Formal Programming Language Semantics note 14

Operational semantics of PCF

In the interests of historical realism, and also for the sake of variety, we decided to give a denotational semantics of **PCF** first. However, it is not immediately obvious from the denotational semantics whether **PCF** is "executable" in any reasonable sense, i.e. whether there is a reasonable procedure for evaluating **PCF** expressions. (If there isn't, we can hardly call **PCF** a programming language!) Here we address this issue by showing how to give an equivalent operational semantics. This was first done by Plotkin in his classic 1977 paper "LCF considered as a programming language".

A small-step semantics. For the sake of variety, we will give a *small-step* operational semantics for PCF, in order to illustrate this style of semantics.[1] We will concentrate on the "pure" fragment of PCF consisting only of expressions. (Clearly, any program involving previously declared variables can be translated to a "pure PCF" term just by expanding out all the definitions — assuming we are thinking of static binding, which I am.) We are only really interested (here) in defining the result of evaluating terms of *ground* type (`int` or `bool`) — a term of function type will not result in a usable "value", but will merely sit there waiting to be applied to something. Let us therefore define a *value* to be either an integer literal n or a boolean literal t; we use v to range over values. We will now give rules to define a relation $e \to e'$, where e and e' are *closed* terms of the same type. This relation may be read as "e reduces to e'", and is supposed to capture the idea of a single computation step. First, we have the following rules for directly reducing expressions of various shapes:

$$
\begin{array}{rll}
n_0 \texttt{ - } n_1 & \to n & (n = n_0 - n_1) \\
n_0 \texttt{ = } n_1 & \to \textit{true} & (n_0 = n_1) \\
n_0 \texttt{ = } n_1 & \to \textit{false} & (n_0 \neq n_1) \\
\texttt{if } \textit{true} \texttt{ then } e_0 \texttt{ else } e_1 & \to e_0 & \\
\texttt{if } \textit{false} \texttt{ then } e_0 \texttt{ else } e_1 & \to e_1 & \\
\texttt{(fn } x \texttt{ => } e_0 \texttt{)} e_1 & \to e_0[e_1/x] & \\
\texttt{fix (} x : \tau \texttt{ = } e \texttt{)} & \to e[\texttt{fix } x : \tau \texttt{ = } e/x] &
\end{array}
$$

Here $e_0[e_1/x]$ means the result of replacing all free occurrences of x in e_0 by e_1 (we should not replace occurrences of x underneath a binder $\texttt{fn } x \texttt{ => } \cdots$ within e_0). We also need some other rules to allow reductions to be applied to subexpressions within a larger expression. Specifically, if $e \to e'$ then

[1] From now on we will work with the version of PCF with `Fix` rather than the one with `fun` declarations as primitive.

$$\begin{array}{rcl} e\,e_0 & \to & e'\,e_0 \\ \texttt{if}\ e\ \texttt{then}\ e_0\ \texttt{else}\ e_1 & \to & \texttt{if}\ e'\ \texttt{then}\ e_0\ \texttt{else}\ e_1 \\ e\ \texttt{-}\ e_0 & \to & e'\ \texttt{-}\ e_0 \\ n\ \texttt{-}\ e & \to & n\ \texttt{-}\ e' \\ e\ \texttt{=}\ e_0 & \to & e'\ \texttt{=}\ e_0 \\ n\ \texttt{=}\ e & \to & n\ \texttt{=}\ e' \end{array}$$

whenever these terms are well-typed. (If preferred, one could write all these as inference rules with a single premise $e \to e'$.) As usual, we take $e \to e'$ to be the *smallest* relation with the above properties.

We now define $\to^*$ to be the reflexive-transitive closure of $\to$ (i.e. $e \to^* e'$ if e can be reduced to e' in zero or more steps). Finally, if e is a closed term of ground type, we write $e \Downarrow v$ to mean that $e \to^* v$ and v is a value.

This completes the operational semantics of PCF. One rather nice feature of this approach is that we have no need for any notion of environment: we can work entirely with closed terms by themselves.

Note that the relation $\to$ is *deterministic*: for any e there is at most one e' such that $e \to e'$. It follows easily that $\Downarrow$ is also deterministic. In fact, the above rules enforce a certain evaluation strategy for programs: for instance, to run a program $e_0\,e_1$, first reduce e_0 as much as possible, then if this yields an abstraction $\texttt{fn}\ x\ \texttt{=>}\ e$, replace x by e_0 in e_1 and continue.

The adequacy theorem. As with **IMP**, we have a central theorem saying that our operational semantics agrees with the denotational semantics of Note 12: For any closed term e of ground type τ, and any value v of type τ,

$$e \Downarrow v \ \text{ iff } \ [\![e]\!] = v.$$

(Strictly speaking, since e is closed, its interpretation $[\![e]\!]_\emptyset$ is a function from a one-element set to $[\![\tau]\!]$; we are writing $[\![e]\!]$ to refer to the element of $[\![\tau]\!]$ picked out by this function. Note that v ranges over *non-bottom* elements of $[\![\tau]\!]$.)

The proof of the left-to-right implication is straightforward, and follows the same pattern as the corresponding proof for **IMP**: we are assuming that $e \Downarrow v$ and can argue by induction on the derivation of this fact. Since we are here using a small-step semantics, it suffices to show as a lemma that if $e \to e'$ then $[\![e]\!] = [\![e']\!]$, and this is shown by an induction on derivation trees with one case for each of the rules given above.

The proof of the right-to-left implication is harder and more interesting. We are assuming that $[\![e]\!] = v$ and so would like to reason by induction on the syntactic structure of e. However, there is a problem: the thing we are trying to prove only makes sense for closed terms of ground type, and though e is a term of this kind, it may well have subterms which contain free variables and/or are of higher type. In order for the induction to go through, we therefore need to formulate a suitable induction claim which makes sense for arbitrary subterms of

e, and which specializes to the property we are interested in in the case of closed terms of ground type. This is a particularly striking example of a phenomenon which arises very often in connection with proofs by induction: we often have to prove something significantly stronger than what we are ultimately interested in, and finding the right thing to prove can require considerable ingenuity.

In order to formulate the appropriate induction claim, let us introduce the notion of a *good* PCF term, defined as follows:

- A closed term e of ground type is good if $[\![e]\!] = v$ implies $e \Downarrow v$ (note that this is the property we are ultimately interested in).
- A closed term $e : \sigma$ `->` τ is good if for all closed $e' : \sigma$, if e' is good then $e\,e'$ is good. (In other words, if $e\,e'$ is bad then it is not the fault of e.)
- A term e with free variables $x_1 : \tau_1, \cdots, x_r : \tau_r$ is good if for all good closed terms $e_1 : \tau_1, \cdots, e_r : \tau_r$, the term $e[e_1/x_1, \ldots, e_r/x_r]$ is good.

We can now prove that *all terms are good* (notice that this immediately implies the right-to-left half of the adequacy theorem). The proof of this is now a straightforward induction on the structure of e, with just a little effort needed in the case of the `Fix` construct [exercise!!]. The hard part of the proof lay in formulating an appropriate definition of "goodness".[2]

As was the case with **IMP**, we now have an adequate and compositional denotational semantics, so it is easy to see that if $[\![e]\!] = [\![e']\!]$ (where e, e' are closed terms of any type) then e and e' are observationally equivalent in PCF (that is, for any context $C[-]$ and value v we have $C[e] \Downarrow v$ iff $C[e'] \Downarrow v$). So one possible use for the denotational semantics is to establish observational equivalences. We will also see other uses of our adequacy result later when we consider axiomatic semantics for PCF.

[Exercise: show that for any term e with free variables $x, y : \tau$, the three terms

`fix (fn` $x : \tau$ `= fix (fn` $y : \tau$ `=` e`))`, `fix (fn` $x : \tau$ `=` $e[x/y]$`)`,
`fix (fn` $y : \tau$ `= fix (fn` $x : \tau$ `=` e`))`

are observationally equivalent. This can be used to show that various ways of expressing *mutual recursion* in PCF are in fact equivalent. It would be quite fiddly to establish these equivalences by purely operational reasoning.]

Call-by-value PCF. You may have noticed that the version of PCF we have presented is a call-by-name language (like Haskell, but unlike ML). This is manifest in the rule for applying a function to an argument:

$$(\texttt{fn}\ x \texttt{=>} e_0)e_1 \ \rightarrow\ e_0[e_1/x].$$

Here we do not evaluate the argument e_1 before passing it to the function, but rather pass the *expression* e_1 as the argument and only evaluate it if and when it

[2] This proof is essentially due to Gordon Plotkin, building on earlier ideas of Tait.

becomes necessary. We will now briefly look at how one would modify both the operational and denotational semantics for a call-by-value language.

First, we need to make a tiny change to the syntax of the language: in expressions `fix` $x : \tau$ `=` e we insist that τ must be a function type (i.e. not `int` or `bool`). This is because, as we shall see, we want to delay the application of the reduction rule for `fix` until an extra argument has been supplied.

In a call-by-value language, only *values* may be passed as parameters. Since a parameter may itself be of function type, we will need a notion of what is meant by a value at function types. So let us revise our definition of a *value* to include integer literals, boolean literals, `fn`-abstractions (that is, terms of the form `fn` x `=>` e), and `fix` expressions. The idea is that if we try to evaluate a term of function type, the computation will "terminate" if we can reduce it as far as either a `fn`-abstraction or a `fix`-expression — we cannot really progress any further than this until an actual argument to the function is supplied. As before, we let v range over values.

We now replace the small-step rules for `fn` and `fix` by

$$\begin{array}{rcl} \texttt{(fn}\, x \,\texttt{=>}\, e_0\texttt{)}v & \rightarrow & e_0[v/x] \\ \texttt{fix}\, e\, v & \rightarrow & e\texttt{(fix}\, e\texttt{)}v \end{array}$$

and add one further rule: if $e \rightarrow e'$ then $v\,e \rightarrow v\,e'$. Thus, to evaluate an expression $e_0 e_1$, we would first reduce e_0 to a value `fn` x `=>` e, then reduce e_1 to a value v, then pass v as a parameter to the `fn`-abstraction. [Exercise: satisfy yourself that our revised operational semantics accurately models evaluation behaviour in ML, both at ground types and at function types.]

Now for the denotational semantics. First, consider how we ought to model a function of type `int->int`. Whereas previously this was modelled by a function $\mathbb{Z}_\perp \rightarrow \mathbb{Z}_\perp$, in a call-by-value language there is no possibility of passing a non-terminating argument as a parameter, so a function $\mathbb{Z} \rightarrow \mathbb{Z}_\perp$ would seem more appropriate. [Exercise: Give two terms of type `int->int` which have different observable behaviour under call-by-name but the same behaviour under call-by-value.] In fact, for each type τ it will be convenient to define two CPOs: a CPO $[\tau]$ specifically for modelling *values* of type τ, and a CPO $[\![\tau]\!]$ for modelling arbitrary terms. The definitions of these CPOs may be given as follows:

$$[\texttt{int}] = \mathbb{Z}, \quad [\texttt{bool}] = \mathbb{T}, \quad [\tau\ \texttt{->}\ \tau'] = [\tau] \Rightarrow [\tau']_\perp; \qquad [\![\tau]\!] = [\tau]_\perp$$

(We have made use here of the *lifting* operation on CPOs: for any CPO X, we write $X_\perp$ for the CPO obtained by adding a new least element, typically called $\perp$.)

Once we have the denotation of types in place, it is not too hard to figure out the appropriate changes to the definition of the denotation of terms for call-by-value PCF (though the details are a bit fiddly). We can also prove an adequacy theorem (exactly as stated above) for call-by-value PCF, reassuring us that our operational and denotational semantics agree.

John Longley