CS3110 Fall 2013 Lecture 18: Computing with Dependent Types (10/31)

Robert Constable

1 Lecture Plan

1. More dependent types
2. Execution of dependently typed programs
3. Implementing interesting types
4. Type checking list induction as fold-right

2 Review, Overview, and Comments

In Lecture 17 we examined more simple dependent types, including the type of even numbers and the type of real numbers. Here we will see how to compute with them and explain the (hard) work that a type checker for dependent types must do.

In Lecture 17 we also gave the dependent type for natural number induction and proved that a particular recursive program called *ind* has this type. This shows one way to implement a type that is easy to recognize as the induction rule. In this lecture we will do the same thing for list induction and see that the program for list induction is fold-right.

3 Reading and Sources

As usual, some elements of this lecture, such as a discussion of fold-right can be found in Prof. Kozen's notes, in this case in his 2009 fall, lecture 5 page 3.

4 Examples of OCaml SL Dependent Types

We want to specify the task of taking the square root of either a rational number or a real number and returning as a result a real number. We want our inputs to be positive so that we don't need to define the complex numbers (although this is not beyond the expressive power of OCaml SL, it is just more involved than we want to consider now). When we take the square root of some rationals, such as 4 or 1/4, we get a rational number, but for uniformity, we will inject these into the reals, for example, the real number 2 is simply `fun n -> 2` and 1/2 is `fun n -> 1/2`.

Here are the types we used in lecture.

```
pos_rat = (q:rat where 0 < q)

pos_real = (r:real * (n:nat where r n > 1/n))

number = Rat of pos_rat | Re of pos_real

nbr: number -> (rt:real where rt * rt = nbr)
```

Here is a suggestive informal specification. It has a lot wrong with it, but you might be able to see the idea behind it and then make it more precise.

$$\alpha + \beta \;=\; \textit{Left of } \alpha \mid \textit{Right of } \beta$$

We should take something like @+ to be an infix operator for the disjoint union of the types alpha and beta. It is nicer to just write this as $\alpha + \beta$ instead of using the OCaml convention for designating infix operators.

What do you think this specification might be asking for?

$$(\alpha + \beta)\ list \rightarrow (\#\alpha \star \#\beta).$$

The answer as given in lecture by the class is that it asks for a function to count the number of α elements and the number of β elements and report them as a pair, such as (17,52).

Warning OCaml won't let us write a type such as

$$\alpha + \beta \ = \ Left\ of\ \alpha \mid Right\ of\ \beta.$$

This will give a syntax error. We must instead write a type definition such as

```
type choice = Left of int | Right of bool.
```

We cannot use polymorphic types such as 'a and 'b as in

```
type choice = Left of 'a | Right of 'b
```

nor can we expose the variant as an explicit type. These sensible types will cause a type error for subtle reasons. The type error reported will be totally uninformative and puzzling.

5 Computing with Dependent Types

Now let's suppose that we have defined some dependent types and found what we believe to be elements of them. What happens at run time?

5.1 Type checking at run time

To see what happens when we execute programs with dependent types, let's consider function applications.

`Let even(n) = (k:nat where 2*k = n).`

Consider this definition of the even natural numbers. Recall that
`nat = (z:int where 0 <= z).`

`(n:nat * (k:nat where 2*k = n)).`

Suppose `f` is a function of type
`(n:nat * (k:nat where 2*k = n)) -> nat`

What kind of input must we provide to `f`?

Can we just compute `f 14`? No, this will not type check since `14` is not in the type `(n:nat * (k:nat where 2*k = n))`.

Can we compute `f (14,7)`? Yes, this will type check because we can validate the where clause boolean by checking that `2*7 = 14`.

Do we need to provide `(14, (7,true))` as the input? No, the type checker could check the boolean with the data given.

As we noted in lecture, we could use a specification of even numbers that is easier to type check, say by just asking whether n = 0 mod 2. If this were our definition, then we would only need to provide an even number as input and do the mod 2 check.

Suppose we know `g` has type `real -> real`. What is the correct input?

`g (λn. 1/n)`

Recall that `real =`
`(r : (nat -> rat) * n : nat -> m : nat -> |r`$_n$ `- r`$_m$`| ≤ 1/n + 1/m)`

So do we need

`g ((`λ`n. 1/n), (`λ`n.`λ`m. |`r_n `- `r_m`| `$\leq$ `1/n + 1/m)) ?`

Must the type checker "prove"

`|`r_n `- `r_m`| `$\leq$ `1/n + 1/m ?`

That is prove

`|`r_n `- `r_m`| `$\leq$ `1/n + 1/m =` λ`n.`λ`m. true ?`

The answer is *yes*. This makes for a "smart" type checker indeed.

Typically we get our reals by methods that guarantee this, using $e^x, cos(x), sine(x)$, and starting with a few specific constants, say injecting all the rationals into `real` or building π or other specific real values.

6 Interesting Examples and How to *Implement them*

Implementing currying

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

λ`f.`λ`x.`λ`y. f(x,y)` does the job

where `f :` α `*` β `->` γ
`x :` α
`y :` β
`(x,y) :` α `*` β
`f(x,y) :` γ

Exercise ***Implement uncurrying***

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

Can anyone do it now?

6.1 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)`

Can you find elements of types such as these? This would make a good Prelim II question about dependent types.

Is this an element of that dependent type? A type checking style question for the Prelim.

7 Type checking definitions

We already looked at type checking the function that computes according to the standard mathematical induction rule. We needed induction to type check it. Let's look at the corresponding rule for list induction. The type will look like the rule for structural induction on lists, and the recursive function that implements it is just fold-right.

```
let rec list_ind (l: 'a list) (base: P [])
(step: u:'a -> v:'a list -> w:P v -> P u::v ) : P l =
match l with [] -> base | h::t -> step h t list_ind(t,base,step)
```

How does `list_ind` compute values in type `P(l)`?

Suppose `l = []`

Then `list_ind([], base, step)` $\downarrow$ `base`. *
We know `base` $\in$ `P([])`.

Suppose `l = [a]` for `a` $\in \alpha$.

Then `list_ind([a], base, step)` $\downarrow$
`step(a, [], list_ind([], base, step))`
by * above this is `step(a, [], base)`

What is the type of `step`?

```
step : u : 'a -> v : 'a list -> w : P(v) -> P(u::v)
       ① a     ② []     ③ base

step (①,②,③) ∈ P(a::[])
             ∈ P([a])
step(a, [], base) ∈ P([a])
```

The next `step` could be `[b;a]`
`step(b, [a], step(a, [], base))`

Then for `[c; b; a]`
`step(c, [b; a], step(b, [a], step(a, [], base)))`

This is just *`fold_right`* `step l base` !

8 Background on Dependent Types

The idea of dependent types originated in logic. It was stimulated by the work of Haskell Curry but goes back further to the writings of Heyting and Kolmogorov in the early 1900's and deBruijn in 1970 [3]. The first connections to programming were made at Cornell [2, 1] and in Chalmers

University in Sweden [6] based on work by Howard [4] and Martin-Löf [5]. The book *Lectures on the Curry-Howard Isomorphism* [7] provides a good history of the subject.

References

[1] Robert L. Constable, Stuart F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, Douglas J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, James T. Sasaki, and Scott F. Smith. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, NJ, 1986.

[2] Robert L. Constable and D. R. Zlatin. The type theory of PL/CV3. *ACM Transactions of Programming Language Systems*, 6(1):94–117, January 1984.

[3] N. G. de Bruijn. The mathematical language Automath: its usage and some of its extensions. In J. P. Seldin and J. R. Hindley, editors, *Symposium on Automatic Demonstration*, volume 125 of *Lecture Notes in Mathematics*, pages 29–61. Springer-Verlag, 1970.

[4] W. Howard. The formulas-as-types notion of construction. In *To H.B. Curry: Essays on Combinatory Logic, Lambda-Calculus and Formalism*, pages 479–490. Academic Press, NY, 1980.

[5] Per Martin-Löf. Constructive mathematics and computer programming. In *Proceedings of the Sixth International Congress for Logic, Methodology, and Philosophy of Science*, pages 153–175, Amsterdam, 1982. North Holland.

[6] Bengt Nordström, Kent Petersson, and Jan M. Smith. *Programming in Martin-Löf's Type Theory*. Oxford Sciences Publication, Oxford, 1990.

[7] M.H. Sørensen and P. Urzyczyn. *Lectures on the Curry-Howard Isomoprhism*. Elsevier, 2006.