

Formal Programming Language Semantics note 16

More datatypes, and reasoning about programs

In this note we will consider some mild extensions of our language PCF, mainly involved with the addition of new datatypes, and show how to give operational and denotational semantics. We will also show how a denotational semantics can give us a framework for proving properties of programs, and will hint at the connection between denotational and *axiomatic* semantics.

Remember that a denotational semantics interprets *types* using mathematical objects such as CPOs, and *terms* as elements of these CPOs. Often the idea behind a denotational semantics is clear as soon as we have specified the interpretation of types — giving the denotation of terms is then usually just a matter of filling in the details. From this point on we will content ourselves with explaining how types are interpreted, leaving the semantics of terms as an exercise for the tireless reader.

Termination at higher types. Before we consider any new datatypes, let us look at a mild variant of PCF, somewhat akin to the call-by-value version considered in Note 14. First of all, notice that (for call-by-name PCF) our statement of adequacy only says anything about terms of *basic* type (`int` or `bool`). This seems reasonable because these are the only types for which values are “printable”, and thus provide a basic notion of “observable output” — we are not allowing ourselves the ability to make direct observations on terms of higher type. In particular, our denotational semantics doesn’t distinguish between the terms

```
fix (f:int->int = f)
fn x:int => fix (y:int = y)
```

In practice, however, in languages like Haskell or ML, one *can* distinguish observationally between these terms, since one can at least observe *termination* for programs of higher type. If you enter something like the first term at the Haskell prompt, the machine will go into an endless loop, but if you enter the second term, the machine will get back to you with something like

```
it = - : int->int
```

Thus, the first term corresponds to an element of type `int->int` which is itself undefined, and the second to a function which is itself defined but whose value at every `x:int` is undefined.

We can capture this idea in our operational semantics by extending the definition of *values* to include fn -abstractions, just as we did in call-by-value PCF. We may then say a term e of any type terminates if it evaluates to some value v using the reduction rules.

Suppose we want our denotational semantics to correctly predict the results of “experiments” of the above kind. We can achieve this by modifying our definition of the interpretation of types by defining

$$\llbracket \sigma \rightarrow \tau \rrbracket = (\llbracket \sigma \rrbracket \Rightarrow \llbracket \tau \rrbracket)_{\perp}$$

(the new bit here is the lifting operator \perp). Obviously, this entails some minor tweaking to our definition of the semantics of terms. Once this is done, the interpretation of the first of the above terms will be the newly added bottom element of $(\mathbb{Z}_{\perp} \rightarrow \mathbb{Z}_{\perp})_{\perp}$, whilst the interpretation of the second term will be the everywhere-bottom function from \mathbb{Z}_{\perp} to \mathbb{Z}_{\perp} , which is now the second-smallest element of $(\mathbb{Z}_{\perp} \rightarrow \mathbb{Z}_{\perp})_{\perp}$.

It’s worth noting that in *call-by-value* PCF, termination at higher types is observable automatically: to see where $e : \tau$ terminates, we can consider a basic type term such as $(\text{fn } x : \tau \Rightarrow 0)(e)$.

Product types. Supposing we wish to add product types to our original version of PCF. We can extend the language of types thus:

$$\tau ::= \text{int} \mid \text{bool} \mid \tau_0 \rightarrow \tau_1 \mid \tau_0 * \tau_1$$

We can also extend the language of terms by adding “pairing” and “projection” operators with typing rules as follows:

$$\frac{\Gamma \vdash e : \sigma \quad \Gamma \vdash e' : \tau}{\Gamma \vdash (e, e') : \sigma * \tau} \quad \frac{\Gamma \vdash e : \sigma * \tau}{\Gamma \vdash \#1 e : \sigma} \quad \frac{\Gamma \vdash e : \sigma * \tau}{\Gamma \vdash \#2 e : \tau}$$

One way to give a denotational semantics of this language would be to interpret product types using the obvious notion of “product of CPOs”:

$$\llbracket \sigma * \tau \rrbracket = \llbracket \sigma \rrbracket \times \llbracket \tau \rrbracket$$

and then to interpret pairing and projections using the evident mathematical pairing and projection functions. This would correspond precisely to giving an operational semantics with the following small-step reduction rules:

- $\#1 (e, e') \rightarrow e$, and $\#2 (e, e') \rightarrow e'$.
- If $e \rightarrow e'$ then $\#1 e \rightarrow \#1 e'$ and $\#2 e \rightarrow \#2 e'$.
- If $e \rightarrow e'$ then $(e, e'') \rightarrow (e', e'')$ and $(e'', e) \rightarrow (e'', e')$.

This represents one possible choice for the semantics of products. Note in particular that in this semantics, a term such as $\#1 (0, \text{diverge})$ has value 0. Intuitively, we do not have to reduce expressions to values before pairing them, so a pair can contain useful information even if one component is “undefined”. Products of this kind are often called *lazy products*.

Many languages, however, employ *strict products*, in which (intuitively) a pair is only defined if both its components are defined. In such a language, the term $\#1 (0, \text{diverge})$ would go undefined. An operational semantics for this kind of product could be given by replacing the first of the above clauses by:

- $\#1 (v, v') \rightarrow v$, and $\#2 (v, v') \rightarrow v'$.

Here v and v' range over *values* — that is to say, integer or boolean literals, fn - expressions, or pairs of things that are themselves values (note that this is now an inductive definition!)

In a denotational semantics for strict products, we would expect the three terms $(0, \text{diverge})$, $(\text{diverge}, 0)$, $(\text{diverge}, \text{diverge})$ all to denote the same element \perp . To give such a semantics, it is as well to be in a setting that is sensitive to termination at all types. As for call-by-value PCF, for each type σ we define a CPO $[\sigma]$ thought of as the domain for *values* of type σ , and a CPO-with-bottom $[\sigma]_{\perp}$ thought of as the domain for arbitrary terms:

$$[\text{int}] = \mathbb{Z}, \quad [\text{bool}] = \mathbb{T}, \quad [\sigma \rightarrow \tau] = [\sigma]_{\perp} \Rightarrow [\tau]_{\perp}, \quad [\sigma * \tau] = [\sigma] \times [\tau], \quad \llbracket \sigma \rrbracket = [\sigma]_{\perp}$$

(Notice that the corresponding call-by-value version is obtained just by replacing $[\sigma]_{\perp}$ by $[\sigma]$ in the interpretation of arrow types.)

As an aside, once we have product types around it is quite easy to give an interpretation of *mutually recursive* function definitions. Suppose $f: \sigma$ and $g: \tau$ are ML-style functions that are defined in terms of each other (and perhaps themselves as well). This amounts to having a pair (f, g) of type $\sigma * \tau$ defined in terms of itself, so we can give the denotation of such a pair by applying the usual fixed-point machinery to the type $\sigma * \tau$. This will result in a simultaneous least solution to the defining equations for f and g respectively.

List types. Next, let us see how to add list types to our language. We will here consider only “strict lists”; question 3 of Exercise Sheet 3 invites you to consider one possible version of “lazy lists” (which are potentially infinite).

Let us extend the language of types by adding the production rule

$$\tau ::= \tau_0 \text{ list}$$

and extend the language of terms by adding constants

$$\begin{array}{ll} \text{nil}_{\tau} : \tau \text{ list} & \text{cons}_{\tau} : \tau \rightarrow \tau \text{ list} \rightarrow \tau \text{ list} \\ \text{head}_{\tau} : \tau \text{ list} \rightarrow \tau & \text{tail}_{\tau} : \tau \text{ list} \rightarrow \tau \text{ list} \\ \text{isNil}_{\tau} : \tau \text{ list} \rightarrow \text{bool} & \end{array}$$

We also extend our definition of *values* by stipulating: nil_τ is a value, and if v, v' are values of suitable types then $\text{cons}_\tau v v'$ is a value. Thus, any value of type $\tau \text{ list}$ will be of the form

$$\text{cons}_\tau v_1 (\text{cons}_\tau v_2 (\cdots (\text{cons}_\tau v_k \text{nil}_\tau) \cdots))$$

where k may be 0, and the v_i (if there are any) are all values. We abbreviate such a list to $[v_1, \dots, v_k]$.

It is straightforward to supply a suitable set of reduction rules for this language, e.g.:

- $\text{isNil}_\tau \text{nil}_\tau \rightarrow \text{true}$, and $\text{isNil}_\tau (\text{cons}_\tau e e') \rightarrow \text{false}$.
- $\text{head}_\tau (\text{cons}_\tau e e') \rightarrow e$, and $\text{tail}_\tau (\text{cons}_\tau e e') \rightarrow e'$.
- ... etc. [Exercise: finish this off!]

Let us now try to design a suitable CPO for values of type $\tau \text{ list}$, assuming we already have some suitable CPO X for values of type τ . Clearly, we can classify lists according to their length $0, 1, 2, \dots$. If v and v' are lists of different length, we can easily write a function which sends v to *true* and v' to *false*; we would therefore not expect v and v' to be comparable via the order relation \sqsubseteq . The appropriate CPO will therefore consist of an infinite family of CPOs placed “side-by-side”: one for lists of length 0, another for lists of length 1, another for lists of length 2 and so on. There is only one list of length 0 — namely nil_τ — so the first (or rather the “zeroth”) CPO in this family will be just the one-element CPO, which we will denote by 1. A list of length 1 just contains a single value of type τ , so here we expect the corresponding CPO to be (isomorphic to) X itself. A list of length 2 is effectively a pair of values of type τ , so we expect the corresponding CPO to be essentially $X \times X$, and so on. We may therefore define the CPO of *lists over X* as:

$$\text{List}(X) = 1 + X + (X \times X) + (X \times X \times X) + \cdots$$

Here we use “+” for the evident operation of putting CPOs side by side (actually we are using an infinitary version of this operation). It is an easy exercise to give a rigorous definition of this operation. We may now complete our semantics of types by defining $[\tau \text{ list}] = \text{List}([\tau])$.

There is another way to look at this CPO of lists. Intuitively, by appealing to an “infinite distributivity law”, one can recast the above definition as

$$\text{List}(X) \cong 1 + X \times (1 + X \times (1 + X \times \cdots \quad \cdots))$$

(This may look like an appeal to dark magic. In fact it can be made perfectly rigorous, but we will not do so here.) This recasting makes it clear that

$$\text{List}(X) \cong 1 + X \times \text{List}(X)$$

This is not at first sight a definition of the CPO $\text{List}(X)$, but rather a property that we would expect it to satisfy. This is rather reminiscent of the situation when we first tried to define the semantics of `while` loops (see Note 10). In that situation, as we have seen, we were able to interpret such properties as genuine definitions by invoking the existence of least fixed points. The difference is that there we were trying to specify an *element* of a given CPO by means of a circular definition, whereas here we are trying to give a circular definition of a CPO itself.

In fact, with a bit more mathematical theory, we can get the theory of fixed points to work here as well, and in fact we can *define* the CPO $\text{List}(X)$ as the “smallest” solution to the above equation. Roughly speaking, we do this by showing that the class of all CPOs is itself something like a gigantic CPO (whose elements are ordinary CPOs), and that therefore any operation on CPOs that we can write down (such as the operation that maps a CPO Y to $1 + X \times Y$) has a “least fixed point”.

In programming language terms, the upshot of all this is that we can interpret recursive definitions at the level of *types* as well as of terms. Just as the ordinary theory of fixed points allows us to make sense of ML-style recursive definitions such as

```
fun f 0 = 1 | f n = n * f (n-1)
```

so the theory of fixed points on the class of all CPOs allows us to make sense of recursive type declarations such as

```
datatype 'a list = nil | cons of 'a * 'a list
```

(which is how one could define the type of lists if it were not already predefined). The same theory allows us to interpret many other recursive datatypes beloved of functional programmers, such as (labelled or unlabelled) trees, lazy lists, lazy trees and so on.

With these ideas in place, we now have essentially enough to give a denotational semantics for a fairly respectable functional programming language (such as the “pure functional” fragment of ML).

John Longley