Lecture Notes on General Recursion & Recursive Types

15-814: Types and Programming Languages
Ryan Kavanagh

Lecture 8
Thursday, September 27, 2018

1 Introduction

To date, our programming examples have been limited to types with no self-referential structure: functions, sums, and products. Yet many useful types are self-referential, such as natural numbers, lists, streams, etc. Not only have our types not exhibited any form of self-reference, but neither have our programs. Today, we will see how to capture recursion in a typed setting, before then expanding our type system with recursive types. Before doing so, we make a brief digression to generalize binary sums (Lecture 6) to finite sums. Though we could encode finite sums as iterated or nested binary sums, the generalization is straightforward and it will allow us to use more descriptive labels for our injections than left "l" and right "r".

2 Finite sums

We generalize the definition of binary sums to finite sums indexed by some finite set I. We begin by extending our syntax as follows:

$$
\begin{array}{ll}
\tau ::= \cdots & \\
\quad \mid \sum_{I}(i : \tau_i) & \text{sum of types } \tau_i \text{ tagged with } i \text{ for } i \in I \\
e ::= \cdots & \\
\quad \mid i \cdot e & \text{inject } e \text{ with tag } i \\
\quad \mid \textbf{case}\ e\ \{i \cdot x_i \Rightarrow e_i\}_{i \in I} & \text{elimination form for finite sums}
\end{array}
$$

We can use this syntax to give a more suggestive definition of the **bool** type:

$$\begin{aligned}\mathbf{bool} &= (t : \mathbf{1}) + (f : \mathbf{1})\\ \mathbf{true} &= t \cdot \langle\,\rangle\\ \mathbf{false} &= f \cdot \langle\,\rangle\\ \mathbf{if}\ e\ \mathbf{then}\ e_t\ \mathbf{else}\ e_f &= \mathbf{case}\ e\ \{t \cdot \_ \Rightarrow e_t \mid f \cdot \_ \Rightarrow e_f\}.\end{aligned}$$

The statics and the dynamics generalize from the binary case in the obvious manner; the reader is referred to [Har16, § 11.2] for details.

3 General recursion

Let us think back to how we implemented recursion in the simply-typed λ-calculus. We wanted to define a recursive function

$$F = \cdots F \cdots ,$$

but found that we could not directly do so because of the circular or self-referential nature of the definition. To get around this, we abstracted out the F on the right hand side

$$F = (\lambda f. \cdots f \cdots)F$$

and observed that we could define F as the fixed point of $\zeta = \lambda f. \cdots f \cdots$. We constructed this fixed point using the fixed point combinator $\mathbf{Y}$, getting

$$F = \mathbf{Y}\zeta = \zeta(\mathbf{Y}\zeta) = \cdots \mathbf{Y}\zeta \cdots = \cdots F \cdots$$

as desired. Though we cannot encode the fixed point combinator in our typed setting, we can introduce a new term former and imbue it with the appropriate semantics. To this end, we introduce a new fixed point construct $\mathbf{fix}(x.e)$, with the intention that, as was the case with $\mathbf{Y}$, we get

$$\mathbf{fix}(f. \cdots f \cdots) = \cdots \mathbf{fix}(f. \cdots f \cdots) \cdots .$$

Its statics are captured by the rule

$$\frac{\Gamma, x : \tau \vdash e : \tau}{\Gamma \vdash \mathbf{fix}(x.e) : \tau}\ (\textsc{Fix})$$

The intention is that x stand for any self-referential use of e in e. The operational behaviour is captured by the rule

$$\frac{}{\mathbf{fix}(x.e) \mapsto [\mathbf{fix}(x.e)/x]e} \;(\text{FIX-STEP})$$

With this construction, we can easily implement a divergent term:

$$\mathit{loop} = \mathbf{fix}(x.x).$$

Then $\mathit{loop} \mapsto [\mathbf{fix}(x.x)/x]x = \mathit{loop}$.

This isn't a very interesting example, so let us consider recursive functions on the natural numbers. Define the type of natural numbers to be

$$\mathbf{nat} \text{ ``=''} \; (\mathsf{z} : \mathbf{1}) + (\mathsf{s} : \mathbf{nat}).$$

This definition is dubious because we are defining **nat** in terms of itself: the type **nat** appears on both sides of the equation and it is unclear that a unique solution exists. We will give a correct definition in section 4, but let us assume the above definition for the sake of illustrating $\mathbf{fix}(x.e)$. We define numerals as follows:

$$\begin{aligned} \overline{0} &= \mathsf{z} \cdot \langle\,\rangle, \\ \overline{n+1} &= \mathsf{s} \cdot \overline{n}. \end{aligned}$$

We can now implement various functions on natural numbers:

$$\begin{aligned} \mathsf{pred} &= \lambda n.\textbf{case}\; n\; \{\mathsf{z} \cdot \_ \Rightarrow \overline{0} \mid \mathsf{s} \cdot n' \Rightarrow n'\} \\ \mathsf{add} &= \mathbf{fix}(f.\lambda n.\lambda m.\textbf{case}\; n\; \{\mathsf{z} \cdot \_ \Rightarrow m \mid \mathsf{s} \cdot n' \Rightarrow \mathsf{s} \cdot (fn'm)\}) \\ \mathsf{mult} &= \mathbf{fix}(f.\lambda n.\lambda m.\textbf{case}\; n\; \{\mathsf{z} \cdot \_ \Rightarrow m \mid \mathsf{s} \cdot n' \Rightarrow \mathsf{add}(m)(fn'm)\}) \\ \mathsf{fact} &= \mathbf{fix}(f.\lambda n.\textbf{case}\; n\; \{\mathsf{z} \cdot \_ \Rightarrow \overline{1} \mid \mathsf{s} \cdot n' \Rightarrow \mathsf{mult}(n)(fn')\}) \end{aligned}$$

To illustrate the typing rule for the $\mathbf{fix}(x.e)$ construct, we show that

$$\cdot \vdash \mathsf{add} : \mathbf{nat} \to \mathbf{nat} \to \mathbf{nat}.$$

Let $\Gamma = f : \mathbf{nat} \to \mathbf{nat} \to \mathbf{nat}, n : \mathbf{nat}, m : \mathbf{nat}$. Then:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash n : (\mathsf{z} : \mathbf{1}) + (\mathsf{s} : \mathbf{nat})}\,(\text{VAR}) \quad \dfrac{}{\Gamma, \_ : \mathbf{1} \vdash m : \mathbf{nat}}\,(\text{VAR}) \quad \mathcal{D}}{\Gamma \vdash \textbf{case}\; n\; \{\mathsf{z} \cdot \_ \Rightarrow m \mid \mathsf{s} \cdot n' \Rightarrow \mathsf{s} \cdot (fn'm)\}}\,(\text{E-+})}{f : \mathbf{nat} \to \mathbf{nat} \to \mathbf{nat}, n : \mathbf{nat} \vdash \lambda m.\textbf{case}\; n\; \{\cdots\} : \mathbf{nat} \to \mathbf{nat}}\,(\text{LAM})}{f : \mathbf{nat} \to \mathbf{nat} \to \mathbf{nat} \vdash \lambda n.\lambda m.\textbf{case}\; n\; \{\cdots\} : \mathbf{nat} \to \mathbf{nat} \to \mathbf{nat}}\,(\text{LAM})}{\cdot \vdash \mathbf{fix}(f.\lambda n.\lambda m.\textbf{case}\; n\; \{\cdots\}) : \mathbf{nat} \to \mathbf{nat} \to \mathbf{nat}}\,(\text{FIX})$$

where $T_f = \mathbf{nat} \to \mathbf{nat} \to \mathbf{nat}$ and $\mathcal{D}$ is the derivation:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma, n' : \mathbf{nat} \vdash f : T_f}\;(\textsc{Var}) \qquad \dfrac{}{\Gamma, n' : \mathbf{nat} \vdash m : \mathbf{nat}}\;(\textsc{Var})}{\Gamma, n' : \mathbf{nat} \vdash f n' : \mathbf{nat} \to \mathbf{nat}}\;(\textsc{App}) \qquad \dfrac{}{\Gamma, n' : \mathbf{nat} \vdash n' : \mathbf{nat}}\;(\textsc{Var})}{\Gamma, n' : \mathbf{nat} \vdash f n' m : \mathbf{nat}}\;(\textsc{App})}{\Gamma, n' : \mathbf{nat} \vdash \mathsf{s} \cdot (f n' m) : \mathbf{nat}}\;(\text{I-+})$$

We can also define the type of lists of elements of type τ:

$$\tau\ \mathbf{list}\ \text{“=”}\ (\mathsf{nil} : \mathbf{1}) + (\mathsf{cons} : \tau \otimes (\tau\ \mathbf{list})).$$

We can then write an append function, that concatenates two lists:

$$\begin{aligned}\mathsf{append} = \mathbf{fix}(a.\lambda l.\lambda r.\mathbf{case}\ l\ \{&\mathsf{nil} \cdot \_ \Rightarrow r \\ &\mid \mathsf{cons} \cdot p \Rightarrow \mathbf{case}\ p\ \{\langle h, t\rangle \Rightarrow \mathsf{cons} \cdot \langle h, atr\rangle\}\})\end{aligned}$$

In assignment 2, you are asked to explore *lazy products* $\tau \,\&\, \sigma$. It is interesting to reflect on what would have happened had we used lazy products instead of eager products in the definition of τ **list**. That is, what values inhabit the following type:

$$\tau\ \mathbf{mystery}\ \text{“=”}\ (\mathsf{nil} : \mathbf{1}) + (\mathsf{cons} : \tau \,\&\, (\tau\ \mathbf{mystery}))?$$

4 Recursive types

We have so far played fast and loose with our definitions of recursive types. We defined recursive types as solutions to type equations, where the type we were defining appeared on both sides of the equation. It is unclear whether a solution to such an equation exists, let alone if it is unique.

To put recursive types on surer footing, we begin by extending our syntax of types and terms:

$$\begin{array}{lr}\tau ::= \cdots & \\ \quad \mid \rho(\alpha.\tau) & \text{recursive type} \\ e ::= \cdots & \\ \quad \mid \mathbf{fold}(e) & \text{fold } e \text{ into a recursive type} \\ \quad \mid \mathbf{unfold}(e) & \text{unfold } e \text{ out of a recursive type}\end{array}$$

We remark that α may appear bound in τ, i.e., that τ may depend on α. Indeed, the intention is that the bound occurrences of α in τ stand in for any self-reference in τ.

The intention is that we "fold" an expression e of type $[\rho(\alpha.\tau)/\alpha]\tau$ into the recursive type $\rho(\alpha.\tau)$:

$$\frac{\Gamma \vdash e : [\rho(\alpha.\tau)/\alpha]\tau}{\Gamma \vdash \mathbf{fold}(e) : \rho(\alpha.\tau)}\ (\textsc{Fold})$$

Symmetrically, given an expression e of type $\rho(\alpha.\tau)$, we can "unfold" its type to get an expression of type $[\rho(\alpha.\tau)/\alpha]\tau$:

$$\frac{\Gamma \vdash e : \rho(\alpha.\tau)}{\Gamma \vdash \mathbf{unfold}(e) : [\rho(\alpha.\tau)/\alpha]\tau}\ (\textsc{Fold})$$

To illustrate these concepts, we revisit the type **nat**. We define

$$\mathbf{nat} = \rho(\alpha.(\mathsf{z} : \mathbf{1}) + (\mathsf{s} : \alpha)).$$

We then define

$$\begin{aligned} \overline{0} &= \mathbf{fold}(\mathsf{z} \cdot \langle\,\rangle), \\ \overline{n+1} &= \mathbf{fold}(\mathsf{s} \cdot \overline{n}). \end{aligned}$$

These definitions type-check:

$$\cfrac{\cfrac{\cfrac{}{\cdot \vdash \langle\,\rangle : \mathbf{1}}\ (\text{I-}\mathbf{1})}{\cdot \vdash \mathsf{z} \cdot \langle\,\rangle : (\mathsf{z} : \mathbf{1}) + (\mathsf{s} : \rho(\alpha.(\mathsf{z} : \mathbf{1}) + (\mathsf{s} : \alpha)))}\ (\text{I-+})}{\cdot \vdash \mathbf{fold}(\mathsf{z} \cdot \langle\,\rangle) : \rho(\alpha.(\mathsf{z} : \mathbf{1}) + (\mathsf{s} : \alpha))}\ (\textsc{Fold})$$

and

$$\cfrac{\cfrac{\cdot \vdash \overline{n} : \rho(\alpha.(\mathsf{z} : \mathbf{1}) + (\mathsf{s} : \alpha))}{\cdot \vdash \mathsf{s} \cdot \overline{n} : (\mathsf{z} : \mathbf{1}) + (\mathsf{s} : \rho(\alpha.(\mathsf{z} : \mathbf{1}) + (\mathsf{s} : \alpha)))}\ (\text{I-+})}{\cdot \vdash \mathbf{fold}(\mathsf{s} \cdot \overline{n}) : \rho(\alpha.(\mathsf{z} : \mathbf{1}) + (\mathsf{s} : \alpha))}\ (\textsc{Fold})$$

We can recover our examples from Section 3 by introducing $\mathbf{fold}(\cdot)$ and $\mathbf{unfold}(\cdot)$ in the appropriate places:

$$\begin{aligned} \mathsf{pred} &= \lambda n.\mathbf{case}\ (\mathbf{unfold}(n))\ \{\mathsf{z} \cdot \_ \Rightarrow \overline{0} \mid \mathsf{s} \cdot n' \Rightarrow n'\} \\ \mathsf{add} &= \mathbf{fix}(f.\lambda n.\lambda m.\mathbf{case}\ (\mathbf{unfold}(n))\ \{\mathsf{z} \cdot \_ \Rightarrow m \mid \mathsf{s} \cdot p \Rightarrow \mathbf{fold}(\mathsf{s} \cdot (fpm))\}) \\ \mathsf{mult} &= \mathbf{fix}(f.\lambda n.\lambda m.\mathbf{case}\ (\mathbf{unfold}(n))\ \{\mathsf{z} \cdot \_ \Rightarrow m \mid \mathsf{s} \cdot n' \Rightarrow \mathsf{add}(m)(fn'm)\}) \\ \mathsf{fact} &= \mathbf{fix}(f.\lambda n.\mathbf{case}\ (\mathbf{unfold}(n))\ \{\mathsf{z} \cdot \_ \Rightarrow \overline{1} \mid \mathsf{s} \cdot n' \Rightarrow \mathsf{mult}(n)(fn')\}) \end{aligned}$$

We give terms inhabiting recursive types an eager dynamics:

$$\frac{e\ \mathit{val}}{\mathbf{fold}(e)\ \mathit{val}} \qquad \frac{e \mapsto e'}{\mathbf{fold}(e) \mapsto \mathbf{fold}(e')}$$

$$\frac{e \mapsto e'}{\mathbf{unfold}(e) \mapsto \mathbf{unfold}(e')} \qquad \frac{\mathbf{fold}(e)\ \mathit{val}}{\mathbf{unfold}(\mathbf{fold}(e)) \mapsto e}$$

These definitions satisfy the usual progress and preservation properties.

We illustrate our dynamics by considering the following example over the natural numbers, recalling that $\overline{1} \equiv \mathbf{fold}(\mathsf{s} \cdot \overline{0})$:

$$\begin{aligned}
&\mathsf{add}\,\overline{1}\,\overline{2}\\
&\equiv \mathbf{fix}(f.\lambda n.\lambda m.\mathbf{case}\ (\mathbf{unfold}(n))\ \{\mathsf{z}\cdot\_ \Rightarrow m \mid \mathsf{s}\cdot p \Rightarrow \mathbf{fold}(\mathsf{s}\cdot(f p m))\})\overline{1}\,\overline{2}\\
&\mapsto (\lambda n.\lambda m.\mathbf{case}\ (\mathbf{unfold}(n))\ \{\mathsf{z}\cdot\_ \Rightarrow m \mid \mathsf{s}\cdot p \Rightarrow \mathbf{fold}(\mathsf{s}\cdot(\mathsf{add}\,p\,m))\})\overline{1}\,\overline{2}\\
&\mapsto (\lambda m.\mathbf{case}\ (\mathbf{unfold}(\overline{1}))\ \{\mathsf{z}\cdot\_ \Rightarrow m \mid \mathsf{s}\cdot p \Rightarrow \mathbf{fold}(\mathsf{s}\cdot(\mathsf{add}\,p\,m))\})\overline{2}\\
&\mapsto \mathbf{case}\ \mathbf{unfold}(\mathbf{fold}(\mathsf{s}\cdot\overline{0}))\ \{\mathsf{z}\cdot\_ \Rightarrow \overline{2} \mid \mathsf{s}\cdot p \Rightarrow \mathbf{fold}(\mathsf{s}\cdot(\mathsf{add}\,p\,\overline{2}))\}\\
&\mapsto \mathbf{fold}(\mathsf{s}\cdot\mathsf{add}\,\overline{0}\,\overline{2})\\
&\mapsto^* \mathbf{fold}(\mathsf{s}\cdot\overline{2})\\
&\equiv \overline{3}
\end{aligned}$$

References

[Har16] Robert Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, 2nd edition, 2016.