1 Semantics of REC (conclusion)

The language uF presented in class is a call-by-value, left-to-right, eager, untyped functional language.

1.1 Definition

Recall our definition of the language REC, where e represents an expression, and d represents all the function definitions:

$$\begin{array}{rcl} \mathit{prog} & ::= & \mathsf{let}\ d\ \mathsf{in}\ e \\ d & ::= & f_1(x_1,\ldots,x_{a_1}),\cdots,f_n(x_1,\ldots,x_{a_n}) \\ e & ::= & n \mid x \mid e_0 \oplus e_1 \mid \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \mid f_i(e_1,\cdots,e_{a_i}) \mid \mathsf{ifz}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \end{array}$$

1.2 Semantics

We didn't explore a semantics of the language in which all functions could refer to each other (with either lazy or eager evaluation).

1.2.1 Eager Semantics

Recall the types of our translations.

$$\begin{array}{rcl} \mathcal{C}[\![e]\!] & \in & \mathit{FEnv} \to \mathit{Env} \to \mathbb{Z}_\perp \\ \phi & \in & \mathit{FEnv} = (\mathbb{Z}^{a_1} \to \mathbb{Z}_\perp) \times \cdots \times (\mathbb{Z}^{a_n} \to \mathbb{Z}_\perp) \\ \mathit{Env} & = & \mathit{Var} \to \mathbb{Z} \end{array}$$

(Recall that this is slightly different for lazy semantics: the domain of each function must be $\mathbb{Z}_\perp$ instead of $\mathbb{Z}$ so that it can accept nonterminating computations).

The semantics for the program $\mathsf{let}\ d\ \mathsf{in}\ e$ will be some translation of the form $\mathcal{C}[\![e]\!]\mathcal{D}[\![d]\!]\phi$.

Consider the ϕ that we want. It will have the form $\phi = \langle F_1, ..., F_n\rangle$. If we already had ϕ, then we could define

$$F_i = \lambda v_1 \in \mathbb{Z}, \ldots, v_{a_i} \in \mathbb{Z}.\mathcal{C}[\![e]\!]\phi[x_1 \mapsto v_1, \ldots, x_{a_i} \mapsto v_{a_i}].$$

This expression is closed except for ϕ. Think of F_i as a function of ϕ. Then, $\mathcal{D}[\![d]\!] = \phi = \langle F_1(\phi), \ldots\rangle$

Then we need to take a fixed point:

$$\begin{array}{rcl} \mathcal{F}(\phi) & = & \lambda\phi \in \mathit{FEnv}.\langle F_1(\phi), \ldots, F_n(\phi)\rangle \\ \mathcal{D}[\![d]\!] & = & \mathit{fix}\ \mathcal{F} \end{array}$$

Is this well-defined? $\mathcal{F}$ needs to be continuous — it is, because we constructed it with our metalanguage. In addition, the domain needs to be pointed (the domain here is *FEnv*).

FEnv is a product domain, and a product domain is pointed if each element of the tuple is pointed. Each element here is a function domain, and a function domain is pointed if its codomain is pointed. This is true here because the codomain of each function is $\mathbb{Z}_\perp$.

The $\perp$ of *FEnv* is this:

$$\perp_\phi = \langle \lambda v_1 \in \mathbb{Z}, \ldots \lambda v_{a_1} \in \mathbb{Z}.\perp, \lambda v_1 \in \mathbb{Z}, \ldots \lambda v_{a_2} \in \mathbb{Z}.\perp, \ldots\rangle$$

1.2.2 Aside on fixed points

Consider drawing a graph of the fixed point equation. One could draw an infinite graph, but from the point of view of someone traversing the graph, this would be equivalent to a graph with a cycle. This is how it would be implemented, and actually all fixed points are in some sense about cycles in graphs.

Fixed points in compilers typically correspond to a placeholder where you have to go back and stick in something when you figure it out.

1.2.3 Lazy semantics

Recall in the previous REC semantics we just had to adjust the functions so that their domains are lifted ($\mathbb{Z}_\perp^{a_i}$). We do the same here.

2 Denotational Semantics of uF

Now we want to use what we have learned to write denotational semantics for a language with higher-order functions — in particular, uF.

2.1 Definition

$$
\begin{array}{rcl}
b & ::= & \#\mathsf{u} \mid \#\mathsf{t} \mid \#\mathsf{f} \mid n \mid s \\
e & ::= & b \mid x \mid e_0 \oplus e_1 \mid \mathsf{if}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \mid \langle e_1, e_2 \rangle \mid \mathsf{left}\ e \mid \mathsf{right}\ e \mid \lambda x\ e \mid e_0\ e_1 \\
& & \mid\ \mathsf{rec}\ y\ (\lambda x\ e) \mid \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2
\end{array}
$$

2.2 Domains

uF is more complex than REC. A value is much more complicated thing, not just a number, but also now a pair, function, boolean value, or a string. So the domain of values is a sum domain of all these possibilities:

$$Value = \mathbb{U} + \mathbb{B} + \mathbb{Z} + String + Pair + Function$$

Our translation of expressions takes the form $\mathcal{C}[\![e]\!]\rho$, where $\rho \in Env = Var \rightharpoonup Value$. It is a partial function that maps free variables of e to their values. We will only ask for the meaninings of expressions e in environments ρ where $\mathcal{FV}[\![e]\!] \in dom(\rho)$.

What is the domain of $\mathcal{C}[\![e]\!]\rho$?

$$Computation = (Value + Error)_\perp$$

How do we define *Error*? There are many ways, but the simplest is

$$Error = \mathbb{U}$$

How do we distinguish the *Value* unit rather than the *Error* unit? We will know because of the the injection functions:

$$
\begin{array}{rcl}
\mathcal{C}[\![\#u]\!]\rho & = & \lfloor in_1(in_1(u)) \rfloor \\
\mathsf{error} & = & \lfloor in_2(u) \rfloor
\end{array}
$$

We define the remaining domains as follows:

$$
\begin{array}{rcll}
String & = & \mathbb{Z} & \text{(since the integers are rich enough to represent all strings.)} \\
Pair & = & Value \times Value & \\
Function & = & Value \to Computation &
\end{array}
$$

2.3 Domain Equations

There is something very fishy here. These definitions are circular. That is because these aren't definitions – they are **equations** that we would like our domains to satisfy. We have not seen yet how to find solutions to domain equations. We will not talk right now about finding these solutions, but rather assert that a solution is findable.

2.4 Translations

Given that we have domains that solve these equations, we can write down semantics. The translations for unit and error are given above.

Some translations are easy:

$$\begin{array}{rcll}
\mathcal{C}[\![\#\mathsf{t}]\!] & = & \lfloor in_1(in_2(\mathsf{true}))\rfloor & \\
\mathcal{C}[\![\#\mathsf{f}]\!] & = & \lfloor in_1(in_2(\mathsf{false}))\rfloor & \\
\mathcal{C}[\![n]\!] & = & \lfloor in_1(in_3(n))\rfloor & \\
\mathcal{C}[\![s]\!] & = & \lfloor in_1(in_4(s'))\rfloor & \text{where } s' \text{ is our integer representation of the string } s. \\
\mathcal{C}[\![x]\!]\rho & = & \lfloor \rho x\rfloor &
\end{array}$$

For translation of something that actually does computation, it gets a little more complicated.

$\mathcal{C}[\![e_0 \oplus e_1]\!]\rho =$
$\mathsf{let}\ x_1 = \mathcal{C}[\![e_1]\!]\rho.$
$\mathsf{case}\ x_1\ \mathsf{of}$
$in_1(v).\mathsf{case}\ v\ \mathsf{of}$
$\mid\ in_1(u).\mathsf{error}$
$\mid\ in_2(b).\mathsf{error}$
$\mid\ in_3(n_1).\mathsf{let}\ x_2 = \mathcal{C}[\![e_2]\!]\rho\ \mathsf{in\ case}\ x_2\ \mathsf{of}$
$in_1(u).\mathsf{error}\ \mid$
$in_2(b).\mathsf{error}\ \mid$
$in_3(v_2).\lfloor(in_3(v_1 \oplus v_2)\rfloor$
$\mid\ in_2(e).\mathsf{error}$

... and there are even more cases.

2.4.1 ecase

How do we avoid this nastiness? We can do continuation-passing semantics and use error-checkers that terminate computation if we have an error. But that is too hard and distracting just now. So we will introduce a new metalanguage statement ecase

ecase e of
(pattern-matching expression) . *value*

- strict in e as long as domain is pointed
- any missing cases go to the error result
- we can do multi levels of cases at once — deep matching, as in ML

2.4.2 if-then-else

$\mathcal{C}[\![\mathsf{if}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2]\!]\rho =$
$\mathsf{ecase}\ \mathcal{C}[\![e_0]\!]\rho\ \mathsf{of}$
$\lfloor in_1(in_2(\mathsf{true}))\rfloor.\mathcal{C}[\![e_1]\!]\rho$
$\mid\ \lfloor in_1(in_2(\mathsf{false}))\rfloor.\mathcal{C}[\![e_2]\!]\rho$
end

And all other cases become errors, as ecase is 'compiled' down into case.

2.4.3 Pairs, Left, and Right

$$\begin{aligned}
\mathcal{C}[\![\langle e_1, e_2\rangle]\!]\rho &= \mathsf{ecase}\ \mathcal{C}[\![e_1]\!]\rho\ \mathsf{of}\ in_1(v_1).\ \mathsf{ecase}\ \mathcal{C}[\![e_2]\!]\rho\ \mathsf{of}\ in_1(v_2).\lfloor in_1(in_5(\langle v_1, v_2\rangle))\rfloor \\
\mathcal{C}[\![\mathsf{left}\ e]\!]\rho &= \mathsf{ecase}\ \mathcal{C}[\![e]\!]\rho\ \mathsf{of}\ in_1(in_5(p)).\lfloor in_1(\pi_1 p)\rfloor \\
\mathcal{C}[\![\mathsf{right}\ e]\!]\rho &= \mathsf{ecase}\ \mathcal{C}[\![e]\!]\rho\ \mathsf{of}\ in_1(in_5(p)).\lfloor in_1(\pi_2 p)\rfloor
\end{aligned}$$

2.4.4 Functions

$$\mathcal{C}[\![\lambda x\ e]\!]\rho = \lfloor in_1(in_6(\lambda v \in \mathit{Value}.\mathcal{C}[\![e]\!]\rho[x \mapsto in_1(v)]))\rfloor$$

2.4.5 Applications

$$\mathcal{C}[\![e_0\ e_1]\!]\rho = \mathsf{ecase}\ \mathcal{C}[\![e_0]\!]\rho\ \mathsf{of}\ in_1(in_6(f)).\mathsf{ecase}\ \mathcal{C}[\![e_1]\!]\rho\ \mathsf{of}\ in_1(v).f\ v$$

Note that one of the things ecase being strict in e means is that we do not have to write lift around the cases.

2.4.6 Recursive Functions

$$\mathcal{C}[\![\mathsf{rec}\ y(\lambda x\ e)]\!] = \lfloor in_1(in_6(\mathit{fix}\ \lambda f \in \mathit{Function}.\lambda v \in \mathit{Value}.\mathcal{C}[\![e]\!]\rho[x \mapsto in_1(v), y \mapsto in_1(in_6(f))].))\rfloor$$

Are we allowed to take this fixed point? The function is continuous, but is the cpo pointed? *Function* is pointed if its codomain *Computation* is pointed — which it is, so this is OK.

2.4.7 Notes on error-checking direct semantics for uF

It looks a lot like the call-by-value λ-calculus translation, but also includes the injection functions. One nice thing here is that the domain equations help keep you honest. Once you have set up the domain equations, the rest is sort of forced on you because you have to write something that 'typechecks.'

2.5 Nontermination

The fact that this language does not always terminate showed up in a couple of places.

For example, recursive functions that call themselves. (rec $y(\lambda x\ y\ x)$). We had to take a fixed point to define rec, introducing a bottom element $\perp$.

Actually, of course we can write nonterminating programs without the rec construct. We did not take a fixed point to define function abstraction and application. So did we need pointed domains? Yes, because we need to solve the domain equations. To solve domain equations, in general uses of the $\rightarrow$ constructor must have a pointed codomain. Hence, *Function* must be pointed.