CS611 Lecture 25 Standard (CPS) semantics and non-hierarchical scope 29 Oct 01

Scribe: Andrew Myers Lecturer: Andrew Myers

1 CPS semantics

Just as we can translate code into continuation-passing style, we can similarly use CPS to define the denotational semantics of programs. This style of denotational semantics is known as *continuation semantics* or *standard semantics*. It has the usual advantages of CPS: the resulting program description is low-level, closer to what a compiler might generate. It is also more compact and expressive than the direct semantics; it is better at describing non-local flows of control. To demonstrate this semantic style, we will write a standard semantics for uF, whose syntax is repeated here:

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

1.1 Domain equations

As usual, the first step in getting denotational semantics right is to write down the domain equations correctly. The rest then follows almost inevitably. We will recall that in CPS style, a continuation looks like a function that takes in the result of previous computation and causes the rest of the computation to occur. The result of applying a continuation is never used. We can capture this by giving continuations an arrow domain in which the codomain has no specified structure (and hence its values cannot be used). Let *Cont* be the domain of continuations:

$$\mathit{Cont} = \mathit{Value} \to \mathit{Answer}$$

We do not actually care about the domain of *Answer*, although we will see that it must be pointed. We obtain an *Answer* by using the special function $\mathit{halt} \in \mathit{Computation} \to \mathit{Answer}$, where as before, $\mathit{Computation} = (\mathit{Value} + \mathit{Error})_\perp$. A natural choice for the domain *Answer*, particularly if we want to establish a connection between the direct and the standard semantics, is $\mathit{Answer} = \mathit{Computation}$. In this case, we can define $\mathit{halt} = \lambda c \in \mathit{Computation}\ .\ c$. Another choiceis to make *Answer* the smallest pointed domain, $0_\perp$ (which is really $\mathcal{U}$). In this case we define $\mathit{halt} = \lambda c \in \mathit{Computation}\ .\ \perp$. The point is that no matter what domain we choose, it will have no impact on the standard semantics.

The CPS meaning function $\mathcal{C}[\![e]\!]$ gives the meaning of an computation that starts from e, given an environment to interpret its identifiers in and a continuation that captures the rest of the computation. That is, $\mathcal{C}[\![e]\!]\rho k$ results in the application of k to the result of e, so the domain of the meaning function is given by:

$$\mathcal{C}[\![e]\!] \in \mathit{Env} \to \mathit{Cont} \to \mathit{Answer}$$

$$\mathcal{C}[\![e]\!]\rho k \in \mathit{Answer}$$

The environment ρ is the naming context for e; the continuation k is the control context for e.

Given the direct semantics meaning function $\mathcal{C}_\mathrm{D}[\![\cdot]\!]$, we expect the following correspondence between the two semantics regardless of the choice of *Answer*:

$$\mathit{halt}(\mathcal{C}_\mathrm{D}[\![e]\!]\rho) = \mathcal{C}[\![e]\!]\rho(\lambda v \in \mathit{Value}\ .\ \mathit{halt}(\lfloor \mathit{in}_1(v) \rfloor))$$

Continuing with our uF semantics, we can repeat some of the domain equations from the previous lecture:

$$\begin{aligned} \mathit{Value} &= \mathbb{U} + \mathbb{B} + \mathbb{Z} + \mathit{Pair} + \mathit{Function} \\ \mathit{Env} &= \mathit{Var} \rightharpoonup \mathit{Value} \\ \mathit{Error} &= \mathbb{U} \\ \mathit{Pair} &= \mathit{Value} \times \mathit{Value} \end{aligned}$$

Note that we can define environments as partial functions from names *Var* to values. We can straightforwardly show by structural induction that these partial functions are applied only where they are defined, assuming that we apply the meaning function to expressions e and environments ρ where ρ is defined on all free variables of e. (This won't work for dynamic scope!)

While all other values have roughly the same structure as in the direct semantics, we will now model functions as functions with two arguments. The first argument is the ordinary argument value; the second is the continuation where the function result is sent. Thus, the domain equation for functions is

$$\mathit{Function} = \mathit{Value} \to \mathit{Cont} \to \mathit{Answer}$$

1.2 Semantic function $\mathcal{C}[\![\cdot]\!]$

We define the semantic function $\mathcal{C}[\![\cdot]\!]$ by induction on the structure of its argument, as usual. Variables and simple constants are straightforward:

$$\begin{aligned} \mathcal{C}[\![\#\mathsf{u}]\!]\rho k &= k\ \mathit{in}_1(u) \\ \mathcal{C}[\![\#\mathsf{t}]\!]\rho k &= k\ \mathit{in}_2(\mathit{true}) \\ \mathcal{C}[\![\#\mathsf{f}]\!]\rho k &= k\ \mathit{in}_2(\mathit{false}) \\ \mathcal{C}[\![n]\!]\rho k &= k\ \mathit{in}_3(n) \\ \mathcal{C}[\![x]\!]\rho k &= k(\rho x) \end{aligned}$$

To describe more interesting computation, we will need to add some run-time error checking. This can be done more conveniently than in the direct semantics:

$$\mathcal{C}[\![\mathsf{if}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2]\!]\rho k = \mathcal{C}[\![e_0]\!]\rho k(\mathit{check\text{-}bool}(\lambda b \in \mathbb{B} \,.\, \mathit{if}\ b\ \mathit{then}\ \mathcal{C}[\![e_1]\!]\rho k\ \mathit{else}\ \mathcal{C}[\![e_2]\!]\rho k))$$

The checking needed to ensure that the term e_0 evaluates to a boolean is encapsulated in a helper function *check-bool* $\in (\mathbb{B} \to \mathit{Answer}) \to \mathit{Cont}$, which we can define in a manner similar to that in lecture 15. This function expands the domain of a continuation that is only able to accept booleans so that it can accept all values. Non-booleans cause immediate termination with an error.

$$\begin{aligned} \mathit{check\text{-}bool} &\overset{def}{=} \lambda k \in \mathbb{B} \to \mathit{Answer} \,. \\ &\quad \lambda v \in \mathit{Value} \,.\, \mathit{case}\ v\ \mathit{of} \\ &\qquad \mathit{in}_1(u)\ .\ \mathit{halt\ error} \\ &\qquad |\ \mathit{in}_2(b)\ .\ k(b) \\ &\qquad |\ \mathit{in}_3(n)\ .\ \mathit{halt\ error} \\ &\qquad |\ \ldots \\ &\quad \mathit{end} \\ \mathit{error} &= \lfloor \mathit{in}_2 u \rfloor \end{aligned}$$

We can similarly define other such functions *check-X* that are elements of $(X \to \mathit{Answer}) \to \mathit{Cont}$.

The rest of the semantics then closely follows the translations defined in lecture 15:

$$\begin{aligned}
\mathcal{C}[\![\langle e_1, e_2\rangle]\!]\rho k &= \mathcal{C}[\![e_1]\!]\rho(\lambda v_1 \in \mathit{Value}\,.\,\mathcal{C}[\![e_2]\!]\rho(\lambda v_2 \in \mathit{Value}\,.\,k\ \mathit{in}_4\langle v_1, v_2\rangle))\\
\mathcal{C}[\![\mathsf{left}\ e]\!]\rho k &= \mathcal{C}[\![e]\!]\rho(\mathit{check\text{-}pair}(\lambda p \in \mathit{Pair}\,.\,k(\pi_1(p))))\\
\mathcal{C}[\![\mathsf{right}\ e]\!]\rho k &= \mathcal{C}[\![e]\!]\rho(\mathit{check\text{-}pair}(\lambda p \in \mathit{Pair}\,.\,k(\pi_2(p))))\\
\mathcal{C}[\![\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2]\!]\rho k &= \mathcal{C}[\![e_1]\!]\rho(\lambda v \in \mathit{Value}\,.\,\mathcal{C}[\![e_2]\!]\rho[x \mapsto v]k)
\end{aligned}$$

The semantics for function expressions and applications must agree with our domain equation for *Function*, above:

$$\begin{aligned}
\mathcal{C}[\![\lambda x\ e]\!]\rho k &= k(\lambda v \in \mathit{Value}\,.\,\lambda k' \in \mathit{Cont}\,.\,\mathcal{C}[\![e]\!]\rho[x \mapsto v]k')\\
\mathcal{C}[\![e_0\ e_1]\!]\rho k &= \mathcal{C}[\![e_0]\!]\rho(\mathit{check\text{-}function}(\lambda f \in \mathit{Function}\,.\,\mathcal{C}[\![e_1]\!]\rho(\lambda v \in \mathit{Value}\,.\,f v k)))
\end{aligned}$$

The semantics for rec are a minor tweak for those to functions. This rule makes it clear that we need *Answer* to be a pointed domain, in order to take the necessary fixed point:

$$\mathcal{C}[\![\mathsf{rec}\ y\ (\lambda x\ e)]\!]\rho k = k(\mathit{fix}\ \lambda f \in \mathit{Function}\,.\,\lambda v \in \mathit{Value}\,.\,\lambda k' \in \mathit{Cont}\,.\,\mathcal{C}[\![e]\!]\rho[x \mapsto v, y \mapsto \mathit{in}_5(f)]k')$$

2 Non-hierarchical scope: Modules

The binding constructs we have seen so far are all hierarchical in nature. Each construct establishes a parent-child relationship between an outer context in which the declaration is not visible and an inner (body) context in which the declaration is visible. In static scoping, the hierarchy is determined by the abstract syntax tree, while in dynamic scoping, the hierarchy is determined by the tree of procedure calls generated at run-time. In both these scoping mechanisms, there is no natural way to communicate a declaration laterally across the tree-structure imposed by the hierarchy.

For small programs, this is not ordinarily a problem, but when a large program is broken into independent pieces, or modules, the constraint of hierarchy can be a problem. Modules connect and communicate with each other via collections of bindings; a module provides services by exporting a set of bindings, and makes use of other modules' services by importing bindings from those other modules. In a hierarchical language, the scope of a binding is a single region of program, so all the clients of a module must reside in the region where the module's bindings are in scope.

The traditional solution to the problem of communicating modules is to use a global namespace. All exported bindings from all modules are defined in a single environment, so all exported bindings are available to all modules. This approach is used in languages like C or FORTRAN. A C program is a bunch of top-level functions and in this way it is possible to call any function anywhere. A global namespace has some major drawbacks: In order to avoid accidental name collisions, every module must be aware of all definitions made by all other modules, even those definitions that are completely irrelevant; the dependencies between modules are poorly documented, making intermodule dependencies difficult to track, and leading to fragile code over time.

A way for languages to overcome the hierarchical scoping of binding constructs is to provide a value with named subparts. For this purpose, we will introduce a new module value that bundles up a set of bindings at one point in a program and can communicate them to a point that is related neither lexically nor dynamically to the declarations of those bindings, Typically, a module defines a set of named values, especially procedures, that provide a particular function. Some examples of these are: modules in ML, objects and classes in C++ and Java, and packages in Java.

2.1 Syntax

To introduce a simple module mechanism into uF, we have to add new expression forms to uF. Our modules will be extremely simple compared to the module mechanisms in other languages. In languages like ML and

Java, modules can define and export types as well as values; since we are in an untyped framework, the uF modules will only export values. Module mechanisms usually permit the definition of both internal names and exported names, where both kinds of names are in scope within the module, but external names may be referenced outside the module through the use of a special expression form. Our simple modules will only permit the definition of external names, and they may only be bound to lambda terms:

$$e \quad ::= \quad \ldots \mid \mathsf{module}\ y_1 = \lambda x_1\ e_1, \ldots, y_n = \lambda x_n\ e_n\ \ \mathsf{end} \mid e_m.x \mid \mathsf{import}\ e_m\ e$$

In one respect, these modules are more powerful than the modules found in most languages: they are *first-class values*. We could restrict module expressions so that they could only appear in a top-level let expression, which would make them easier to implement efficiently. However, first-class modules hold some interest of their own.

Modules are often used to implement *abstract datatypes*, in which a concrete representation is chosen for some datatype and the names exported from the module correspond to the datatype operations. For example, we might define a module rational that implements the abstraction of rational numbers. We'll represent a rational number as a pair of integers, although other representations are possible:

```
let rational = module
  create = λ p λ q ⟨p,q⟩,
  add = λ r1 λ r2 ⟨left r1*right r2+right r1*left r2, right r1*right r2⟩,
  ...
end in (rational.create 1 2) + import rational in (create 1 2)
```

Of course, we have no way to really defend the abstraction boundary that this module creates. Any user of the module can take one of the rational numbers it creates and pick it apart to discover it's actually a pair of numbers. As we'll see later, we can use a type system to prevent clients from misusing values of abstract datatypes in this way. Defensible abstraction boundaries are important for practical engineering of software, because they prevent different components of the system from becoming unnecessarily entangled. This makes reasoning about the system easier; it also facilitates the assignment of blame when something goes wrong!

2.2 Semantics

We need to extend the domain equations to allow new kinds of values – module values. Modules are similar to the naming environments ρ that we use elsewhere. However, because we are in an untyped language we have no way to prevent clients from accessing module exports that do not exist. Therefore modules must be total functions so we can tell when we have attempted to select a component that does not exist. If module definitions can be imported into the current naming environment, we also lose the nice property that we can tell at 'compile time' what names are in scope, so the naming environment *Env* must be total too. Once we introduce a typing discipline that lets us assign types to module expressions, we will be able to avoid this run-time error checking.

$$\begin{array}{rcl} \mathit{Value} & = & \ldots + \mathit{Module} \\ \mathit{Module} & = & \mathit{Var} \to (\mathit{Function} + \mathit{Unbound}) \\ \mathit{Env} & = & \mathit{Var} \to (\mathit{Value} + \mathit{Unbound}) \\ \mathit{Unbound} & = & \mathbb{U} \end{array}$$

We can now extend the semantic function $\mathcal{C}[\![\cdot]\!]$ to cover the next terms. In writing out the semantics, we immediately see that there are some language design choices that are easy to overlook in an informal presentation. One such question is whether, in a module definition, the names y_i are in scope in the definition bodies e_i. If not, the semantics for a module are easily defined:

$$\mathcal{C}[\![\mathsf{module}\ \ldots y_i = \lambda x_i\ e_i, \ldots \mathsf{end}]\!]\rho k = \mathit{empty\text{-}module}[\ldots, y_i \mapsto \mathit{in}_1(\lambda v \in \mathit{Value}\,.\,\lambda k' \in \mathit{Cont}\,.\,\mathcal{C}[\![e_i]\!]\rho k'), \ldots]$$

The empty module *empty-module* $\in$ *Module* is the module in which every name is unbound:

$$\mathit{empty\text{-}module} \overset{\mathit{def}}{=} \lambda x \in \mathit{Var}\,.\,\mathit{in}_2(u)$$

The semantics of import and selection are readily defined using a helper function *extend-with-module* $\in$ *Env* $\to$ *Module* $\to$ *Env*.

$$\begin{array}{rcl} \mathcal{C}[\![e_m.x]\!]\rho k & = & \mathcal{C}[\![e_m]\!]\rho(\mathit{check\text{-}module}(\lambda m \in \mathit{Module}\,.\,\mathit{case}\ m(x)\ \mathit{of}\ \mathit{in}_1(f).\mathit{in}_5(f)\ |\ \mathit{in}_2(u).\mathit{halt\ error})) \\ \mathcal{C}[\![\mathsf{import}\ e_m\ \mathsf{in}\ e_b]\!]\rho k & = & \mathcal{C}[\![e_m]\!]\rho(\mathit{check\text{-}module}(\lambda m \in \mathit{Module}\,.\,\mathcal{C}[\![e_b]\!](\mathit{extend\text{-}with\text{-}module}\ \rho\ m)k)) \end{array}$$

$$\mathit{extend\text{-}with\text{-}module} \overset{\mathit{def}}{=} \lambda \rho \in \mathit{Env}\,.\,\lambda m \in \mathit{Module}\,.\,(\lambda x \in \mathit{Var}\,.\,\mathit{case}\ m(x)\ \mathit{of}\ \mathit{in}_1(f).\mathit{in}_1(\mathit{in}_5(f))\ |\ \mathit{in}_2(u).\rho(x))$$

If we want the names y_i to be in scope within the module itself, we need to take a fixed point to construct the module, because the environment that the e_i's are interpreted with respect to must be extended with the y_i's. This is similar to the problem of defining the semantics of other mechanisms for mutual recursion such as $\mathsf{let\ rec}$. The obvious approach is to wrap a *fix* around the definition just given:

$$\begin{array}{l} \mathcal{C}[\![\mathsf{module}\ \ldots y_i = \lambda x_i\ e_i, \ldots \mathsf{end}]\!]\rho k = \\ \quad \mathit{fix}\,\lambda m \in \mathit{Module}\,.\,\mathit{empty\text{-}module}[\ldots, y_i \mapsto \mathit{in}_1(\lambda v \in \mathit{Value}\,.\,\lambda k' \in \mathit{Cont}\,.\,\mathcal{C}[\![e_i]\!](\mathit{extend\text{-}with\text{-}module}\,\rho\,m)k'), \ldots] \end{array}$$

However, this approach doesn't quite work because *Module* isn't pointed: its codomain is a sum. To avoid this problem, we observe that the module being defined can only be used within itself in a particular, limited way: to access its components y_i. Therefore, we can just take a fixed point over the tuple of the components, which *is* a pointed domain:

$$\begin{array}{l} \mathcal{C}[\![\mathsf{module}\ \ldots y_i = \lambda x_i\ e_i, \ldots \mathsf{end}]\!]\rho k = \\ \quad \mathit{let}\ t = \mathit{fix}\ \lambda t' \in \mathit{Function} \times \ldots \times \mathit{Function}\,. \\ \qquad \langle \ldots, \lambda v \in \mathit{Value}\,.\,\lambda k' \in \mathit{Cont}\,.\,\mathcal{C}[\![e_i]\!]\rho[y_1 \mapsto \pi_1(t'), \ldots, y_n \mapsto \pi_n(t')]k', \ldots \rangle \\ \quad \mathit{in} \\ \qquad \mathit{empty\text{-}module}[y_1 \mapsto \pi_1 t, \ldots, y_n \mapsto \pi_n t] \end{array}$$

Note that this wouldn't work if the domain of the components, *Function*, were not itself pointed. At least in a CBV language, mutually recursive module components are difficult to support if they can be bound to arbitrary expressions.