1 Introduction

Up to this point, we have been looking at the denotational semantics for languages with deterministic evaluation. What happens when we add nondeterminism? Nondeterminism could result from several sources. One possibility is that some terms in the language have more than one reduction rule. Even though a particular implementation might choose a particular reduction rule, the language semantics should reflect all the possible evaluations. The language might also support explicit features for nondeterminism. For example, we might add to IMP a nondeterministic choice operator $c_1 \,[]\, c_2$, which executes either command c_1 or c_2, but does not say which. Finally, concurrency usually introduces nondeterminism to programming languages, because the various operations performed by different threads can be interleaved in different ways.

Let us try to construct a denotational semantics for IMP with the construct $c_1 \,[]\, c_2$. Notice that from the operational perspective, the semantic extension is simple; we just add the following rules to the small-step rules:

$$\frac{}{\langle c_1 \,[]\, c_2, \sigma\rangle \longrightarrow \langle c_1, \sigma\rangle} \qquad \frac{}{\langle c_1 \,[]\, c_2, \sigma\rangle \longrightarrow \langle c_2, \sigma\rangle}$$

Recall that the meaning of a command c is a function from states to lifted states: $\mathcal{C}[\![c]\!] : \Sigma \to \Sigma_\perp$. Since a nondeterministic program can have more than one outcome, a natural choice of semantics would seem to be a function that maps to a nonempty *set* of states (or bottom). That is, $\mathcal{C}[\![c]\!] : \Sigma \to (\wp(\Sigma_\perp) - \emptyset)$, where $\wp(S)$ is the powerset of S, also written as 2^S. Interestingly, we can write the nondeterministic semantics by making only small modifications to the previous semantics, and in fact, we can write the semantics in a more general way that captures both semantics.

1.1 Expressing the meaning generically

To avoid having to write $(\wp(\Sigma_\perp) - \emptyset)$ everywhere, let's define a *functor* M that is applied to Σ to obtain either $M(\Sigma) = \Sigma_\perp$ (in the case of the deterministic semantics) or $M(\Sigma) = \wp(\Sigma_\perp) - \emptyset$ (in the case of the nondeterministic semantics. A functor is a function on domains that maps both the elements of the domain and the $\sqsubseteq$ relationships between elements on the domain. So the various domain constructions that we have seen already (lifting, products, sums, and so on) are functors.

In writing the semantics, we make use of two key operations:

- The *unit* operation converts an element of D into an element of $M(D)$. We will write $[d]$ to mean the lifting of $d \in D$ into $M(D)$. We will be using this operation only on $D = \Sigma$.

  For example, in the original IMP semantics, $[d] = \lfloor d \rfloor$, since $\lfloor \cdot \rfloor$ has the signature $\Sigma \to M(\Sigma)$.

- The *bind* operation converts a computation of the form $D \to M(E)$, which expects a simple state or value from D, and constructs a computation that can receive the result of a computation of D. We need this ability to sequence two computations to capture the meaning of **if** and **while**. The signature of *bind* is therefore $(D \to M(E)) \to (M(D) \to M(E))$. We will be using it only with $D = E = \Sigma$.

  We will write f^* to mean the application of the bind operation to the function f. This is the same notation we used in the original IMP semantics, where if $f \in \Sigma \to \Sigma_\perp$, then $f^* \in \Sigma_\perp \to \Sigma_\perp$, and was defined as $f^*(\perp) = \perp$ and $f^*(d) = \lfloor d \rfloor$ for $d \neq \perp$.

1.2 A generic semantics

With these two functions, we can write the semantics more generically:

$$
\begin{aligned}
\mathcal{C}[\![\textbf{skip}]\!]\sigma &= [\sigma] \\
\mathcal{C}[\![x := a]\!]\sigma &= [\sigma[x \mapsto \mathcal{A}[\![a]\!]\sigma]] && \text{(unfortunately there are two different kinds of brackets here)} \\
\mathcal{C}[\![\textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2]\!]\sigma &= \text{if } \mathcal{B}[\![b]\!]\sigma \text{ then } \mathcal{C}[\![c_1]\!]\sigma \text{ else } \mathcal{C}[\![c_2]\!]\sigma \\
\mathcal{C}[\![c_1; c_2]\!]\sigma &= \mathcal{C}[\![c_2]\!]^*(\mathcal{C}[\![c_1]\!]\sigma) && \text{(here the bind operation is used to feed the result of } c_1 \text{ to } c_2\text{)} \\
\mathcal{C}[\![\textbf{while } b \textbf{ do } c]\!] &= \text{fix}\lambda w \in \Sigma \to M(\Sigma). \\
&\qquad \lambda\sigma \in \Sigma. \text{ if } \mathcal{B}[\![b]\!]\sigma \text{ then } w^*(\mathcal{C}[\![c]\!]\sigma) \text{ else } [\sigma]
\end{aligned}
$$

These semantics don't care what functor M is chosen, so we can give a nondeterministic version of the semantics by choosing the functor M and the unit and bind operations differently. Here is a first cut:

$$
\begin{aligned}
M(\Sigma) &= \wp(\Sigma_\perp) - \emptyset && \text{(as above)} \\
[\sigma] &= \{\lfloor\sigma\rfloor\} \\
f^*(S : M(\Sigma)) &= (\bigcup_{\lfloor\sigma\rfloor \in S} f(\sigma)) \cup (\{\perp\} \cap S)
\end{aligned}
$$

With this domain for the computation, we can express the semantics of nondeterministic choice simply:

$$\mathcal{C}[\![c_1 \,[]\, c_2]\!]\sigma = \mathcal{C}[\![c_1]\!]\sigma \cup \mathcal{C}[\![c_2]\!]\sigma$$

However, there is still one problem we haven't addressed: how do we know that $\Sigma \to M(\Sigma)$ is a CPO, so that we can take a fixed point over it? We know that it is a CPO if $M(\Sigma)$ is a CPO, but this means we need to have an ordering on $\wp(\Sigma_\perp) - \emptyset$, and one in which we can take least upper bounds.

2 Powerdomain constructions

Given a pointed CPO D, we can construct a new domain called the *powerdomain* whose elements are nonempty subsets of D. There are three standard powerdomain constructions, corresponding to different ways of thinking about nondeterminism. These different constructions define the ordering on the subsets of D differently, which in turn leads to powerdomains whose carrier sets are different.

2.1 The Hoare powerdomain

To obtain the Hoare, or *lower* powerdomain, we use the following ordering on two sets X and Y. Intuitively, it says that everything X can do, Y can do better:

$$X \sqsubseteq Y \iff \forall x \in X.\, \exists y \in Y.\, x \sqsubseteq y$$

This ordering is not a partial order, because it fails antisymmetry. In partial, consider the *downward closure* of X, written $\downarrow X$:

$$\downarrow X = \{x' \in D \mid \exists x \in X. x' \sqsubseteq x\}$$

A downward closed set X, where $X = \downarrow X$, is called a *downset* for short.

According to the ordering above, we have for any set X both $X \sqsubseteq \downarrow X$ and $\downarrow X \sqsubseteq X$. For every element in X there is trivially an element at least as high in $\downarrow X$, so $X \sqsubseteq \downarrow X$. Going the other direction, for every element in $\downarrow X$, there is also an element at least as high as it in X. Thus, the ordering fails antisymmetry, which means that $\sqsubseteq$ is a *preorder* rather than a partial order.

However, we can define an equivalence relation on sets: if $X \sqsubseteq Y$ and $Y \sqsubseteq X$, we write $X \sim Y$. This must be an equivalence relation (i.e., reflexive, symmetric, and transitive) because $\sqsubseteq$ is a preorder.

For any set X, there is then a whole collection of equivalent sets, of which $\downarrow X$ is the best representative: it must be the downward closure of every set equivalent to X. So we can focus just on the downsets as being the meaningful sets.

Since downsets always include $\bot$, the semantics doesn't tell us very much about divergence. Lower powerdomains give an *angelic* semantics to nondeterminism. For example, consider the following three programs that we can use to test the semantics of nondeterminism:

1. x := 1
2. x := 1 □ while true do skip
3. while true do skip

Under the Hoare semantics, test programs 1 and 2 have exactly the same semantics; only program 3 differs.

Now, suppose that we have a chain of such downsets such that $S_0 \sqsubseteq S_1 \sqsubseteq S_2 \sqsubseteq \ldots$. By the definition of the ordering above, we must have $S_0 \subseteq S_1 \subseteq S_2 \subseteq \ldots$. Intuitively, we'd like to take the union of the whole chain to find the least upper bound. However, this isn't enough in general. For example, consider $D = [0, 1]$ ordered in the usual way so that $\bot = 0$. The union of the whole chain $[0, 1/2] \sqsubseteq [0, 2/3] \sqsubseteq [0, 3/4] \sqsubseteq \ldots$ is $[0, 1)$. But this is problematic. For example, it means that the unit operation is not continuous, because when applied to the LUB of the chain $0 \sqsubseteq 1/2 \sqsubseteq 2/3 \sqsubseteq \ldots$, it gives a different answer ($[0, 1]$) than when the LUB is taken after the unit operation is applied to all the elements of that chain.

The solution to this problem is to only include sets in the powerdomain in which all chains have LUBs. Such sets are said to be *Scott-closed*, after Dana Scott. Given a set $S \subseteq D$, we can take its Scott closure by adding to it all elements of D that are LUBs of chains within S. We write $\overline{S}$ to denote the Scott closure of S.

2.2 The Smyth powerdomain

The Smyth powerdomain is based on a different ordering between sets.

$$X \sqsubseteq Y \iff \forall y \in Y.\, \exists x \in X.\, x \sqsubseteq y$$

This ordering says that everything Y can do is approximated by some behavior of X.

With this ordering every set is equivalent to its *upward* closure:

$$\uparrow X = \{x' \in D \mid \exists x \in X. x \sqsubseteq x'\}$$

Upward closed sets are called *upsets*. Given a chain of upsets $S_0 \sqsubseteq S_1 \sqsubseteq S_2 \sqsubseteq \ldots$, we have $S_0 \supseteq S_1 \supseteq S_2 \supseteq \ldots$, so we can take the LUB of the chain using intersection:

$$\bigsqcup S_i = \bigcap S_i$$

With the upper powerdomain, any program that can diverge has the semantics $\uparrow \{\bot\} = D$. This is a *demonic* semantics. Considering the three test programs above, the second two programs have the same semantics according to the Smyth powerdomain; only program 1, which cannot diverge, differs.

2.3 The Plotkin powerdomain

The Plotkin, or convex, powerdomain uses the Egli-Milner ordering on sets, which is the intersection of the Hoare and Smyth orderings:

$$X \sqsubseteq Y \iff (\forall x \in X.\, \forall y \in Y.\, x \sqsubseteq y) \wedge (\forall y \in Y.\, \exists x \in X.\, x \sqsubseteq y)$$

Consider a set X containing two elements x and y. This set is equivalent to a set containing those two elements plus any or all elements between x and y:

$$\{x, y\} \sim \{z \mid x \sqsubseteq z \sqsubseteq y\}$$

The similarity of this to the geometric definition of convexity explains why this is called the convex powerdomain. With this powerdomain construction, appealingly, all three test programs have different semantics, corresponding to our intuition that they are different.

The elements of the convex powerdomain are sets that are closed under the addition of all such intermediate elements z, and also Scott-closed. The LUB of a chain can then be taken as the intersection of the Hoare and Smyth LUBs:

$$\bigsqcup S_i = (\bigcup \uparrow S_i) \cap \overline{(\bigcap \downarrow S_i)}$$

3 Comments

We have only scratched the surface of the domain theory associated with powerdomains, exposing just enough to make sense of the semantics of IMP with nondeterminism. In languages with higher-order functions and recursive types, domains need even more structure than just being CPOs. Ensuring that the various domain constructions, such as powerdomains, preserve these properties is correspondingly more challenging. There are also other useful powerdomain constructions, such as probabilistic powerdomains, which map sets of possible outcomes to probabilities.

As a result of the mathematical challenges posed by domain theory, denotational semantics has fallen out of favor compared to its heyday in the 70's and 80's. Operational semantics has become the most common approach because it suffices to prove many of the important properties that we care about, such as strong typing and normalization properties. However, there are signs that denotational semantics is becoming more important again, especially in recent work that reasons more deeply about the semantics of probabilistic and reactive programming languages.