (takeListTree n ts)

takeListTree :: Chain (Defl [Tree a])
takeListTree (S n) [] = []
takeListTree (S n) (t : ts) =
  takeTree n t : takeListTree n ts
```

These take functions correspond to a mutual induction rule with separate predicates for trees and lists of trees.

Fancier example: Trees

Using functor properties of the list type constructor

```
takeTree :: Chain (Def (Tree a))
takeTree (S n) (Node x ts) =
  Node x (dmapList (takeTree n) ts)

dmapList :: Defl a -> Defl [a]
dmapList d [] = []
dmapList d (x : xs) = d x : dmapList d xs
```

This take function corresponds to a weaker induction rule.
However, this style generalizes more easily

- can define dmap for any type constructor
- can parametrize Tree datatype over list type constructor

Conclusions

Defining types using deflations

- Recursively defined deflations give solutions to domain equations
- Expressing deflations as lubs of chains gives induction rules

Remaining questions

- For some types, more than one induction rule may be possible
- What is the best style to use?

The End

Thank you