In the last couple of lectures we introduced the typed lambda calculus language $\lambda^{\rightarrow}$. We have seen that it is strongly normalizing: every expression terminates. Additionally, we have seen that we lost some expressive power by introducing types into our language. For example, we cannot write infinite loops and we don't have recursion. In this lecture we will go further with our $\lambda^{\rightarrow}$ language and first add some new types. The new language will be an extension of $\lambda^{\rightarrow}$ and we will call it tF. After that we will add recursion into our tF language.

1 Syntax

The first thing that we are going to do is adding two new types to $\lambda^{\rightarrow}$ language: sum (+) and product (×) types. The syntax, is an extension of $\lambda^{\rightarrow}$. Suppose n denotes an integer literal, u unit value, x denotes a variable name and e denotes an expression.

$$
\begin{array}{rcl}
e & ::= & x \mid b \mid \mathsf{fn}\ x : \tau\ .e \mid e_1\ e_2 \mid e_1 \oplus e_2 \mid \langle\ e_1,\ e_2\ \rangle \mid \mathsf{first}\ e \mid \mathsf{rest}\ e \\
 & \mid & \mathsf{inl}\ e \mid \mathsf{inr}\ e \mid \mathsf{case}\ e_0\ e_1\ e_2 \\
b & ::= & n \mid \#\mathsf{u}
\end{array}
$$

Basically, we have moved the $\lambda^{\rightarrow}$ language closer to the meta language. Here are the allowed types:

$$
\begin{array}{rcl}
B & ::= & \mathsf{int} \mid \mathsf{unit} \\
\tau & ::= & B \mid \tau_1 \rightarrow \tau_2 \mid \tau_1 * \tau_2 \mid \tau_1 + \tau_2
\end{array}
$$

In these definitions B denotes base types and as it can be seen we have added sum and product types.

2 Structural Operational Semantics

The new tF language is an eager,call-by value language. There is new a expression $\mathsf{case}\ e_0\ e_1\ e_2$ which works as follows: if e_0 is in the form $\mathsf{inl}\ v_0$ then the whole case expression evaluates to e_1 applied to v_0 without evaluating e_2. The same thing is with $\mathsf{inr}\ v_0$ when result is e_2 applied to v_0. In order to present operational semantics let's define what we consider as values.

$$
v \ ::= \ b \mid \mathsf{fn}\ x : \tau.\ e \mid \langle\ v_1,\ v_2\ \rangle \mid \mathsf{inl}\ v \mid \mathsf{inr}\ v
$$

The operational semantics is pretty much the same as we have in uF. So we will just present rules for the case expression.

$$
\frac{}{(case\ (\mathsf{inl}\ v_0)\ e_1\ e_2) \rightarrow e_1\ v_0} \qquad \frac{}{(case\ (\mathsf{inr}\ v_0)\ e_1\ e_2) \rightarrow e_2\ v_0}
$$

We don't have a let expression but we can apply the same desugaring as we did in uF. As it can be seen we didn't define *bool* as base types, but we can emulate boolean as follows:

$$\begin{aligned}
\mathcal{D}[\![\textsf{bool}]\!] &= \mathit{Unit} \oplus \mathit{Unit} \\
\mathcal{D}[\![\#\textsf{t}]\!] &= \textsf{inl}\ (\#\textsf{u}) \\
\mathcal{D}[\![\#\textsf{f}]\!] &= \textsf{inr}\ (\#\textsf{u}) \\
\mathcal{D}[\![\textsf{if}\ e_0\ e_1\ e_2]\!] &= \textsf{case}\ e_0\ (\textsf{fn}\ u : \mathit{unit}.\ e_1)\ (\textsf{fn}\ u : \mathit{unit}.\ e_2)
\end{aligned}$$

3 Static semantics

Now we can present typing rules for tF language

$$\frac{}{\Gamma, x : \tau \vdash x : \tau} \qquad \frac{}{\Gamma \vdash n : \textsf{int}} \qquad \frac{}{\Gamma \vdash u : \textsf{unit}}$$

$$\frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash (\textsf{fn}\ x : \tau.\ e) : \tau \to \tau'} \qquad \frac{\Gamma \vdash e_1 : \tau \to \tau' \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash (e_1\ e_2) : \tau'} \qquad \frac{\Gamma \vdash e_1 : \textsf{int} \quad \Gamma \vdash e_2 : \textsf{int}}{\Gamma \vdash (e_1 + e_2) : \textsf{int}}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \langle e_1\ e_2 \rangle : \tau_1 * \tau_2} \qquad \frac{\Gamma \vdash e : \tau_1 * \tau_2}{\Gamma \vdash (\textsf{first}\ e) : \tau_1} \qquad \frac{\Gamma \vdash e : \tau_1 * \tau_2}{\Gamma \vdash (\textsf{rest}\ e) : \tau_2}$$

$$\frac{\Gamma \vdash e : \tau_1}{\Gamma \vdash (\textsf{inl}_{\tau_1+\tau_2}\ e) : \tau_1 + \tau_2} \qquad \frac{\Gamma \vdash e : \tau_2}{\Gamma \vdash (\textsf{inr}_{\tau_1+\tau_2}\ e) : \tau_1 + \tau_2} \qquad \frac{\Gamma \vdash e_0 : \tau_1 + \tau_2 \quad \Gamma \vdash e_1 : \tau_1 \to \tau_3 \quad \Gamma \vdash e_2 : \tau_2 \to \tau_3}{\Gamma \vdash (\textsf{case}\ e_0\ e_1\ e_2) : \tau_3}$$

As it can be seen we only have binary sums and products. This is not a problem because we simulate multiple arguments. We have $\tau_1 * \tau_2$ and we can start to use these things to build up data types. The following desugaring translates a language with multi-component products/tuples into pairs

$$\begin{aligned}
\mathcal{D}[\![\tau_1 * \ldots\ldots * \tau_n]\!] &= \mathcal{D}[\![\tau_1]\!] * \mathcal{D}[\![\tau_2 * \ldots\ldots * \tau_n]\!] \\
\mathcal{D}[\![\langle e_1, \ldots\ldots, e_n \rangle]\!] &= \langle \mathcal{D}[\![e_1]\!], \mathcal{D}[\![\langle e_2, \ldots\ldots e_n \rangle]\!] \rangle
\end{aligned}$$

We can use a similar desugaring to reduce multi-arm sums into two-arm sums.

4 Recursion

Now we are in a position to actually make tF Turing-equivalent. Right now, it is still strongly normalizing. The type domains and the denotational semantics are as below.

$$\begin{aligned}
\mathcal{T}[\![\tau_1 \to \tau_2]\!] &= \mathcal{T}[\![\tau_1]\!] \to \mathcal{T}[\![\tau_2]\!] \\
\mathcal{T}[\![\tau_1 * \tau_2]\!] &= \mathcal{T}[\![\tau_1]\!] * \mathcal{T}[\![\tau_2]\!] \\
\mathcal{T}[\![\tau_1 + \tau_2]\!] &= \mathcal{T}[\![\tau_1]\!] + \mathcal{T}[\![\tau_2]\!]
\end{aligned}$$

Some examples of the meanings we associate with these terms are as follows:

$$
\begin{aligned}
\mathcal{C}[\![\Gamma \vdash \langle e_1, e_2\rangle : \tau_1 * \tau_2]\!]\rho &= \langle \mathcal{C}[\![\Gamma \vdash e_1 : \tau_1]\!]\rho, \mathcal{C}[\![\Gamma \vdash e_2 : \tau_2]\!]\rho\rangle \\
\mathcal{C}[\![\Gamma \vdash \mathsf{inl}_{\tau_1+\tau_2}\ e : \tau_1 + \tau_2]\!]\rho &= \mathsf{in}_1(\mathcal{C}[\![\Gamma \vdash e : \tau_1]\!]\rho) \in \mathcal{T}[\![\tau_1]\!] + \mathcal{T}[\![\tau_2]\!] \\
\mathcal{C}[\![\Gamma \vdash \mathsf{case}\ e_0\ e_1\ e_2\ : \tau_3]\!]\rho &= \mathsf{case}\ \mathcal{C}[\![\Gamma \vdash e_0 : \tau_1 + \tau_2]\!]\rho\ \mathsf{of} \\
&\quad\ x_1\ .\ (\mathcal{C}[\![\Gamma \vdash e_1 : \tau_1 \to \tau_3]\!]\rho)x_1 \\
&\quad\ \mid x_2\ .\ (\mathcal{C}[\![\Gamma \vdash e_2 : \tau_2 \to \tau_3]\!]\rho)x_2
\end{aligned}
$$

Now we throw recursion into the language in order to be able to write divergent programs! Here is the altered language definition with the modified operational and denotational semantics.

$$e \quad ::= \quad \ldots \ \mid\ \mathsf{rec}\ y : \tau \to \tau'.\ \mathsf{fn}\ x : \tau.\ e$$

$$\mathsf{rec}\ y : \tau \to \tau'.\mathsf{fn}\ x\ e \quad \to \quad \mathsf{fn}\ x : \tau.e\{\mathsf{rec}\ y : \tau \to \tau'.\mathsf{fn}\ x\ e/y\}$$

$$\frac{\Gamma, x : \tau, y : \tau \to \tau' \ \vdash\ e : \tau'}{\Gamma \vdash (\mathsf{rec}\ y.\mathsf{fn}\ x\ e) : \tau \to \tau'}$$

$$
\begin{aligned}
\mathcal{C}[\![\Gamma \vdash (\mathsf{rec}\ y.\mathsf{fn}\ x\ e) : \tau \to \tau']\!]\rho \quad = \quad & \mathsf{fix}\ (\lambda f \in \mathcal{T}[\![\tau \to \tau']\!]. \\
& \lambda v \in \mathcal{T}[\![\tau]\!].\ \mathcal{C}[\![\Gamma, x : \tau, y : \tau \to \tau' \vdash e : \tau']\!]\rho[x \mapsto v, y \mapsto f])
\end{aligned}
$$

Notice that we are taking fixed points now, which requires that the domain $\mathcal{T}[\![\tau \to \tau']\!]$ is a pointed cpo. Thus, we need $\bot$ and we finally have divergent programs in our language.

$$
\begin{aligned}
\mathcal{T}[\![\tau_1 \to \tau_2]\!] &= \mathcal{T}[\![\tau_1]\!] \to \mathcal{T}[\![\tau_2]\!]_\bot \\
\rho \models \Gamma &\Rightarrow \mathcal{C}[\![\Gamma \vdash e : \tau]\!]\rho \in \mathcal{T}[\![\tau]\!]_\bot
\end{aligned}
$$

Note that in a CBN language, we would have $\mathcal{T}[\![\tau_1 \to \tau_2]\!] = \mathcal{T}[\![\tau_1]\!] \to \mathcal{T}[\![\tau_2]\!]$ and $\mathcal{T}[\![int]\!] = Z_\bot$. Thus, all types would be modeled by pointed domains.

5 A limitation

tF does not have recursive type definitions, which means we still cannot define reasonable data structures. This will be addressed in subsequent lectures.