COSC 3015: Lecture 8

Lecture given by Prof. Caldwell and scribed by Sunil Kothari

September 18, 2008

1 Recap

We had this lemma last time:

Lemma 1. $\forall k, n : Nat.succ\, k + n = k + succn$

There might be other forms of this lemma:

Lemma 2. $\forall k, n : Nat.succ\, k + n = n + k$

Recall the hint for the HW. How do you know to do induction on the last argument ?

Look at the definition of addition and try to match up

$m + Zero = m$
$m + (Succ\, k) = Succ\, (m + k)$

In above, we have induction on the second argument. Therefore, it pays to do induction on the last argument in the HW. We define Nat as: data Nat = Zero | Succ Nat deriving (Eq,Ord,Show)

Recall that induction principle for the Nat data type

$$(P(Zero) \wedge \forall m : Nat.P(m) \Rightarrow P(Succ\, m)) \Rightarrow \forall m : Nat.P(m)$$

$\perp$ is used to denote loop forever or non-termination or also the program just stops producing an answer (error condition).

undefined1 = undefined1

infinity = Succ infinity

Previously we noted that $\forall a : Type.\perp \in a$ That means, $\perp \in N$. Strictly speaking, we haven't considered bottom in our induction principle.

There are 2 classes of objects in Nat

1. Finite nats - $\{Zero, SuccZero, \cdots\}$
2. Partial nats - $\{\perp, succ\perp, succ(succ\perp), \cdots\}$

If you add infinite number of 1's you will get a natural number - which some students believe is true. There is an infinite natural numbers but infinity is not one of them

The book has *full induction*. You have 3 cases:

1. Case ($\perp$)
2. Case (*Zero*)
3. Case (*Succ* k)

To prove something about finite nats - do case 2 and 3. To prove something about partial nats - do case 1 and 3.

The book's author gives the following example in the book.

Example 1.

Theorem 1. $\forall m : Partial\ Nat. n + m = m$

Proof by induction. Choose arb. $n \in Nat$ and do partial induction on m. $P(m) \stackrel{\text{def}}{=} n + m = m$.

Case $\perp$. Show P($\perp$) i.e. that $n + \perp = \perp$ - by definition of + the evaluator will try to evaluate the second argument - but this loops forever - and so is $= \perp$.

Case *Succ* k. Assume $P(k)$ and show $P(Succk)$ $P(k) = n + k = k$, and $P(succ\ k) = n + (succ\ k) = succ\ k$.

But,

$$\begin{array}{ll} & (n + succ\ k) \\ \stackrel{<<\text{def of } +>>}{=} & succ\ (n + k) \\ \stackrel{<<\text{ind. hyp.}>>}{=} & succ\ k \end{array}$$

□

Q: If m is partial - is m + n = n? It seems to be clear when n is zero but what about succ ?

This is an interesting thing but a bit of hack that we are defining by pattern matching on the second argument but the evaluation takes place — there is a kind of asymmetry.

2 List

The rest of the material in Chapter 3 is all interesting. But we will move on.

2.1 fold

This is just a generalization of a pattern of definitions that we have use for a lot of things. recall we have

$$\begin{aligned} m + Zero &= m \\ m + (Succ k) &= Succ(m + k) \\ m \times Zero &= 0 \\ m \times (succ n) &= (m \times n) + m \\ m \uparrow zero &= Succ zero \\ m \uparrow succ n &= m \uparrow n * m \end{aligned}$$

They all follow the same pattern

$$\begin{aligned} f\ Zero &= c \\ f\ (Succ\ k) &= g\ (f\ k) \end{aligned}$$

$for+$, f is (m+)
c is m
g is Succ

$for\times$, f is (m$\times$)
c is Zero
g is (f m)

This is an aside on a bit of haskell notation called section. For Infix operator $\oplus$ $(\oplus)$ gives a curryied form $(+) :: Nat\ a \Rightarrow a \rightarrow a \rightarrow a$

The mechanism of sections - allows us to add an arguments to the left or right. $(5+) = \lambda x \rightarrow 5 + x$ $(+5) = \lambda x \rightarrow x + 5$

so (m+) above is add m n (m 'add') –end of aside

so we can define $foldn$ as :

$$\begin{aligned} foldn :: (a \rightarrow a) &\rightarrow a \rightarrow Nat \rightarrow a \\ foldn\ g\ c\ zero &= c \\ foldn\ g\ c\ (Succ\ k) &= g\ (foldn\ g\ c\ k) \end{aligned}$$

So what is m+n using fold ? Alternate defintions of $+s$, $*$, $\uparrow$ using foldn

$$\begin{aligned} m + n &\stackrel{\text{def}}{=} foldn\ Succ\ m\ n \\ m \times n &\stackrel{\text{def}}{=} foldn\ (+m)\ Zero\ n \\ m \uparrow n &\stackrel{\text{def}}{=} foldn\ m\ (Succ\ Zero)\ n \end{aligned}$$

Let's compute using this definition

$$\begin{aligned} m\ +_f\ Zero &= foldn\ Succ\ m\ Zero \text{ (defn of +f)} \\ &= m \text{ (defn of foldn)} \end{aligned}$$

$$\begin{aligned} m +_f Succk &= foldn\ Succ\ m\ (Succ\ k) \\ &= Succ\ (foldn\ Succ\ m\ k) \\ &= Succ\ (m +_f k) \end{aligned}$$

Theorem 2. $\forall m, n : Nat. m + n = m +_f n$

Proof. Choose arb. $m \in Nat$ and do induction on n. $P(n) \stackrel{\text{def}}{=} m + n = m +_f n$

Case zero See above.

Case Succ k assume P(k) : $m + k = m +_f k$ Show $P(Succ\ k) : m + Succ\ k = m +_f Succ\ k$ On the left side:
$m + succk = succ\ (m + k)$
On the right side:

$$\begin{aligned} m +_f (succ\ k) &\stackrel{<<\text{defn of } +_f>>}{=} foldn\ succ\ m\ (succ\ k) \\ &\stackrel{<<\text{defn of foldn}>>}{=} succ\ (foldn\ succ\ m\ k) \\ &\stackrel{<<\text{defn of } +_f>>}{=} succ\ (m +_f k) \\ &\stackrel{<<\text{ind. hyp.}>>}{=} succ\ (m + k) \end{aligned}$$

□

What's the definition of doing this ? We have defined this for each of the above defn. We can prove the properties more generally and so we don't need to prove again. The book's author has examples of such properties. It's worth looking at such proofs.

We can define fold for a lot of structure - trees, lists. The defn here is a bit complex because you need to know what fold does. One thing that is slightly annoying you might have to rename plus. For example, use "+" instead of "+++".

3 Chapter 4

In some ways, lists are like the bread and butter of functional programming. Almost all can be modeled using lists.

If we wanted to we can define our own lists. In Haskell, lists are defined as data List a = Nil | Cons a (List a)

But we can define in hasekll with special notations for Nil, Cons and the list type constructor.

data [a] = [] | a:: [a]

So [] ::[a]

(:):: a → [a] → [a]

We can define a predicate *null* as

$$\begin{aligned} null :: [a] &\rightarrow Bool \\ null\ [] &= True \\ null\ (h:t) &= False \end{aligned}$$

$$\begin{aligned} head :: [a] &\rightarrow a \\ head\ [] &= error\ "head\ nil" \\ head\ (h:t) &= h \end{aligned}$$

So what happens when we evaluate head (1:(2:[])) ? It pattern matches the definition with the above term, where

1. h - 1
2. t - (2:[])

We can make a second function

$$\begin{aligned} second\ (h1:h2:t) &= h2 \\ second\ \_ &= error"second : nosecond" \end{aligned}$$

So what does the pattern match ? it matches any list having exactly two elements.