

Comp 411 Notes - Fri, Mar 25, 2005

Recursive Types

Scribe: Seth J. Fogarty

This lecture concerns the typing of various structures using recursive typing. We discuss strictly and possibly infinite lists, a couple other structures, and type-enforced termination.

1 Strictly infinite lists

Can we construct the type for a strictly infinite list in the call-by-value lambda calculus with pairs, sum types, and recursive typing.

Recall that our syntax, lacking any explicit support for the recursive types, is:

$$e = \lambda x.e|ee|e, e|inj_1e|inj_2e|()$$

and our types are

$$t = 1|t + t|t \times t|t \rightarrow t|\mu\alpha.t|\alpha$$

We will extend the type and syntax for demonstration purposes as necessary, for instance to include natural numbers. We will also omit left and right injection when they would clutter the description of our values.

We wish to create a type that represents an infinite list and is populated by a value in the CBN STLC. Last lecture we had infinite CBN structures, with infinite terms to represent infinite values. Here we need a finite term that represents an infinite list. The definition we will use is to be able to access any element with a finite index.

In the STLC one can easily create lists of finite length using 'pair' as the cons operation. So let us attempt to model an infinite list using this conception of pair: $\mu\alpha.Nat \times \alpha$.

But note that there is no term of this type. While OCaml will let you create cyclic data structures, this is not possible in the STLC. Further any term that was not cyclic would never terminate and generate a value.

So what we need to do is somehow imitate CBN semantics in a CBV language. We need a lazy version of this datatype. But rather than introduce an entire new type, we would like to express this lazy datatype as a schema that converts any CBV structure into something that behaves like a CBN structure.

The problem with $\mu\alpha.Nat \times \alpha$ is the α must be evaluated before we can have a value. So let us make a function that returns the α , and we won't have to evaluate anything immediately. Functions are not evaluated until they are passed the argument, so we can create a function that takes a unit. In this way it delays execution until the results are needed and simulates CBN semantics.

$$\mu\alpha.Nat \times 1 \rightarrow \alpha$$

This is exactly what we were looking for: a way of encoding a lazy value in CBV semantics: $lazyt = 1 \rightarrow t$

There are multiple places we can put this function. In addition to the above, we could make the entire list lazy, instead of just the cons cell. $\mu\alpha.1 \rightarrow (Nat \times \alpha)$.

In the first construction we only lazily evaluate the cons cell. In the second construction we lazily evaluate the entire list.

This does represent an infinite data structure. For any k we can get the k th element. For any k th cell we generate the list that follows it.

2 Possibly infinite lists

$\mu\alpha.1 + Nat \times \alpha$

Totally infinite lists are called streams, but they are not necessarily very useful. What if we want lists of arbitrary lengths. Note that these are not possibly infinite lists, just lists the length of which we don't know. All we do is add a 'null' case. $\mu\alpha.1 + Nat \times \alpha$

If you want possibly infinite lists, you again need to make the cons cell, or the entire structure, lazy. $\mu\alpha.1 + Nat \times (1 \rightarrow \alpha)$

So so far we have the following uses of μ

Arbitrary length lists: $\mu\alpha.1 + Nat \times \alpha$

Strictly infinite list: $\mu\alpha.Nat \times lazy\alpha$

Potentially infinite list: $\mu\alpha 1 + (Nat \times lazy\alpha)$

Recursive types using the μ constructor generate context free grammars. These in particular are actually regular.

Arbitrary length lists: $\mu\alpha.1 + Nat \times \alpha$ is populated by terms that look like $(n)^*1$, if you drop the injections.

Strictly infinite list: $\mu\alpha.Nat \times lazy\alpha$ is populated by n^∞ , if we ignore the functions.

Potentially infinite list: $\mu\alpha 1 + (Nat \times lazy\alpha)$ is populated by the union of the first two.

3 Total functional programmers

One of the features many people dislike about the STLC is that all terms are guaranteed to terminate. There are a group of people, the "total functional programmers" who say "Your programs should always terminate or not terminate. Pick." Most things you run you want to terminate, and be guaranteed to terminate. Others, like you watch, you don't want to terminate.

In a reactive system we want a non-terminating process composed out of terminating interactions. An infinite stream of finite computations. So whereas FRP (functional reactive programming) is a way of encoding reactive programs with no terminate guarantees, EFRP guarantees an infinite stream of bounded terminating interactions.

Note that termination this is something we always want to be true of types: it is called decidability, and it is rather important. So although μ types look like functions, they don't allow us to actually perform any computation. We have to be very careful if we add computation to our type system.

4 Homework for Monday

So terms of the STLC are guaranteed to terminate. We need to add something like the `fix` operator to our language in order to generating non-terminating programs. The reason for this is that the Y combinator is not typeable. If we add recursive typing, can we type the Y combinator.

Homework for Monday: Try to type the Y combinator with μ without looking in the book.

Recall the Y combinator is $\lambda f.(\lambda x.(f(xx))\lambda x.(f(xx)))$

You will need the following equivalence:

$$\mu\alpha.t = t[\alpha := \mu\alpha.t]$$

As an example: $\mu\alpha.1 + \alpha = 1 + \alpha[\alpha := \mu\alpha.1 + \alpha] = 1 + \mu\alpha.1 + \alpha$