Review of
Control Flow Semantics*

Riccardo Pucella

Department of Computer Science
Cornell University

April 19, 2001

I have to admit, I was looking forward to reviewing this book. It answered what was for me a 6-year old question. Six years ago, I was pursuing a Master's degree at McGill University. working on programming language semantics. My training in semantics was rather traditional: lambda calculus, functional programming languages, denotational semantics based on complete partial orders, etc. At the time, Franck van Breugel was visiting McGill, and I came across the fact that Franck was also working on semantics of programming languages, but on semantics based on metric spaces. For someone with an undergraduate background in mathematics, this was seriously intriguing. Unfortunately, I never got around to asking Franck about his work. This book is an answer to that question that never was.

The purpose of the book is to describe an approach to provide semantics to imperative languages with various types of control flow models, including concurrency. The approach handles both operational semantics and denotational semantics, all in a topological setting. (We will come back to this later.) The issue of relating the two semantics for any given language is a central theme of the approach. I will provide in the next section an introduction to topological semantics. For now, let me say a word on the applicability of the approach. As stated, the main interest is in providing semantics to *imperative* languages. Imperative programs can be thought of, for our purposes, as sequences of actions performed on a state. Typically, states are sets of variables, and actions include modifying the value of a variable in a state. An important characteristic of imperative programs is that they embody the notion of a *computation step*: a program being a sequence of actions, it forces a sequence of intermediates states. Typically, the intermediate states are *observable*, meaning that one can observe something about that intermediate state, either by looking up the value of a variable, by witnessing an output operation, etc. When the intermediate states of a computation are observable, it becomes reasonable to talk about infinite (nonterminating) computations. (The classical example of this is of course an operating system, which at least theoretically is an infinite process; the main motivation for the topological approach, as we shall see, is to make sense of such infinite computations.) Contrast this with functional languages, which are often used as motivating examples for the study of semantics. In a pure functional language, all infinite programs are equivalent, and in a traditional denotational semantics based on partial orders (à la Scott and Strachey [6]), every nonterminating program is mapped to $\perp$, the least element of the appropriate partial order. This is not helpful in a setting where we want to discuss observably different infinite behaviors.

We distinguish two kinds of imperative languages. The distinction between them is akin to the distinction between propositional and first-order logic.[1] *Uniform languages* are based on a primitive, abstract notion of action, combined with appropriate operators. For instance, a program may have the form $a; b; c; (d + a)$ where ; represents sequential composition and $+$ a nondeterministic choice operator. The primitive actions a,b,c,d are uninterpreted. The state is implicitly defined by the actions that have been performed. On the other hand, *nonuniform languages* have an interpretation associated with the actions; typically, as we mentioned, the state is a set of variables, and actions include modifying the value of a variable in a state.

To showcase the versatility of their approach, the authors study different languages. The main difference between the various languages, aside from the question of uniformity, is the *program composition operators* considered. The

*J. de Bakker, E. de Vink, *Control Flow Semantics*, MIT Press, 1996, 564pp, ISBN 0262041545.

[1]This analogy can be made precise when looking at dynamic logics, a family of logics for reasoning about programs in such languages [1].

following groups of related operators are studied:

- The first group consists of operators including *sequential composition* and *choice*. The latter introduces nondeterminism in the framework, with suitable complications. Many versions of choice are investigated, including backtracking choice.
- The second group of operators consists of *recursion* and *iteration*. Such operators are required to get universality (in the computability theory sense).
- The third group of operators includes *parallel composition* operators. Modeling such operators forces one to deal with issues such as deadlock, synchronization and communication. Languages with such operators include CSP [2] and CCS [3].
- Related to the last group of operators, one may distinguish between *static* and *dynamic* configuration of processes.
- Finally, we can investigate issues of *locality* and *scope* of variables.

All in all, 27 languages are studied in the book, encompassing various features described above (and others, such as the kernel of a logic programming language). For each language, an operational semantics is given, along with a denotational semantics based on topological spaces. The relationship between each semantics is investigated. Here are the chapter titles, to give an idea of the breakdown of content: 1. Recursion and Iteration, 2. Nondeterminacy, 3. Variations, 4. Uniform Parallelism, 5. Unbounded Nondeterminism, 6. Locality, 7. Nonuniform parallelism, 8. Recursion Revisited, 9. Nested Resumptions, 10., Domain Equations and Bisimulation, 11. Branching Domains at Work, 12. Extensions of Nonuniform Parallelism, 13. Concurrent Object-oriented Programming, 14. Atomization, Commit, and Action Refinement, 15. The Control Flow Kernel of Logic Programming, 16. True Concurrency, 17. Full Abstractness, 18. Second-order Assignment.

In the next section, I summarize the first chapter of the book, to give a feel for the approach.

Overview of topological semantics

Given $\mathcal{L}$ a collection of programs in a language, a *semantics* for $\mathcal{L}$ is a mapping $\mathcal{M} : \mathcal{L} \to \mathcal{P}$ taking a program p to an element $\mathcal{M}(p)$ from a domain of meanings $\mathcal{P}$. The domain $\mathcal{P}$ should have enough mathematical structure to capture what we want to model. The study of semantics centers around the development of methods to specify $\mathcal{M}$ and associated $\mathcal{P}$ for a range of languages $\mathcal{L}$. We can distinguish essentially two ways of specifying $\mathcal{M}$:

Operational $\mathcal{O} : \mathcal{L} \to \mathcal{P}_O$, which captures the operational intuition about programs by using a transition system (axioms and rules) describing the actions of an abstract machine. This is the structural approach to operational semantics (SOS) advocated by Plotkin [4].

Denotational $\mathcal{D} : \mathcal{L} \to \mathcal{P}_D$, which is *compositional*; the meaning of a composite program is given by the meaning of its parts. This is helpful to derive program logics, to reason about correctness, termination and equivalence. Also, in general, denotational semantics are less "sensitive" to changes in the presentation of a language.

Consider the following simple example, to highlight the difference between the two styles of semantics. Let A be an alphabet, and W the set of structured words over A, given by the following BNF grammar:

$$w ::= a \mid (w_1 \cdot w_2)$$

where a is an identifier ranging over the elements of A. If $A = \{a, b, c\}$, then $(a \cdot (b \cdot a))$, $((a \cdot b) \cdot (c \cdot b))$, $(((a \cdot b) \cdot a) \cdot b)$ are structured words over A. We choose to assign, as the meaning of an element of W, its *length*. We derive both an operational and a denotational semantics to assign such a meaning to elements of W. We take $\mathcal{P}_O = \mathcal{P}_D = \mathbb{N}$ (where $\mathbb{N}$ is the set of natural numbers). To define the operational semantics, we consider the slightly extended language $V = W \cup \{E\}$, where intuitively E stands for the empty word. We define a transition system with transitions of the

form $(v, n) \longrightarrow (v', n')$ where $v, v' \in V$ and $n, n' \in \mathbb{N}$. (Such a transition "counts" one letter of the word v.) Let $\longrightarrow$ be the least relation satisfying the following inference rules:

$$\frac{}{(a, n) \longrightarrow (E, n+1)}$$

$$\frac{(v_1, n) \longrightarrow (v_1', n')}{((v_1 \cdot v_2), n) \longrightarrow ((v_1' \cdot v_2), n')}$$

$$\frac{(v_1, n) \longrightarrow (E, n')}{((v_1 \cdot v_2), n) \longrightarrow (v_2, n')}$$

We can define the operational semantics $\mathcal{O}$ by:

$$\mathcal{O}(w) = n \quad \text{if and only if} \quad (w, 0) \longrightarrow (v_1, 1) \longrightarrow \cdots \longrightarrow (E, n)$$

The denotational semantics $\mathcal{D}$ is much easier to define:

$$\begin{aligned} \mathcal{D}(a) &= 1 \\ \mathcal{D}(w_1 \cdot w_2) &= \mathcal{D}(w_1) + \mathcal{D}(w_2) \end{aligned}$$

It is straightforward to show, by structural induction, that in this case the operational and denotational semantics agree (that is, they give the same result for every word $w \in W$).

Let us now turn to a somewhat more realistic example. Recall that there are two kinds of imperative languages we consider, uniform and nonuniform. Let's define a simple uniform language with a recursive operator. This example is taken straight from Chapter 1 of the book. The language, $\mathcal{L}_{rec}$, is defined over an alphabet A of primitive actions. We assume a set of program variables $PVar$.

$$\begin{aligned} (Stat) \quad s &::= a \mid x \mid (s_1; s_2) \\ (GStat) \quad g &::= a \mid (g; s) \end{aligned}$$

A *statement* s is simply a sequence of actions; variables are bound to *guarded statements* g, which are simply statements that are forced to initially perform an action. When a variable is encountered during execution, the corresponding guarded statement is executed. A declaration is a binding of variables to guarded statements, and the space of all declarations is defined as $Decl = PVar \to GStat$. The language $\mathcal{L}_{rec}$ is defined as $\mathcal{L}_{rec} = Decl \times Stat$. We write an element of $\mathcal{L}_{rec}$ as $(x_1 \Leftarrow g_1, \ldots, x_n \Leftarrow g_n \mid s)$, representing the statement s in a context where $x_1, \ldots, x_n$ are bound to $g_1, \ldots, g_n$, respectively.

The operational semantics is defined by a transition system over $Decl \times Res$, where $Res = Stat \cup \{E\}$; the intuition is that E denotes a statement that has finished executing. We notationally identify the sequence $E; s$ with the statement s. This will simplify the presentation of the reduction rules. The transitions of the system take the form $s \xrightarrow{a}_D r$ where $s \in Stat$, $r \in Res$, $a \in A$, and $D \in Decl$; this transition should be interpreted as the program statement s rewriting into the statement r, along with a computational effect a. (For simplicity the computational effect is taken to be the action performed.) Again, the $\xrightarrow{a}_D$ relation is the least relation satisfying the following inference rules:

$$\frac{}{a \xrightarrow{a}_D E}$$

$$\frac{g \xrightarrow{a}_D r}{x \xrightarrow{a}_D r} \qquad \text{if } D(x) = g$$

$$\frac{s_1 \xrightarrow{a}_D r_1}{s_1; s_2 \xrightarrow{a}_D r_1; s_2}$$

We take the domain $\mathcal{P}_O$ of operational meanings to be the set of finite and infinite sequences of actions, $\mathcal{P}_O = A^\infty = A^* \cup A^\omega$. We define the operational semantics $\mathcal{O} : Decl \times Res \to \mathcal{P}_O$ as:

$$\mathcal{O}(D \mid r) = \begin{cases} a_1 a_2 \cdots a_n & \text{if } r \xrightarrow{a_1}_D r_1 \xrightarrow{a_2}_D \cdots \xrightarrow{a_n}_D r_n = E \\ a_1 a_2 \cdots & \text{if } r \xrightarrow{a_1}_D r_1 \xrightarrow{a_2}_D \cdots \end{cases}$$

For instance, we have $\mathcal{O}(D \mid a_1; (a_2; a_3)) = a_1 a_2 a_3$, and $\mathcal{O}(x \Leftarrow (a; y), y \Leftarrow (b; x) \mid x) = (ab)^\omega$.

Deriving a denotational semantics is slightly more complicated. A program in $\mathcal{L}_{rec}$ may describe infinite computations. To make sense of those, we need the notion of the limit of a computation. In mathematical analysis, limits are usually studied in the context of metric spaces [5]. This is the setting in which we will derive our semantics.

A *metric space* is a pair (M, d) with M a nonempty set and $d : M \times M \to \mathbb{R}_{\geq 0}$ (where $\mathbb{R}_{\geq 0}$ is the set of nonnegative real numbers) satisfying: $d(x, y) = 0$ iff $x = y$, $d(x, y) = d(y, x)$, and $d(x, y) \leq d(x, z) + d(z, y)$. A metric space (M, d) is α-*bounded* (for $\alpha < \infty$) if $d(x, y) \leq \alpha$ for all x and y in M.

We can define a metric on A^∞ as follows. For any $w \in A^\infty$, let $w[n]$ be the prefix of w of length at most n. The Baire-distance metric $d_B : A^\infty \times A^\infty \to \mathbb{R}_{\geq 0}$ is defined by

$$d_B = \begin{cases} 0 & \text{if } v = w \\ 2^{-n} & \text{if } v \neq w \text{ and } n = \max\{k \ : \ v[k] = w[k]\} \end{cases}$$

We say a sequence $\{x_n\}_{n=1}^\infty$ is *Cauchy* if for all $\epsilon > 0$ there exists an i such that for all $j, k \geq i$, $d(x_j, x_k) \leq \epsilon$. In other words, the elements of a Cauchy sequence get arbitrary close with respect to the metric. A metric space (M, d) is *complete* if every Cauchy sequence converges in M. It is easy to check that the metric space (A^∞, d_B) is complete.

If (M, d) is α-bounded for some α, and X is any set, let $(X \to M, d_F)$ be the *function space* metric space defined as follows: $X \to M$ is the set of all functions from X to M, and $d_F(f, g) = \sup\{d(f(x), g(x)) \ : \ x \in X\}$ (α-boundedness on M guarantees that this is well-defined). One can check that if (M, d) is complete, then so is $(X \to M, d_F)$.

A central theorem of the theory of metric spaces, which is used heavily in the book, is Banach's fixed point theorem. Essentially, this theorem says that every function f from a metric space to itself that decreases the distance between any two points must have a fixed point (a point x such that $f(x) = x$). We need more definitions to make this precise. Define a function $f : (M_1, d_1) \to (M_2, d_2)$ to be contractive if there exists an α between 0 and 1 such that $d_2(f(x), f(y)) \leq \alpha d_1(x, y)$. For example, the function $f : (A^\infty, d_B) \to (A^\infty, d_B)$ defined by $f(x) = a \cdot x$ is $\frac{1}{2}$-contractive.

Theorem (Banach): Let (M, d) be a complete metric space, $f : (M, d) \to (M, d)$ a contractive function. Then

1. there exists an x in M such that $f(x) = x$,
2. this x is unique (written $\mathit{fix}(f)$), and
3. $\mathit{fix}(f) = \lim f^n(x_0)$ for an arbitrary $x_0 \in M$, where $f^{n+1}(x_0) = f(f^n(x_0))$.

This is the basic metric space machinery needed to get a simple denotational semantics going. Returning to our sample language $\mathcal{L}_{rec}$, we take as a target of our denotational semantics the domain $\mathcal{P}_D = A^\infty - \{\epsilon\}$. (We do not allow the empty string for technical reasons. Notice that the empty string cannot be expressed by the language in any case.) What we want is a function $\mathcal{D}$ defined as follows:

$$\begin{aligned} \mathcal{D}(D \mid a) &= a \\ \mathcal{D}(D \mid x) &= \mathcal{D}(D \mid D(x)) \\ \mathcal{D}(D \mid s_1; s_2) &= \mathbf{;}(\mathcal{D}(D \mid s_1), \mathcal{D}(D \mid s_2)) \end{aligned}$$

(for some function **;** defined over A^∞, meant to represent sequential composition, and to be defined shortly.) Notice that this definition of $\mathcal{D}$ is not inductive. We will use Banach's theorem to define the function **;** over A^∞, and to define the function $\mathcal{D}$.

Let us concentrate on **;**. Intuitively, we want **;** to be a function $A^\infty \times A^\infty \to A^\infty$ satisfying

$$\begin{aligned} \mathbf{;}(a, p) &= a \cdot p \\ \mathbf{;}(a \cdot p', p) &= a \cdot \mathbf{;}(p', p) \end{aligned}$$

Note that the above properties do not form an inductive definition of **;** due to the presence of infinite words in A^∞. Instead, we will define **;** as the fixed point of the appropriate higher-order operator. Let $Op = A^\infty \times A^\infty \to A^\infty$ be the complete metric space of functions.[2] Define the following operator $\Omega_{;} : Op \to Op$:

$$\begin{aligned} \Omega_{;}(\phi)(a, p) &= a \cdot p \\ \Omega_{;}(\phi)(a \cdot p', p) &= a \cdot \phi(p', p) \end{aligned}$$

Note that the above equations *do* define a function $\Omega_{;}$. One can check that $\Omega_{;}$ is in fact a $\frac{1}{2}$-contractive mapping from Op to Op. Therefore, by Banach's theorem, there exists a unique fixed point (call it **;**) such that $\Omega_{;}(\mathbf{;}) = \mathbf{;}$. It is easy to see that this **;** satisfies the original equations we were aiming for.

Now that we have such a function **;**, let us turn to the problem of actually defining $\mathcal{D}$. We proceed similarly, by defining $\mathcal{D}$ as the fixed point of the appropriate higher-order operator, through an application of Banach's theorem. Consider the metric space $Sem_D = \mathcal{L}_{rec} \to A^\infty$, which is complete since A^∞ is complete. Define the following function $\Psi : Sem_D \to Sem_D$ by:

$$\begin{aligned} \Psi(S)(D \mid a) &= a \\ \Psi(S)(D \mid x) &= \Psi(S)(D \mid D(x)) \\ \Psi(S)(D \mid s_1; s_2) &= \mathbf{;}(\Psi(S)(D \mid s_1), S(D \mid s_2)) \end{aligned}$$

(There is some subtlety in coming up with the last equation; as you'll notice from looking at the righthand side, there is recursion over Ψ in only one of the two cases. The book explains this.) Once again, we can show that Ψ is a $\frac{1}{2}$-contractive function (in S), and thus by Banach's theorem there is a unique fixed point of Ψ (call if $\mathcal{D}$) such that $\Psi(\mathcal{D}) = \mathcal{D}$. It is straightforward to check that this $\mathcal{D}$ satisfies our requirements for the denotational semantics function.

A final result of interest after all of these developments is the relationship between $\mathcal{O}$, the operational semantics based on an intuitive notion of computation, and $\mathcal{D}$, the denotational semantics with its compositional properties. It turns out that in this case, $\mathcal{O} = \mathcal{D}$, and moreover this result can be derived from a third application of Banach's theorem. The details can be found in the book.

Opinion

As a technical book, aimed at describing an approach to provide semantics to a wide variety of imperative language control flow structures, this book is complete. All the examples are worked out with enough details to grasp the subtleties arising. Let the reader be warned, however, that the book is dense—both in terms of the technical material, and in terms of the presentation. The first few chapters should be read slowly and with pencil in hand.

The book does not require as much background knowledge of topology as one may expect. Prior exposure is of course beneficial, but in fact, only the basics of topology and metric spaces are actually used, and whatever is needed is presented in the first few chapters. On the other hand, the presentation does assume what may best be called mathematical maturity.

In the grand scheme of things, a problem with this book is one of motivation and followup. This is hardly new in the field of semantics. Specifically, the use of denotational semantics is hardly motivated, considering that most of the machinery in the book is aimed at coming up with denotational semantics and showing that it agrees with the intuitive operational semantics. There is a throw-away line about the fact that denotational semantics can help in developing logics for reasoning about programs, but most of the interesting developments are buried in the bibliographical notes at the end of the chapters. This will not deter the hardcore semanticist, but may have other readers go: "so what?". Sad, since denotational semantics *is* useful.

And for the curious, Franck's actual work can be found in [7].

[2] Strictly speaking, we need to consider the space of bounded functions to ensure that the space is complete. This will be irrelevant at our level of discussion.

References

[1] D. Harel, D. Kozen, and J. Tiuryn. *Dynamic Logic*. The MIT Press, Cambridge, Massachusetts, 2000.

[2] C. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[3] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[4] G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, University of Aarhus, 1981.

[5] W. Rudin. *Principles of Mathematical Analysis*. McGraw-Hill, third edition, 1976.

[6] D. S. Scott and C. Strachey. Toward a mathematical semantics for computer languages. In J. Fox, editor, *Proceedings of the Symposium on Computers and Automata*, New York, 1971. Polytechnic Institute of Brooklyn Press.

[7] F. van Breugel. *Comparative Metric Semantics of Programming Languages: Nondeterminism and Recursion*. Birkhäuser, 1998.