CS 611 Lecture 30: Adding More Types to $\lambda^{\rightarrow}$ November 8, 1999

Richard Samuels

In the last two lectures, we introduced $\lambda^{\rightarrow}$ and showed that it is strongly normalizing in that every expression terminates. Understandably, a strongly normalizing language cannot be as powerful as a language that allows recursion. In this lecture, we will first expand the language by adding sum and product types, and then make it more powerful by introducing the recursive operator **rec** to allow looping.

$\lambda^{\rightarrow\times+}$: Sum and Product Types

Our first task is to expand $\lambda^{\rightarrow}$ to include sum (+) and product (×) types. We call the resulting language $\lambda^{\rightarrow\times+}$. Sum and product types are not codependent—we could have chosen to add only sums without products, resulting in $\lambda^{\rightarrow+}$, or products without sums, yielding $\lambda^{\rightarrow\times}$. Adding both sums and products together, though, gives an interesting symmetry to the language. For example, notice in the syntax definition below that sum types have two constructors and one destructor, while product types have one constructor and two destructors.

Syntax

	Existing $\lambda^{\rightarrow}$ syntax	Products	Sums
e ::=	x | $e_0\ e_1$ | $\lambda\, x{:}\tau.e$ | n | **#u**	| $<e_0, e_1>$ | **left** e | **right** e	| $\mathbf{inl}^{\tau_0+\tau_1}$ e | $\mathbf{inr}^{\tau_0+\tau_1}$ e | **case** $e_0\ e_1\ e_2$
τ ::=	unit | int | $\tau_0 \rightarrow \tau_1$	| $\tau_0 * \tau_1$	| $\tau_0 + \tau_1$

$<e_0, e_1>$ acts as the **cons** operator to create product types, and **left** e and **right** e return the left and right expressions from a product type e.

The $\mathbf{inl}^{\tau_0+\tau_1}$ function injects an expression of type τ_0 into the left side of a sum type $\tau_0 + \tau_1$, and the $\mathbf{inr}^{\tau_0+\tau_1}$ function injects an expression of type τ_1 into the right side. **case** $e_0\ e_1\ e_2$ will invoke the function e_1 on the contents of e_0 if e_0 is an **inl** value, or the function e_2 if e_0 is an **inr** value. **case** should be evaluated lazily—for example, when e_0 = (**inl** v), (**case** $e_0\ e_1\ e_2$) should evaluate to e_1 even if e_2 diverges. In our type semantics, we require that all branches of a **case** statement return the same type.

Operational Semantics

Products:

$$\frac{e_0 \rightarrow e_0'}{<e_0, e_1> \rightarrow <e_0', e_1>} \qquad \frac{e_1 \rightarrow e_1'}{<e_0, e_1> \rightarrow <e_0, e_1'>}$$

$$\frac{e \rightarrow e'}{\mathbf{left}\ e \rightarrow \mathbf{left}\ e'} \qquad \frac{e \rightarrow e'}{\mathbf{right}\ e \rightarrow \mathbf{right}\ e'}$$

$$\frac{}{\mathbf{left}\ <e_0, e_1> \rightarrow e_0} \qquad \frac{}{\mathbf{right}\ <e_0, e_1> \rightarrow e_1}$$

Sums:

$$\frac{e \rightarrow e'}{\mathbf{inl}\ e \rightarrow \mathbf{inl}\ e'} \qquad \frac{e \rightarrow e'}{\mathbf{inr}\ e \rightarrow \mathbf{inr}\ e'} \qquad \frac{e_0 \rightarrow e_0'}{\mathbf{case}\ e_0\ e_1\ e_2 \rightarrow \mathbf{case}\ e_0'\ e_1\ e_2}$$

$$\frac{}{\mathbf{case}\ (\mathbf{inl}\ v)\ e_1\ e_2 \rightarrow e_1\ v} \qquad \frac{}{\mathbf{case}\ (\mathbf{inr}\ v)\ e_1\ e_2 \rightarrow e_2\ v}$$

Static Semantics

Products:

$$\frac{\Gamma \vdash e_0: \tau_0 \qquad \Gamma \vdash e_1: \tau_1}{\Gamma \vdash \langle e_0, e_1\rangle: \tau_0 * \tau_1} \qquad\qquad \frac{\Gamma \vdash e: \tau_0 * \tau_1}{\begin{array}{c}\Gamma \vdash \textbf{left}\ e: \tau_0\\ \Gamma \vdash \textbf{right}\ e: \tau_1\end{array}}$$

Sums:

$$\frac{\Gamma \vdash e: \tau_0}{\Gamma \vdash \textbf{inl}^{\tau_0+\tau_1}\ e: \tau_0+\tau_1} \qquad\qquad \frac{\Gamma \vdash e: \tau_1}{\Gamma \vdash \textbf{inl}^{\tau_0+\tau_1}\ e: \tau_0+\tau_1}$$

$$\frac{\Gamma \vdash e_0: \tau_1+\tau_2 \qquad \Gamma \vdash e_1: \tau_1 \to \tau \qquad \Gamma \vdash e_2: \tau_2 \to \tau}{\Gamma \vdash \textbf{case}\ e_0\ e_1\ e_2: \tau}$$

Typing Denotation

Sums: $\boldsymbol{T}[\![\tau_0 + \tau_1]\!] = \boldsymbol{T}[\![\tau_0]\!]_\perp + \boldsymbol{T}[\![\tau_1]\!]_\perp$
Products: $\boldsymbol{T}[\![\tau_0 * \tau_1]\!] = \boldsymbol{T}[\![\tau_0]\!]_\perp \times \boldsymbol{T}[\![\tau_1]\!]_\perp$

Notice that $*$ represents a product in $\lambda^{\to\times+}$, while $\times$ represents a sum in our metalanguage. Likewise, there is a distinction between the "+" in $\lambda^{\to\times+}$ and the mathematical "+" in our metalanguage, but we don't have enough characters to make this distinction clear.

Sugaring for Boolean Types

Now that we have sum types, we can represent boolean values. If ***D*** is our "desugaring" function:
- $\boldsymbol{D}[\![\text{bool}]\!] = \text{unit} + \text{unit}$
- $\boldsymbol{D}[\![\textbf{true}]\!] = \textbf{inl}^{\text{unit+unit}}\ \text{unit}$; $\boldsymbol{D}[\![\textbf{false}]\!] = \textbf{inr}^{\text{unit+unit}}\ \text{unit}$
- $\boldsymbol{D}[\![\textbf{if}\ e_0\ e_1\ e_2]\!] = (\textbf{case}\ e_0\ (\lambda u{:}\text{unit}.e_1)\ (\lambda u{:}\text{unit}.e_2))$, $u \notin \text{FV}[\![e_1, e_2]\!]$

More Powerful Constructions: λ^{rec}

Though we have gained a lot with the introduction of sums and products, such as the ability to define boolean values in the language, there still haven't been enough changes to make our lambda calculus a usable language. For that, we need to introduce a structure that will allow us to use loops. We have seen such a structure before: it is the **rec** construct.

We have two options available to us for the **rec** construct: one that will allow **rec** to be used universally, to define any recursive structure, and one that will only allow functions to be defined recursively. As before, the former works only in call-by-name languages, and the latter must be used in call-by-value languages:

rec $x{:}\tau.e$	(universal; call-by-name)
rec $f{:}\ \tau_1 \to \tau_2.(\lambda x{:}\tau.e)$	(only for functions; call-by-value)

Recall that the operational semantics for **rec** involves substituting the entire **rec** expression into each reference to the recursive variable:

$$\frac{}{\textbf{rec}\ x{:}\tau\ e \rightarrow e[x \mapsto (\textbf{rec}\ x{:}\tau\ e)]}$$

Static Semantics

The typing rules for λ^{rec} are deceptively simple: if an expression e has type τ, so does (**rec** e):

$$\frac{\Gamma[x\mapsto\tau] \vdash e{:}\ \tau}{\Gamma \vdash (\textbf{rec}\ x{:}\tau\ e){:}\ \tau}$$

Example: FACT

Using λ^{rec}, we can finally write and type-check the FACT function (assuming the existence of our sugaring notation for booleans and the < operator for integers):

FACT $\triangleq$ (**rec** f: int$\rightarrow$int (λx: int.(if (< x 2) 1 ($*$ x (f (- x 1))))))

Since FACT uses **rec** f: int$\rightarrow$int, the type of FACT is int$\rightarrow$int.

Denotational Semantics for λ^{rec}

The **rec** construct adds a lot of power to our language, but it results in one disadvantage: the language is no longer strongly normalizing. Programs can no longer be assumed to terminate, and the size of types does not necessarily decrease as execution progresses. However, recall that we developed λ^{rec} in an attempt to get away from the limitations of strongly normalizing languages!

Note that the **rec** construct in λ^{rec} isn't as powerful as it was in the F_0 language. In particular, we can't create recursive types such as (**rec** x (**cons** 1 x)), as we could in F_0, because such an expression doesn't type-check. To see why, look at the λ^{rec} representation of the construction: (**rec** x: (int$*\tau$) <1,x>). For this expression to work, we need to find a type τ such that int$*\tau = \tau$. We need to introduce recursive types (next lecture) to solve this equation.

Since expressions are no longer guaranteed to terminate, we must modify our denotational semantics to pass around a bottom value representing the result of divergent expressions, as we did in the denotational semantics for IMP.

The general structure for denotations that result in a lifted type is this:

$$\rho \vDash \Gamma \wedge \boldsymbol{C}[\![\Gamma \vdash e{:}\ \tau]\!]\rho \in \boldsymbol{T}[\![\tau]\!]_{\perp}$$

The sum and product types work out as expected. For **rec**, we have to lift the domain of the binding variable as well as the expression, because the expression might not terminate:

$$\boldsymbol{C}[\![\Gamma \vdash (\textbf{rec}\ x{:}\tau\ e){:}\ \tau]\!]\rho = \textbf{fix}\ \lambda\ (v\in \boldsymbol{T}[\![\tau]\!]_{\perp})\ .\ \boldsymbol{C}[\![\Gamma[x\mapsto\tau] \vdash e{:}\ \tau]\!]\ \rho[x\mapsto v]$$

For call-by-value semantics, we cannot use **rec** on normal types. We need to restrict the usage of **rec** to function values, and we need a slightly more complicated expression to support it:

$$\boldsymbol{C}[\![\Gamma \vdash (\textbf{rec}\ f{:}\tau\rightarrow\tau'\ (\lambda x{:}\tau.e)) : \tau\rightarrow\tau']\!]\rho =$$
$$\textbf{fix}\ f'\in(\boldsymbol{T}[\![\tau]\!]\rightarrow\boldsymbol{T}[\![\tau']\!]_{\perp}).\ \lambda x'\in \boldsymbol{T}[\![\tau]\!].\ \boldsymbol{C}[\![\Gamma[f \mapsto (\tau\rightarrow\tau'),\ x \mapsto \tau] \vdash e{:}\ \tau']\!]\ \rho[f \mapsto f',\ x \mapsto x']$$

Even though we can't deal with bottom values under call-by-value semantics, we can manipulate functions that return $\perp$ as long as we don't evaluate them. Since the bottom of the function domain is simply the function that returns $\perp$, we can manipulate functions as lifted c.p.o's within **rec.**

We need to redefine our existing typing denotations for λ^{rec}, so as to make all of the operations strict (i.e. any value that causes a subexpression to diverge should cause the operation to diverge).

Strict Products

At first we may be tempted to write a rule for products like this:

$$\boldsymbol{C}[\![\Gamma \vdash \langle e_0, e_1\rangle : \tau_0 * \tau_1]\!]\rho = \langle \boldsymbol{C}[\![\Gamma \vdash e_0 : \tau_0]\!]\rho, \boldsymbol{C}[\![\Gamma \vdash e_1 : \tau_1]\!]\rho\rangle$$

But this doesn't quite work. Notice what happens when e_0 or e_1 diverges: the pair construction still works. We call this a *lazy* construction. To make our pair construction strict, we can use the **strict** function that we defined earlier (**strict**: $T_\perp \rightarrow (T\rightarrow U) \rightarrow U_\perp$). Recall that **strict** takes a value $v \in T_\perp$ and a function f, and returns $\perp_U$ if $v = \perp_T$, or (f v) if $v \neq \perp_T$.

$$\begin{aligned}\boldsymbol{C}[\![\Gamma \vdash \langle e_0, e_1\rangle : \tau_0 * \tau_1]\!]\rho = \mathbf{strict}(&\boldsymbol{C}[\![\Gamma \vdash e_0 : \tau_0]\!]\rho, \lambda\, v_0 \in \boldsymbol{T}[\![\tau_0]\!].\\ &\mathbf{strict}(\, \boldsymbol{C}[\![\Gamma \vdash e_1 : \tau_1]\!]\rho, \lambda\, v_1 \in \boldsymbol{T}[\![\tau_1]\!].\ \langle v_0, v_1\rangle\ ))\end{aligned}$$

Strict Sums

$$\boldsymbol{C}\,[\![\Gamma \vdash \text{left } e : \tau_0]\!]\rho = \mathbf{strict}(\boldsymbol{C}[\![\Gamma \vdash e : \tau_0 * \tau_1]\!]\rho, \lambda v \in (\boldsymbol{T}[\![\tau_0]\!] \times \boldsymbol{T}[\![\tau_1]\!]).\ \pi_1\ v)$$
$$\boldsymbol{C}\,[\![\Gamma \vdash \text{right } e : \tau_0]\!]\rho = \mathbf{strict}(\boldsymbol{C}[\![\Gamma \vdash e : \tau_0 * \tau_1]\!]\rho, \lambda v \in (\boldsymbol{T}[\![\tau_0]\!] \times \boldsymbol{T}[\![\tau_1]\!]).\ \pi_2\ v)$$

where $(\pi_1\ v)$ and $(\pi_2\ v)$ return the first and second items of the tuple v, respectively.

$$\begin{aligned}&\boldsymbol{C}\,[\![\Gamma \vdash \text{case } e_0\ e_1\ e_2 : \tau]\!]\rho = \mathbf{strict}(\boldsymbol{C}[\![\Gamma \vdash e_0 : \tau_1+\tau_2]\!]\rho, \lambda v \in \boldsymbol{T}[\![\tau_1]\!]+\boldsymbol{T}[\![\tau_2]\!].\\ &\qquad \text{case } v \text{ of}\\ &\qquad\quad in_1(v_1 \in \boldsymbol{T}[\![\tau_1]\!]) \Rightarrow \mathbf{strict}(\boldsymbol{C}[\![\Gamma \vdash e_1 : \tau_1 \rightarrow \tau]\!]\rho, \lambda f \in \boldsymbol{T}[\![\tau_1]\!] \rightarrow \boldsymbol{T}[\![\tau]\!]_\perp.\ f\ v_1)\\ &\qquad\quad |\ in_2(v_2 \in \boldsymbol{T}[\![\tau_2]\!]) \Rightarrow \mathbf{strict}(\boldsymbol{C}[\![\Gamma \vdash e_2 : \tau_2 \rightarrow \tau]\!]\rho, \lambda f \in \boldsymbol{T}[\![\tau_2]\!] \rightarrow \boldsymbol{T}[\![\tau]\!]_\perp.\ f\ v_2))\end{aligned}$$