1 Data structures

We showed that we could encode booleans and arithmetic using lambda calculus expressions. This is a good place to start, but we still lack any sort of data structure. Consider, for example, ordered pairs. It would be nice to have functions *PAIR*, *LEFT*, and *RIGHT* which obeyed the following equational specifications:

$$\begin{aligned} \mathit{LEFT}(\mathit{PAIR}(e_1\ e_2)) &= e_1 \\ \mathit{RIGHT}(\mathit{PAIR}(e_1\ e_2)) &= e_2 \\ \mathit{PAIR}\ (\mathit{LEFT}\ p)\ (\mathit{RIGHT}\ p) &= p \end{aligned}$$

We can begin with *PAIR*, trying to wrap two given values for later use as the arguments to a yet-unused third value, similar to the manner *IF* wrapped its two branch options for extraction by the appropriate boolean:

$$\mathit{PAIR} \triangleq \lambda a\ b.\, \lambda f.\, f a b$$

To get the first element given to a *PAIR*, we need some function which will take two arguments and return the first. Conveniently, that is exactly what *TRUE* does, as we defined it (note: other encodings of booleans are not guaranteed to work!). Similarly, $\mathit{PAIR}\ a\ b\ \mathit{FALSE} = b$. Thus, we can define

$$\begin{aligned} \mathit{LEFT} &\triangleq \lambda p.\, p\ \mathit{TRUE} \\ \mathit{RIGHT} &\triangleq \lambda p.\, p\ \mathit{FALSE} \end{aligned}$$

Again, if p isn't a term of the form $\mathit{PAIR}\ a\ b$, expect the unexpected.

2 Recursion and the Y-combinator

With an encoding for *IF*, we have some control over the flow of a program, but we do not yet have the ability to write a loop, such as a factorial *FACT* function. In OCaml, a variant of ML, we can write

let rec fact(n)=if n<2 then 1 else n*fact(n-1)

But how can this be pulled off in the λ-calculus, where all the functions are anonymous? As a first guess, we might start with the equational specification

$$\mathit{FACT} = \lambda n.\, \mathit{IF}(<\ n\ 2)\ 1\ (*\ n\ (\Big(\mathit{FACT}\Big)(-\ n\ 1)))$$

Alas, *FACT* is just shorthand for the stuff on the RHS, and until we know what needs to be written down for *FACT*, we can't write down *FACT*. Now if only there were a way to remove the recursive call...

2.1 The recursion removal trick

Suppose we break up the recursion into two steps. First, make a function *FACT'* which says "If I was given my own name, I could do what *FACT* should". Thus, if $f = \mathit{FACT'}$, then we can treat $\mathit{FACT'}(f)$ as if it were just *FACT*:

$$\mathit{FACT'} \triangleq \lambda f.\, \lambda n.\, \mathit{IF}(<\ n\ 2)\ 1\ (*\ n\ (\Big(f\ f\Big)(-\ n\ 1)))$$

And, since *FACT' FACT'* should behave as we want *FACT* to,

$$\mathit{FACT} \triangleq \mathit{FACT'}\ \mathit{FACT'}$$

We can now see the recursion working:

$$\begin{aligned}
\mathit{FACT}(4) &= (\mathit{FACT'}\ \mathit{FACT'})\ 4 \\
&= \lambda n.\, \mathit{IF}(<n\ 2)\ 1\ (*\ n\ (\Big(\mathit{FACT'}\ \mathit{FACT'}\Big)(-\ n\ 1)))\ (4) \\
&= \mathit{IF}(<4\ 2)\ 1\ (*\ 4\ (\Big(\mathit{FACT}\Big)(-\ 4\ 1))) \\
4 * FACT(3) &= (*\ 4\ (\mathit{FACT}(-\ 4\ 1)))
\end{aligned}$$

2.2 Fixed points and the Y-Combinator

For reasons which will soon become apparent, it might be useful to look at the problem of finding *FACT* as a problem of finding a fixed point. Note that for the following function, $\mathit{FACT} = F(\mathit{FACT}) = (n\ F)(\mathit{FACT})$:

$$F = \lambda f.\, \lambda n.\, \mathit{IF}(<n\ 2)\ 1\ (*\ n\ (\Big(f\Big)(-\ n\ 1)))$$

This comes essentially from wrapping the RHS of the equational specification for *FACT* in $\lambda \mathit{FACT}.\, RHS$, and could be applied to other recursive functions: that is, every recursive function R can be viewed as the fixed point of an associated function R'. So, if we had a function Y which returns a fixed point of a given input, we could find recursive functions R by applying Y to their associated R'.

Equationally, we can specify such a Y by

$$YF = F(YF)$$

This would give us

$$Y = \lambda F.\, \lambda x.\, F(YF)x$$

While this is, itself, recursive, we can apply the recursion removal trick to get

$$\begin{aligned}
Y &\triangleq Y'\ Y' \\
Y' &\triangleq \lambda y.\, \lambda F.\, \lambda x.\, (F((y\ y)F))x
\end{aligned}$$

This Y is the famous "Y-combinator": A closed λ-term (combinator) which finds solutions to recursion situations. It is more commonly written

$$Y \triangleq \lambda f.\, ((\lambda x.\, fxx)(\lambda x.\, fxx))$$

Regardless of how it is written, it can be used to define recursive functions without stepping through particular cases of recursion removal, such as

$$\mathit{FACT} \triangleq Y\ \lambda f.\, (\lambda n.\, \mathit{IF}(<n\ 2)\ 1\ (*\ n\ (f(-\ n\ 1))))$$

3 Call-by-name and call-by-value

In the pure λ-calculus, we can start with a term and perform reductions on subterms in any order. However, for modeling programming languages, it is useful to restrict which β reductions are allowed and in what order they can be performed.

In general there may be many possible β-reductions that can be performed on a given λ-term. How do we choose which beta reductions to perform next? Does it matter?

A specification of which reduction to perform next is called a *reduction strategy*. Let us define a *value* to be a closed λ-term to which no β-reductions are possible, given our chosen reduction strategy. For example, $\lambda x.x$ would always be a value, whereas $(\lambda x.x)\ 1$ would most likely not be.

Most real programming languages based on the λ-calculus use a reduction strategy known as *Call By Value* (CBV). In other words, functions may only be applied to (called on) values. Thus $(\lambda x.e)\ e'$ only reduces if e' is a value v. Here is an example of a CBV evaluation sequence, assuming 3, 4, and S (the successor function) are appropriately defined.

$$((\lambda x.\lambda y.y\ x)\ 3)\ S \quad\longrightarrow\quad (\lambda y.y\ 3)\ S \quad\longrightarrow\quad S\,3 \quad\longrightarrow\quad 4.$$

Another strategy is *call by name* (CBN). We defer evaluation of arguments until as late as possible, applying reductions from left to right within the expression. In other words, we can pass an incomplete computation to a function as an argument. Terms are evaluated only once their value is really needed.

What happens if we try using Ω as a parameter? It differs when we use CBV vs. CBN.

Consider this example:

$$e = (\lambda x.(\lambda y.y))\ \Omega$$

Using the CBV evaluation strategy, we must first reduce Ω. This puts the evaluator into an infinite loop, so $e \Uparrow_{CBV}$. On the other hand, CBN reduces the term above to $\lambda y.\, y$. CBN has an important property: CBN will not loop infinitely unless every other semantics would also loop infinitely, yet it agrees with CBV whenever CBV terminates successfully:

$$e \Downarrow_{CBV} \implies e \Downarrow_{CBN}$$

4 Structural Operational Semantics (SOS)

Let's formalize CBV. First, we need to define the values of the language. These are simply the lambda terms:

$$v ::= \lambda x.\, e$$

Next, we can write inference rules to define when reductions are allowed:

$$\frac{}{(\lambda x.\, e)\ v \longrightarrow e\{v/x\}}\quad[\beta\text{-reduction}]$$

$$\frac{e_1 \longrightarrow e_1'}{e_1\ e_2 \longrightarrow e_1'\ e_2}$$

$$\frac{e \longrightarrow e'}{v\ e \longrightarrow v\ e'}$$

This is an example of an operational semantics for a programming language based on the lambda calculus. An operational semantics is a language semantics that describes how to run the program. This can be done through informal human-language text, as in the Java Language Specification, or through more formal rules. Rules of this form are known as a Structural Operational Semantics (SOS). They define evaluation as the result of applying the rules to transform the expression, and the rules are defined in term of the structure of the expression being evaluated.

This kind of operational semantics is known as a *small-step* semantics because it only describes one step at a time, by defining a transition relation $e \longrightarrow e'$. A program execution consists of a sequence of these small steps strung together. An alternative form of operational semantics is a *big-step* (or *large-step*) semantics that describes the entire evaluation of the program to a final value.

As defined above, CBV evaluation is *deterministic*: there is only one evaluation leading from any given term. (We leave proving this for later). If we allow evaluation to work on the right-hand side of an application, evaluation will be nondeterministic:

$$\frac{e_2 \longrightarrow e_2'}{e_1\ e_2 \longrightarrow e_1\ e_2'}$$

We will see other kinds of semantics later in the course, such as *axiomatic semantics*, which describes the behavior of a program in terms of the observable properties of the input and output states, and *denotational semantics*, which translates a program into an underlying mathematical representation.

Expressed as SOS, CBN has slightly simpler rules:

$$\frac{}{(\lambda x.\, e_1)\ e_2 \longrightarrow e_1\{e_2/x\}} \quad [\beta\text{-reduction}]$$

$$\frac{e_0 \longrightarrow e_0'}{e_0\ e_1 \longrightarrow e_0'\ e_1}$$

We don't need the rule for evaluating the right-hand side of an application because β-reductions are done immediately once the left-hand side is a value.

5 Term Equivalence

When are two terms equal? This is not as simple a question as it may seem. As *intensional* objects, two terms are equal if they are syntactically identical. As *extensional* objects, however, two terms should be equal if they represent the same function. We will say that two terms are *equivalent* if they are equal in an extensional sense.

For example, it seems clear that the terms $\lambda x.\, x$ and $\lambda y.\, y$ are equivalent. The name of the variable is not essential. But we also probably think that $\lambda x.\, (\lambda y.\, y)\ x$ is equivalent to $\lambda x.\, x$ too, in a less trivial sense. And there are even more interesting cases, like $\lambda x.\, \lambda y.\, x\ y$.

But what function does a term like $\lambda x.\, x$ represent? Intuitively, it's the identity function, but over what domain and codomain? We might think of it as representing the set of all identity functions, but this interpretation quickly leads to Russell's paradox. In fact, defining a precise mathematical model for lambda-calculus terms is far from straightforward, requiring some sophisticated domain theory.

One possible meaning of a term is divergence. There are infinitely many divergent terms; one example is Ω. In some sense, all divergent terms are equivalent, since none of them produce a value. The implication is that it is undecidable to determine whether two terms are equivalent, because otherwise, given the relationship between the λ-calculus and Turing machines, we could solve the halting problem on lambda calculus terms by testing equivalence to Ω.

5.1 Observational equivalence

Another way of approaching the problem is to say that two terms are equivalent if they behave indistinguishably in all possible contexts.

More precisely, two terms will be considered equal if in every context, either

- they both converge and produce the same value, or
- they both diverge.

A *context* is just a term $C[[\,\cdot\,]]$ with a single occurrence of a distinguished special variable, called the *hole*. The notation $C[e]$ denotes the context $C[[\,\cdot\,]]$ with the hole replaced by the term e. Then we then define equality in the following way:

$$e_1 = e_2 \quad \Longleftrightarrow \quad \text{for all contexts } C[[\,\cdot\,]],\ C[e_1] \Downarrow v \text{ iff } C[e_2] \Downarrow v.$$

Without loss of generality, we can simplify the definition to

$$e_1 = e_2 \quad \Longleftrightarrow \quad \text{for all contexts } C[[\,\cdot\,]],\ C[e_1] \Downarrow \text{ iff } C[e_2] \Downarrow,$$

because if they converge to different values, it is possible to devise a context that causes one to converge and the other to diverge. Suppose that $C[e_1] \Downarrow v_1$ and $C[e_2] \Downarrow v_2$, where v_1 and v_2 have different behavior. Then we can find some context $C'[[\,\cdot\,]]$, which applied to v_1 converges, and applied to v_2 diverges. Therefore, the context $C'[C[\cdot]]$ is a context that causes the original e_1, e_2 to converge and diverge respectively, satisfying the simpler definition.

A conservative approximation (but unfortunately still undecidable) is the following. Let e_1 and e_2 be terms, and suppose that e_1 and e_2 converge to the same value when reductions are applied according to some strategy. Then e_1 is equivalent to e_2. This *normalization* approach (in which terms are reduced to a a *normal form* on which no more reductions can be done) is useful for compiler optimization and for checking type equality in some advanced type systems.