A Case Study of Operational and Denotational Semantics

Maja Frydrychowicz

23/04/2007

Abstract

We examine the concepts upon which program evaluation and domain theory are based by studying the semantics of a small programming language. In particular, we discuss the properties of Scott domains and their role in describing principles such as recursive programming via fixed points, computation as finite approximation, as well as, the importance of compositionality in denotational semantics. The definition of a correspondence between these two semantics styles is then explored.

Introduction

Operational semantics describe evaluation strategies: the procedures by which a programming language arrives at its output. Denotational semantics abstract away from step-by-step evaluation, and instead provide means to establish general properties of a programming language. The field of domain theory thus relates orders and topological spaces to the seemingly far-removed study of programming language behaviour and design. The stark contrast in approach might lead one to think that operational and denotational semantics have very little in common. However, the relationship between these two semantics styles is a rich and well-studied topic. As an introduction to these ideas, we define operational and denotational semantics for a small programming language, and systematically review the principles behind the latter approach. This allows us to then consider what one might need to show in order to conclude that the two methods are in fact in agreement.

Simple Applicative Language

We restrict our study of semantics to the Simple Applicative Language (SAL). As its name indicates, its syntax is not very elaborate. However, the constructs it provides are sufficient for the study of the core concepts at hand. For convenience, we assume that basic boolean logic and arithmetic are given, and we describe SAL's constructs in these terms. This additional simplification does not diminish the relevance of this case study, since the keys ideas to consider are directly related to abstraction and recursion, as we will soon demonstrate.

SAL is equipped with abstractions, pairs, basic arithmetic, conditionals and most importantly, recursion via the Fix operator. Throughout the remaining sections of this document, $\mathsf{L},\mathsf{M},\mathsf{N}$ refer to general terms, x, y, z, to variables, σ, τ, to types, and n, to values associated with natural numbers, including 0.

$$\frac{}{\Gamma, x : \tau \vdash x : \tau}\ hyp$$

$$\frac{}{\Gamma \vdash \mathsf{n} : \mathsf{Nat}} \quad \forall \mathsf{n} \in \mathsf{Naturals}$$

$$\frac{}{\Gamma \vdash \mathsf{false} : \mathsf{Bool}}$$

$$\frac{}{\Gamma \vdash \mathsf{true} : \mathsf{Bool}}$$

$$\frac{\Gamma, x : \tau_1 \vdash \mathsf{M} : \tau_2}{\Gamma \vdash \lambda x : \tau_1.\ \mathsf{M} : \tau_1 \to \tau_2}\ lam$$

$$\frac{\Gamma \vdash \mathsf{M} : \mathsf{Nat} \quad \Gamma \vdash \mathsf{N} : \mathsf{Nat}}{\Gamma \vdash \mathsf{M\ plus\ N} : \mathsf{Nat}}\ plus$$

$$\frac{\Gamma \vdash \mathsf{M} : \tau_1 \to \tau_2 \quad \Gamma \vdash \mathsf{N} : \tau_1}{\Gamma \vdash \ \mathsf{M\ N}\ : \tau_2}\ app$$

$$\frac{\Gamma \vdash \mathsf{M} : \tau_1 \times \tau_2}{\Gamma \vdash \pi_i(\mathsf{M}) : \tau_i}\ proj_i \qquad i \in \{1, 2\}$$

$$\frac{\Gamma \vdash \mathsf{M} : \tau_1 \quad \Gamma \vdash \mathsf{N} : \tau_2}{\Gamma \vdash \langle \mathsf{M},\ \mathsf{N} \rangle : \tau_1 \times \tau_2}\ pair$$

$$\frac{\Gamma \vdash \mathsf{M} : \mathsf{Nat} \quad \Gamma \vdash \mathsf{N} : \mathsf{Nat}}{\Gamma \vdash \mathsf{M\ minus\ N} : \mathsf{Nat}}\ minus$$

$$\frac{\Gamma \vdash \mathsf{M}_1 : \mathsf{Bool} \quad \Gamma \vdash \mathsf{M}_2 : \tau \quad \Gamma \vdash \mathsf{M}_3 : \tau}{\Gamma \vdash \ \mathsf{if\ M_1\ then\ M_2\ else\ M_3} : \tau}\ cond$$

$$\frac{\Gamma \vdash \mathsf{M} : \tau \to \tau}{\Gamma \vdash \ \mathsf{Fix\ \ M}\ : \tau}\ fix$$

Figure 1: SAL Type System.

$$\begin{array}{lll} \text{Types} & \tau ::= & \mathsf{Bool} \mid \mathsf{Nat} \mid \tau \to \tau \mid \tau \times \tau \\ \text{Terms} & \mathsf{M} ::= & x \mid \lambda x : \tau.\ \mathsf{M} \mid \mathsf{M\ M} \mid \langle \mathsf{M},\ \mathsf{M} \rangle \mid \pi_i(\mathsf{M}) \mid \mathsf{Fix\ \ M} \mid \mathsf{M\ plus\ M} \mid \mathsf{M\ minus\ M} \\ & & \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{if\ M\ then\ M\ else\ M} \mid \mathsf{n} \end{array}$$

$$\begin{array}{r} \mathsf{n} \in \mathsf{Naturals} \\ x \in \mathsf{Variables} \\ i \in \{1, 2\} \end{array}$$

The type inference rules presented in Figure 1 are defined in the usual manner. Γ denotes the context of assumptions in the type derivation. The discussion of free variables and context substition is omitted.

SAL's type system is an essential part of our discussion of denotational semantics, and will guide us through the process of defining appropriate semantic maps. ‘

Operational Semantics

We describe the operational semantics (Figure 2) of SAL by using inference rules to define a binary relation "$\longrightarrow$" on terms. $\mathsf{M} \longrightarrow \mathsf{M'}$ represents a single evaluation step from term M. The constant terms $\mathsf{true}, \mathsf{false}$ and $\mathsf{n} : \mathsf{Nat}$ are regarded as values, and conclude all evaluations. Since the terms n are restricted to the natural numbers, 0 minus n is defined to evaluate to 0 for all n.

Furthermore, it should be noted that the evaluation of conditional statements is restricted to a specific order: the boolean "guard" is reduced to a constant term first, afterwhich the appropriate case is evaluated. The operator Fix is used to define recursive functions: when a function Fix M of type $\sigma \to \sigma$ is applied to an argument, it replicates itself and that argument inside M. Thus, the operational semantics express that the argument to the Fix operator is reduced step by step until it reaches the form of an abstraction, at which point a substitution is performed according to the usual definition of fixed point.

$$\lambda x : \tau.\ \mathsf{M}\quad \mathsf{N} \longrightarrow [\mathsf{N}/x]\mathsf{M} \qquad \frac{\mathsf{M} \longrightarrow \mathsf{M'}}{\text{if M then } \mathsf{N}_1 \text{ else } \mathsf{N}_2 \longrightarrow \text{ if M' then } \mathsf{N}_1 \text{ else } \mathsf{N}_2}$$

$$0 \text{ minus n} \longrightarrow 0$$

$$\frac{\mathsf{M} \longrightarrow \mathsf{M'}}{\mathsf{M} \text{ plus } \mathsf{N} \longrightarrow \mathsf{M'} \text{ plus } \mathsf{N}} \qquad \text{if true then M else N} \longrightarrow \mathsf{M}$$

$$\text{if false then M else N} \longrightarrow \mathsf{N} \qquad \frac{\mathsf{M} \longrightarrow \mathsf{M'}}{\mathsf{M} \text{ minus } \mathsf{N} \longrightarrow \mathsf{M'} \text{ minus } \mathsf{N}}$$

$$\pi_i(\langle \mathsf{M}_1,\ \mathsf{M}_2 \rangle) \longrightarrow \mathsf{M}_i \qquad \frac{\mathsf{N} \longrightarrow \mathsf{N'}}{\mathsf{M} \text{ plus } \mathsf{N} \longrightarrow \mathsf{M} \text{ plus } \mathsf{N'}}$$

$$\mathsf{n}_1 \text{ plus } \mathsf{n}_2 \longrightarrow n_1 + n_2$$

$$\frac{\mathsf{N} \longrightarrow \mathsf{N'}}{\mathsf{M} \text{ minus } \mathsf{N} \longrightarrow \mathsf{M} \text{ minus } \mathsf{N'}}$$

$$\mathsf{n}_1 \text{ minus } \mathsf{n}_2 \longrightarrow n_1 - n_2$$

$$\frac{\mathsf{M} \longrightarrow \mathsf{M'}}{\text{Fix M} \longrightarrow \text{ Fix M'}} \qquad \frac{\mathsf{M} \longrightarrow \mathsf{M'}}{\pi_i(\mathsf{M}) \longrightarrow \pi_i(\mathsf{M'})}$$

$$\frac{\mathsf{M} \longrightarrow \mathsf{M'}}{\mathsf{M}\ \mathsf{N} \longrightarrow \mathsf{M'}\ \mathsf{N}} \qquad \frac{\mathsf{M} \longrightarrow \mathsf{M'}}{\langle \mathsf{M},\ \mathsf{N} \rangle \longrightarrow \langle \mathsf{M'},\ \mathsf{N} \rangle}$$

$$\text{Fix } \lambda x : \tau.\ \mathsf{M} \longrightarrow [\ \text{Fix } \lambda x : \tau.\ \mathsf{M}\ /x]\mathsf{M} \qquad \frac{\mathsf{N} \longrightarrow \mathsf{N'}}{\langle \mathsf{M},\ \mathsf{N} \rangle \longrightarrow \langle \mathsf{M},\ \mathsf{N'} \rangle}$$

Figure 2: SAL Operational Semantics.

Denotational Semantics

Denotational semantics describe the meaning of a program in terms of mappings from types and terms to mathematical objects. We call these objects *denotations*. The goal is to capture the notion of computation with these mappings. We will study SAL with semantic maps based on Scott domains. These are endowed with many useful properties. In particular, the set of

all continuous functions on Scott domains forms a Scott domain, as does the product of Scott domains.

Consider the following model. Let Scott domains represent types. Suppose the present computation is always associated with some *environment* ρ, which maps variable names to elements of the appropriate Scott domain, according to the variable's type. We observe that environment is reminiscent of Γ, the assumption context in the inference rules for typing SAL. Given a term and an environment, we want to produce an element from the appropriate Scott domain.

$$\begin{aligned} [\![\cdot]\!] &: \mathsf{Type} \rightarrow \mathsf{Scott\ Domains} \\ [\![\cdot]\!] &: \mathsf{Term} \rightarrow \mathsf{Environment} \rightarrow \mathsf{Element\ of\ Scott\ Domain} \\ \mathsf{Environment}\ \rho &: \mathsf{Variable} \rightarrow \mathsf{Element\ of\ Scott\ Domain} \end{aligned}$$

Note that we are overloading our notation $[\![\cdot]\!]$ for maps on terms and maps on types.

Suppose we are given the following explicit definitions for the type maps in our model. The domains representing Bool and Nat are illustrated in Figure 3.

$$\begin{aligned} [\![\mathsf{Bool}]\!] &= \mathbb{B}_\perp \\ [\![\mathsf{Nat}]\!] &= \mathbb{N}_\perp \\ [\![\sigma \rightarrow \tau]\!] &= [[\![\sigma]\!] \rightarrow [\![\tau]\!]] \text{ Set of Scott-continuous functions.} \\ [\![\sigma \times \tau]\!] &= [\![\sigma]\!] \times [\![\tau]\!] \text{ Product of Scott domains} \end{aligned}$$

The partial order imposed by these domains qualifies each element's information content. For domain elements x, y, if $x \leq y$ then y can be thought to contain more information than x. Furthermore, the information associated with y is consistent with that of x. It is therefore easy to see that the bottom element of each domain represents a term with no information content, or a term whose evaluation diverges.

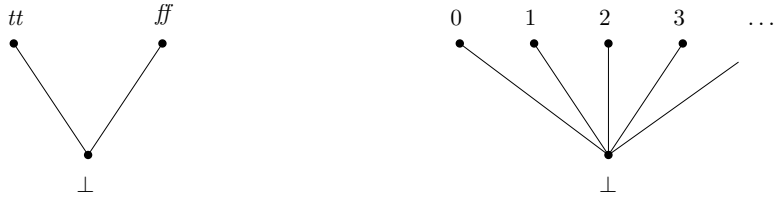

Figure 3: $\mathbb{B}_\perp$ and $\mathbb{N}_\perp$

Given the above setting, how should we define the corresponding term maps in order to accurately denote the meaning of a term? One reasonbale intuition, inspired by the resemblance of the environment ρ to SAL's typing assumption context Γ, we attempt to model the required term maps after SAL's type inference rules.

We begin with the simplest cases. Applying a term map to any constant term $\mathsf{n} : \mathsf{Nat}$ and some environment ρ should yield the element corresponding to n in the domain $[\![\mathsf{Nat}]\!]$, since no further computation can narrow n down to a more precise value. Also, the environment ρ has no impact on the result, since no variable names are involved in term n. The same holds for terms false and true.

$$[\![\mathsf{true} : \mathsf{Bool}]\!]\rho = tt \in [\![\mathsf{Bool}]\!]$$
$$[\![\mathsf{false} : \mathsf{Bool}]\!]\rho = \mathit{ff} \in [\![\mathsf{Bool}]\!]$$
$$[\![\mathsf{n} : \mathsf{Nat}]\!]\rho = n \in [\![\mathsf{Nat}]\!]$$

As for typing judgements in SAL's type system, the meaning associated with the above terms, depends only on the terms, and not on any other information.

Next, we consider arithmetic operations. SAL restricts the types of the subterms of plus and minus to Nat. It also describes a dependence of this term's type on the type of each of its subterms. The definition below expresses this same dependence. As in our description of SAL's operational semantics, we express the meaning of plus and minus in terms of basic arithmetic operations, which we assume to be given. Note that, it is also possible that one of M,N is not well-typed, or never converges to a value, in which case it is mapped to $\bot$. The parent term should therefore also be mapped to $\bot$.

$$[\![\mathsf{M}\ \mathsf{plus}\ \mathsf{N}]\!]\rho = \begin{cases} \bot & \text{if } [\![\mathsf{M}]\!]\rho = \bot \text{ or } [\![\mathsf{N}]\!]\rho = \bot \\ ([\![\mathsf{M}]\!]\rho + [\![\mathsf{N}]\!]\rho) \in [\![\mathsf{Nat}]\!] & \text{otherwise} \end{cases}$$

$$[\![\mathsf{M}\ \mathsf{minus}\ \mathsf{N}]\!]\rho = \begin{cases} \bot & \text{if } [\![\mathsf{M}]\!]\rho = \bot \text{ or } [\![\mathsf{N}]\!]\rho = \bot \\ ([\![\mathsf{M}]\!]\rho - [\![\mathsf{N}]\!]\rho) \in [\![\mathsf{Nat}]\!] & \text{otherwise} \end{cases}$$

where M,N are of type Nat

The semantic maps for pairs, projections and conditionals exhibit similar behaviour, in the sense that the result of the mapping depends exclusively on subterms.

$$[\![\langle \mathsf{M},\ \mathsf{N}\rangle]\!]\rho = \begin{cases} \bot & \text{if } [\![\mathsf{M}]\!]\rho = \bot \text{ or } [\![\mathsf{N}]\!]\rho = \bot \\ ([\![\mathsf{M}]\!]\rho, [\![\mathsf{N}]\!]\rho) \in [\![\tau_1]\!] \times [\![\tau_2]\!] & \text{otherwise} \end{cases}$$

where $\mathsf{M} : \tau_1, \mathsf{N} : \tau_2$

$$[\![\pi_i(\langle \mathsf{M}_1,\ \mathsf{M}_2\rangle)]\!]\rho = [\![\mathsf{M}_i]\!]\rho \in [\![\tau_i]\!]$$

where $\langle \mathsf{M}_1,\ \mathsf{M}_2\rangle : \tau_1 \times \tau_2$

$$[\![\ \mathsf{if}\ \mathsf{L}\ \mathsf{then}\ \mathsf{M}\ \mathsf{else}\ \mathsf{N}\ ]\!]\rho = \begin{cases} \bot & \text{if } [\![\mathsf{L}]\!]\rho = \bot \\ [\![\mathsf{M}]\!]\rho & \text{if } [\![\mathsf{L}]\!]\rho = tt \in \mathbb{B}_\bot \\ [\![\mathsf{N}]\!]\rho & \text{if } [\![\mathsf{L}]\!]\rho = \mathit{ff} \in \mathbb{B}_\bot \end{cases}$$

So far, the environment ρ has not had any influence on the outputs of our semantic maps. We now examine the terms that explicitly involve variables and employ substitutions. The upcoming denotations are more intricate, although they still match SAL's type inference rules.

The semantic map for variables is straightforward: the variable x is mapped to a domain that reprepresents its type. It is mapped to a specific element in that domain, as indicated by the current environment ρ, which may of course be a partial function:

$$[\![x : \tau]\!]\rho = \rho(x) \in [\![\tau]\!]$$

The semantic map for abstraction is of particular interest. Consider an abstraction with formal parameter $x : \sigma$ and body $\mathsf{M} : \tau$. The semantic map $[\![\cdot]\!]$ applied to this abstraction and a given environment ρ yields a function $f : [\![\sigma]\!] \to [\![\tau]\!]$. For any $d \in [\![\sigma]\!]$, $f(d)$ updates the current environment ρ according to d, and then applies a semantic map to the body M with the *new* environment $\rho' = \rho[x \mapsto d]$. Thus $\rho'(x) = d$, but is otherwise identical to ρ.

$$[\![\lambda x : \sigma.\ \mathsf{M}\ : \sigma \to \tau]\!]\rho = f \in [[\![\sigma]\!] \to [\![\tau]\!]]$$

$$\text{where } \mathsf{M} : \tau \text{ and } \forall d \in [\![\sigma]\!]\ f(d) = [\![\mathsf{M}]\!]\rho[x \mapsto d] \in [\![\tau]\!]$$

To confirm that this definition is correct, we must show that f is indeed a Scott-continuous function as required by the type map provided:

1. Is f monotone?
2. Does f preserve directed suprema?

One approach is to verify the desired property case by case, for all possible terms M. Of course, there are infinitely many possible terms; however, the semantic map definitions we've examined so far indicate that term denotations in this model are constructed entirely with the denotations of subterms. This is a property we should watch for in the upcoming term map definitions for application and the Fix operator. If this constructive property does hold, it is reasonable to attempt a proof by induction on the structure of terms. The likely difficulties one would encounter in such a proof is with establishing the continuity of the domain functions used in recursion, and abstraction, since they modify their environments.

The semantic map for applications is closely related to that for abstractions. In essence, we are applying the function $f = [\![\mathsf{M} : \sigma \to \tau]\!] \in [[\![\sigma]\!] \to [\![\tau]\!]]$ described earlier.

$$[\![\mathsf{M}(\mathsf{N}) : \tau]\!]\rho = \overbrace{[\![\mathsf{M} : \sigma \to \tau]\!]\rho(\underbrace{[\![\mathsf{M} : \sigma]\!]\rho}_{\in [\![\sigma]\!]})}^{\in [\![\tau]\!]}$$

Thus $[\![\mathsf{M}(\mathsf{N}) : \tau]\!]\rho = f([\![\mathsf{N} : \sigma]\!]\rho) = [\![L]\!]\rho[x \to [\![N : \sigma]\!]\rho]$ where M is of the form $\lambda x : \sigma.L$. Yet again, this fits nicely with SAL's type system, and is expressed using subterms, as desired.

Finally, we arrive at the mythical Fix operator, which is meant to represent the computational fixed point of an abstraction. Our goal is to semantically capture the behaviour that $\mathsf{Fix}\ \mathsf{M}$ is a fixed point of the abstraction M. That is, $\mathsf{Fix}\ \mathsf{M} = \mathsf{M}\ \mathsf{Fix}\ \mathsf{M}$. In addition, we would like to define the semantics of $\mathsf{Fix}\ \mathsf{M}$ exclusively in terms of its subterm M. Let *fix* g denote *any* fixed point of a function g. Note that *fix* g may be undefined for some inputs. We propose the following map for $\mathsf{Fix}\ \mathsf{M}$:

$$[\![\ \mathsf{Fix}\ \mathsf{M} : \tau]\!]\rho = \mathit{fix}([\![\mathsf{M}]\!]\rho)$$

where M is of the form $\lambda x : \tau.\mathsf{N}$

For this semantic map definition to be accurate, we need to establish that

1. A fixed point of M exists.
2. $\mathsf{Fix\ M}$ is indeed a fixed point of M.

As dictated by the established denotation for abstraction, $[\![\mathsf{M}]\!]\rho$ is in fact a function, call it f, of type $[\![\tau]\!] \to [\![\tau]\!]$. Assuming that f is indeed continuous (and thus also monotone) as required by the type map, it is guaranteed to have a fixed point (Scott's fixed point theorem). The denotation of a recursive function $\mathsf{Fix\ M}$ amounts to applying f repeatedly to $\perp \in [\![\tau]\!]$, which produces a chain $\perp \leq f(\perp) \leq f^2(\perp) \leq f^3(\perp) \leq \ldots$ in $[\![\tau]\!]$. Since $[\![\tau]\!]$ is a domain according to our model, this chain has a supremum $v \in [\![\tau]\!]$. By continuity and the properties of suprema, $f(v) = v$, so v corresponds to the fixed point of f. We may look at the elements of the above-mentioned chain as increasingly accurate approximations of the final value obtained by the recursive program represented by $\mathsf{Fix\ M}$.

The denotation for $\mathsf{Fix\ M}$ brings to light why the continuity of f is a key property of these semantic maps.

The Requirements of Semantic Correspondence

Establishing that two semantic styles for a given programming language agree leads to useful insights regarding program equivalence. The first step toward proving such a correspondence is to define what it means for semantic styles to coincide.

We've presented an evaluation relation $\longrightarrow$ on terms to describe the operational semantics of SAL, as well as, maps $[\![\cdot]\!]$ on terms and types to Scott domains that describe SAL's denotational semantics. Our present goal is to observe what properties of the relationship between $\longrightarrow$ and $[\![\cdot]\!]$ would indicate that to the two definitions of program semantics are in essence the same.

One aspect of correspondence between the two approaches concerns the equivalence of terms. If $[\![\mathsf{M}]\!] = [\![\mathsf{N}]\!] = x \in [\![\tau]\!]$, then there should exist a constant term v such that there are finite series of evaluation steps from M to v and from N to v in terms of $\longrightarrow$ and vice versa.

A stronger property to consider is that which restricts the results of evaluation. If a given term evaluates to v via $\longrightarrow$, that same term should be mapped with $[\![\cdot]\!]$ to the denotation of v - the domain element that represents v. The converse should also be true. Formally, we require

$$\mathsf{M} \longrightarrow \ldots \longrightarrow v \iff [\![\mathsf{M}]\!]\rho = [\![\mathsf{v}]\!]\rho'$$

While these properties are essential for a correspondence between $\longrightarrow$ and $[\![\cdot]\!]$, we cannot claim, based on our current knowledge, that these are the only requirements.

Conclusion

By developing the operational and denotational semantics of the Simple Applicative Language, we demonstrated the role Scott domains and continuous functions play in the denotation of recursive and non-recursive functions. An interesting connection between the programming language type inference rules and the definitions of semantic maps on terms was used to guide our completion of the denotational semantics. The modular way in which the term maps are defined indicates

that structural induction is available as a general proof strategy for properties of the model such as continuity of functions, perhaps. We then identified three properties that illustrate, at least in part, what it means for semantic styles to agree: essentially the term maps and evaluation relation defined must both associate terms with the same values.

References

[1] C. Gunter. *Semantics of Programming Languages: Structures and Techniques.* (Chapter 2), MIT Press, 1992.

[2] B. C. Pierce. *Types and Programming Languages.* MIT Press, 2002.