

Comp 411 Notes - Wed, Mar 23, 2005

Recursive Types

Scribe: Moez A. Abdel-Gawad
(moez@cs.rice.edu)

This lecture presents recursive types. Recursive types allow typing lists, naturals, and other recursive types. Without recursive types we might have to add the types for lists, naturals, and other recursive types as primitive/base types in our type system, or to resort to other solutions, like having a type for each fixed-length list, and a type for each natural! (Dependent types?)

1 Types We Know So Far

$t ::= \mathbf{1} \mid t + t \mid t \times t \mid t \rightarrow t \mid \{l_i : t_i\}$ (records) $\mid \langle l_i : t_i \rangle$ (variants)

We have six type variants, yet with these types can we type lists so far? No. What about fixed-length lists? Yes we can type them, according to their lengths. All lists (of a specific component type) of length l can be represented by a product type with l components.

2 Typing Lists

For typing all lists, irrespective of their length, however, let's note that (up to isomorphism) natural numbers are lists that carry *unit* in their cells. A list of *units* of length 1 can represent the natural 1. Let's then try to type naturals (as the most primitive lists) without having to add a `Nat` type. This might give us a clue about how to type lists in general.

2.1 Typing Naturals

For naturals, we naturally mathematically speaking, have: $1 = 1$, $2 = 1 + 1$, $3 = 1 + (1 + 1) = 1 + 2$, $3 = 1 + (1 + (1 + 1)) = 1 + 3$ and so on. If we decide to give each value its own (small) type, this recursive definition of naturals suggests some pattern in typing them. Each single value (each natural) will be given the type `unit`. (Up to isomorphism, all single-valued types are equivalent to type `Unit`). Naturals could thus be typed as follows:

$\llbracket \{1\} \rrbracket = \mathbf{1}$ (where $\llbracket \dots \rrbracket$ means the type of ...),
 $\llbracket \{2\} \rrbracket = \mathbf{1}$,
 $\llbracket \{3\} \rrbracket = \mathbf{1}$,

and now comes the interesting part, where we can easily say that

$$\begin{aligned} \llbracket \{1, 2\} \rrbracket &= \mathbf{1} + \mathbf{1} = \llbracket \{1\} \rrbracket + \llbracket \{2\} \rrbracket \\ \llbracket \{1, 2, 3\} \rrbracket &= \mathbf{1} + (\mathbf{1} + \mathbf{1}) = \llbracket \{1\} \rrbracket + \llbracket \{2\} \rrbracket + \llbracket \{3\} \rrbracket \end{aligned}$$

Recalling that, as discussed in previous lectures, the introduction and elimination rules for the sum type constructor (+) are:

$$\frac{\Gamma \vdash e : A}{\Gamma \vdash \mathbf{inl} \ e : A + B} \text{ and } \frac{\Gamma \vdash e : B}{\Gamma \vdash \mathbf{inr} \ e : A + B}.$$

which gives a tag for each type's values, and shows how to type a *match* statement for (finite) variants in general.

We can see now that the set of naturals can be typed if we have an infinite variant type expressed as follows:

$$\llbracket \{1, 2, 3, \dots\} \rrbracket = \llbracket \{1\} \rrbracket + \llbracket \{2\} \rrbracket + \llbracket \{3\} \rrbracket + \dots = \sum_{i=1} \llbracket \{i\} \rrbracket.$$

But, how then can we write a *match* statement on an infinite variant type? A recursive *match* would do. That suggests to us that the input type for the *match* could also be a type that refers to itself, or a recursive type. Applying this to naturals, we could easily see that **Nat**, the type for naturals could be written as:

$$\mathbf{Nat} = \mu\alpha. \mathbf{1} + \alpha,$$

where μ is the recursive *fold* operator *for types*, and α is a type (meta-)variable that stands for the type being defined itself (**Nat** in this case).

It is imperative to note that (in the equi-recursive setting – see next lecture) we have: $\mathbf{Nat} = \mu\alpha. \mathbf{1} + \alpha \cong \mathbf{1} + (\mathbf{1} + (\mathbf{1} + (\dots)))$

We can see thus, that to express recursive types we need to add type variable, and μ -types to the syntax for types: $t ::= \dots \mid \alpha \mid \mu.t$

And now, finally, naturals lists (a **nil**, or a **cons** of a natural and a list) can be typed as:

$$\mathbf{NatList} = \mu\alpha. \mathbf{1} + (\mathbf{Nat} \times \alpha)$$

(The \times is used because a **cons** is a **pair** constructor).

3 More Recursive Types

Since we are talking about *recursive* constructs, and their *fixed points*, we'd also like to see how this interacts with our “type-evaluation” semantics (CBV and CBN). For CBV, our recursive types can represent only arbitrarily large *finite* recursive data structures (e.g., all lists of finite length), while in CBN we can

also type (using the greatest fixed point of a type) *infinitely* large data structures (e.g., streams, and ∞ for naturals).

For CBV, we have our list recursive type stands for either \perp , or for $\mathbf{inr}^*(\mathbf{inl}\ 1)$, while for CBN it stands for \perp , $\mathbf{inr}^*(\mathbf{inl}\ 1)$, or \mathbf{inr}^∞ .

In other words, we have:

CBV	CBN
\perp	\perp
$\mathbf{inr}^*(\mathbf{inl}\ 1)$	$\mathbf{inr}^*(\mathbf{inl}\ 1)$
	\mathbf{inr}^∞

Note also that our syntax for types would allow a valid type of the form $\mu\alpha.\alpha + \alpha$, which stands for the types of $(\mathbf{inl} \mid \mathbf{inr})^\infty$ (a recursive type that types an infinite data structure, that is thus valid only in CBN semantics, and the data structure would be an infinite stream of 0's and 1's). Our types syntax also allows a valid type of the form $\mu\alpha.\alpha \times \alpha$, which is surprisingly isomorphic to *Unit*, as it has one and only one value. That type could be the type of the recursive data structure that is defined by the following pseudo-code: `let p = (p, p) in p`.

Finally, in ML, we can write down the type `NatList` as follows:

```
type NatList = Nil of 1 | Cons of Nat * NatList
```