CS3110 Fall 2013 Lecture 17: Specification with Dependent Types (10/29)

Robert Constable

1 Lecture Plan

1. Parameterized types
2. More dependent type examples
3. Interesting types
4. More about induction and recursion
5. Implementing dependent types (inhabitation)
6. Type checking dependent types

Note Well (NB): The topic of dependent types will show up on all the remaining problem sets, on the next prelim, and on the final exam. In the problem sets we will want you to use these types as specifications put into comments.

You will need to know how to create values in dependent types, that is the same as knowing *how to implement them*. You should know how to informally argue that a value is in such a type, as we will illustrate in this lecture. You should be able to translate informal specifications into dependent types.

2 Review, Overview, and Comments

In Lectures 15 and 16 we examined specifications of computational problems using mathematics and using types.

We also examined in more detail the specification given on Prelim 1 for the task of listing all the positions in a given list at which a specific element occurs. We showed how to gradually transform a natural language specification into symbolic language and then into logic, and then into dependent types.

3 Reading and Sources

As mentioned in Lecture 15, logic and verification were covered in lectures 14,15,16, and 28 of CS3110 in 2013sp. The material for this lecture is also related to those lectures as we will see in Lecture 18. The recitation leaders know how to help you with questions about dependent types.

4 Parameterized Types

To create interesting dependent types we need to use parameterized types such as $Date = m : Month \times \{n : Int\ where\ 1 \leq n \leq Num\_days(m)\}$

We need these for all interesting examples of dependent types. Here are more examples of dependent types. For each of them you should be able to pick out the parameterized type.

5 More Dependent Types – seven of them

- `nat = (z : int where 0` $\leq$ `z)`

- `reals = (r : (nat -> rat) * n : nat -> m : nat ->`
  `|`r_n `-` r_m`|` $\leq$ `1/n + 1/m)`

- α `vector(n) = (l :` α `list where len l = n)`

- `sorted_list = (l : int list where ordered l)`

```
ordered l = match l with [] -> true
            | h::t -> match t with
                      | [] -> true
                      | h'::t' ->
                        if h < h' then ordered t
                                  else false
```

- `long_date = (yr : (y : nat where -46 < y <` 10^{10}`) *`
  `(m : month * days(yr, m)) )`

- `n : nat -> (p : nat where prime(p) && n < p)`

- `n : nat -> Even of even(n) | Odd of odd(n)`
  `even(n) = (k : nat where n = 2 * k)`

6 Interesting types – three of them

We've seen currying

- `((`α `*` β`) ->` γ`) ->` α `->` β `->` γ

What about dependent currying?

- `(u : (x :` α `*` β`(x)) ->` γ `u) -> x :` α `-> v :` β `x ->` γ `(x,v)`

- `(y :` α `* (x :` α `->` γ `x y)) -> x :` α `-> (y :` α `*` γ `x y)`

7 Relating induction on $\mathbb{N}$ to terminating recursion on $\mathbb{N}$

Standard Method of Proof

Let $P(n)$ be a proposition about $n \in \mathbb{N}$ (a *parameterized proposition* or $P : \mathbb{N} \to \mathit{Prop}$, a propositional function).

To prove *For all* $n : \mathbb{N}.\, P(n)$ do this:

Prove the *base case* $P(0)$.
Also prove that if we assume $P(u-1)$, for $u > 0$ we can show $P(u)$, i.e.
the *induction hypothesis* $P(u-1)$ for $u > 0$ implies $P(u)$.

That is enough.

The "one line summary" of all this instruction is that we take this formula below as an *axiom*, the *induction axiom*:

$$P(0) \Rightarrow \forall u : \{i : \mathbb{N} \,|\, 0 < i\}.\big(P(u-1) \Rightarrow P(u)\big) \Rightarrow \underline{\forall x : \mathbb{N}.\, P(x)}.$$

8 Expressing the induction rule as a dependent type in OCaml SL

Let `P n` be a *parameterized type* (compare to the math, $P(n)$ a parameterized proposition).

Consider the *dependent type*

```
P 0 -> (u : (i : nat where 0 < i) -> (P(u - 1) -> P u) )
        -> (x : nat -> P x)
```

(Compare to the one line induction axiom
$P(0) \Rightarrow \forall u : \{i : \mathbb{N} \,|\, 0 < i\}.\big(P(u-1) \Rightarrow P(u)\big) \Rightarrow \forall x : \mathbb{N}.\, P(x).$)

The form of these expressions is *remarkably similar*, like the similarity of $A \Rightarrow B$ and α `->` β or of $A \,\&\, B$ and α `*` β. This similarity can act as a guide to an important correspondence between *logical formulas* and (*dependent*) *types*.

9 Recursion as induction

We can define a recursive function with exactly this "one line induction" type

```
P 0 -> (u : (i : nat where 0 < i) -> (P(u - 1) -> P u) )
        -> (x : nat -> P x)
```

```
let rec ind (x : nat) (base : P 0)
            (up : u : (i : nat where 0 < i) -> (P(u - 1) -> P u) )
            : (x : nat -> P x) =
            if x = 0 then base
            else up x (ind (x - 1) base up)
```

* *Notice that the recursive function definition type checks, but the argument that it does uses induction!*

$$\begin{aligned}&\lambda x.\,\text{if } x = 0 \text{ then } \mathit{base}\\ &\quad\;\text{else } \mathit{up}\ x\ (\mathit{ind}\ (x-1)\ \mathit{base}\ \mathit{up})\\ &\in x : \mathit{nat} \to P\,x\end{aligned}$$

By cases if $x = 0$, $\mathit{base} \in P\,0$ by its type.
Assume $(\mathit{ind}\ (x-1)\ \mathit{base}\ \mathit{up}) \in P\,(x-1)$ and up has the right inputs $0 < x$ then $\mathit{up}\ x\ (\mathit{ind}\ (x-1)\ \mathit{base}\ \mathit{up}) \in P\,x$.