Composite Semantics: One Generic Environment is Enough

Jonathan Young

Research Report YALEU/DCS/RR-595
December 1987

This research was supported in part by National Science Foundation grant CCR-845145.

Composite Semantics: One Generic Environment is Enough

Jonathan Young

December 1987

Yale University
Department of Computer Science [1]

Abstract

A *composite semantics* is one in which several semantic analyses are combined into one; for example, strictness analysis combined with sharing analysis. While denotational semantics is a good framework in which to write a composite semantics, problems often arise when one analysis calls upon another analysis which uses a different environment domain. In this paper we discuss the deficiencies with existing techniques for solving this problem, and present a new solution using *generic environments*, unevaluated mappings of identifiers to syntactic objects. We show how to write a standard semantics using this technique, and how the value domain which results corresponds to the usual reflexive domain.

1 Introduction

Denotational semantics [10,3,9] has recently become one of the tools of choice for research in the formal description of programming languages because it enables the researcher to abstract away from "operational" or implementation issues and to concentrate instead on the more algebraic properties of

[1]This research was supported in part by National Science Foundation grant DCR-845145.

a language. Within the more formal and mathematical framework of denotational semantics, it is easy to apply formal techniques to reason about programs without reference to a particular implementation, to verify the correctness of different implementations, and to prove the safety of various program optimizations.

While denotational semantics is a useful tool for describing and understanding programmming language semantics, problems have arisen recently as researchers have tried to build on their previous successes. For example, an exact semantics which captures sharing properties depends on the standard semantics of the predicate in a conditional [2]. Variations on strictness analysis need to call the original strictness analysis [1]. It is desirable to prove corresponding properties between a new semantics and an old (or standard) semantics [6,8], but the two semantics have different types of environments. What do we do?

2 Previous Techniques

We begin with a criticism of some existing techniques and then present our new method for solving this problem. For simplicity we refer to the "new" analysis (or semantics) which wants to call the "old" analysis. (This is not to imply that the same problem does not arise when there are mutual dependencies between semantic analyses.)

The first technique is a simple appeal to the reader's intuition. Quoting from a recent paper,

> "When we use the semantic function **A** in the definition of **S** we assume that the environment *aenv* results from a computation of **A** for the same expression as the one for which the action of **S** is being defined." [8] [2]

While this is fine for a technical paper in which the details are intuitively clear from the context and do not contribute to the presentation, researchers who actually try to implement this correspondence quickly find that there is more to it than meets the eye. In particular, it is not sufficient to merely

[2] This citation is not meant to indicate that the paper in question is the only offender, since the author [6] and others have done the same. This paper is perhaps unique in admitting to doing so in such a direct manner.

track *aenv* as a function of the *expression* being evaluated. It is important to maintain the *context* of this expression – including the values of all bound variables.

A second technique involves asking an *oracle* for the meaning of an expression in the middle of an exact semantic analysis [7,4]. In particular, oracles are useful for choosing which arm of a conditional to evaluate. Although the oracle is understood to correspond to the standard semantics, the reader's intuition is again being appealed to; it is no easier to implement oracles than "corresponding environments."

Some researchers [2,5] have tried to avoid the difficulty of incompatible environments by computing the old semantics as part of their new interpretation. The new result domain is a tuple, one component of which contains the result under the old analysis. While this method is easy to implement (and debug!), it violates a fundamental precept of software engineering: it is not modular – small changes to one semantics may necessitate changing several others.

Of course, one solution to the problem of incompatible environments is to keep track of the environment for the old analysis in addition to the new environment. While this is a correct way to implement a new semantics, this method violates the modularity principle in two ways. First, it is not easy to modify the new semantics to call a *third* analysis... or a fourth... The second, and perhaps more difficult problem, is that the new analysis must understand how to create and maintain the environments for all of the old analyses which it uses. (It should be noted, of course, that some researchers [1,8] have managed to avoid the proliferation of environments described above by the simple expedient of using the same types of environments for several related analyses, but this technique does not appear to be generally applicable.)

In this paper, we introduce a new technique, *generic environments*, for solving this problem. We begin by introducing a simple, higher-order language (equivalent to the untyped lambda calculus) and presenting its "traditional" standard semantics. After some minor domain changes to allow generic environments, we present the new standard semantics and another semantics which calls the standard semantics using the generic environments. We also construct a correspondence between the new result domain and the old result domain.

3 The language

We use a language similar to the higher-order, untyped lambda calculus with constants.

$$
\begin{array}{rcll}
c & \in & Con & \text{constants} \\
f, x & \in & Id & \text{identifiers} \\
e & \in & Exp & \text{expressions defined by:} \\
 & & e ::= c \mid x \mid (e_1\ e_2) \mid (\lambda x.e) \mid eg & \\
 & & eg ::= \textbf{letrec}\ \ f_i = e_i\ \textbf{in}\ \ e &
\end{array}
$$

All functions take one argument (curried). For conciseness, we write $\lambda xyz.e$ for $\lambda x.\lambda y.\lambda z.e$. Some constants are (possibly higher-order) primitive functions, such as $+ = \lambda ab.$ "$a + b$" or $if = (\lambda pca.$ "if p then c else a").

4 "Traditional" Standard Semantics

A word on notation: $[\![e]\!]env$ is to be read as an abbreviation for the pair $\langle e, env\rangle$. While this is not standard, it is a minor variation on the most common practice. Our reason for using this variation will become apparent in section 5.

Traditionally, the standard semantics for this language would look like this:

Standard Semantic Domains

$$
\begin{array}{rcll}
Bas & = & Int + Bool + \ldots & \text{domain of basic values.} \\
D_{Std} & = & Bas + (D_{Std} \rightarrow D_{Std}) & \text{domain of denotable values.} \\
 & & & \\
Ep_{Std} & = & Exp \times Env_{Std} & \text{domain of exp, env pairs.} \\
Env_{Std} & = & Id \rightarrow D_{Std} & \text{domain of environments.}
\end{array}
$$

Standard Semantic Functions

$$
\begin{array}{rl}
\mathcal{K}_{\mathcal{E}_{Std}}: & Con \rightarrow D_{Std} \\
\mathcal{E}_{Std}: & Ep \rightarrow D_{Std}
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{E}_{Std}\ [\![c]\!]env & = & \mathcal{K}_{\mathcal{E}_{Std}}[\![c]\!] \\
\mathcal{E}_{Std}\ [\![x]\!]env & = & env[\![x]\!] \\
\mathcal{E}_{Std}\ [\![e_1\ e_2]\!]env & = & (\mathcal{E}_{Std}\ [\![e_1]\!]env)\ (\mathcal{E}_{Std}\ [\![e_2]\!]env) \\
\mathcal{E}_{Std}\ [\![\lambda x.e]\!]env & = & \lambda v.\ \mathcal{E}_{Std}\ [\![e]\!]env[v/x] \\
\mathcal{E}_{Std}\ [\![\textbf{letrec}\ \ f_i = e_i\ \textbf{in}\ \ e]\!]env & = & \mathcal{E}_{Std}\ [\![e]\!]env' \\
 & & \textbf{whererec}\ \ env' = env[\ \mathcal{E}_{Std}\ [\![e_i]\!]env'/f_i\ ]
\end{array}
$$

The following primitive functions might be used:

$$
\begin{array}{rcl}
\mathcal{K}_{\mathcal{E}_{Std}}[\![+]\!] & = & \lambda ab.\ a + b \\
\mathcal{K}_{\mathcal{E}_{Std}}[\![if]\!] & = & \lambda pca.\ \textbf{if}\ \ p\ \textbf{then}\ \ c\ \textbf{else}\ \ a
\end{array}
$$

5 New Standard Semantics

If we were to write a new semantics $\mathcal{S} : Ep \rightarrow D_{\mathcal{S}}$, the environment domain would probably be $Env_{\mathcal{S}} : Id \rightarrow D_{\mathcal{S}}$. But the environment domain for the standard semantics was $Env_{Std} = Id \rightarrow D_{Std}$. This creates a problem if $\mathcal{S}$ needs to call $\mathcal{E}$. What we desire is an *unevaluated* environment, containing objects which are evaluated later. A naïve attempt at this would map each bound identifier to the "code" (in Exp) to which it was bound. But the expression alone is not enough; we must also remember the environment in which the expression occurred. This pair of an expression and an environment corresponds exactly to elements of our Ep domain. Our new domains are:

New Standard Semantic Domains

$$
\begin{array}{rcll}
Ep & = & Exp \times Env & \text{domain of exp, env pairs} \\
Env & = & Id \rightarrow Ep, & \text{domain of environments.}
\end{array}
$$

It turns out that the standard value domain D_{Std} is not exactly what we want for our new standard semantics. Instead, we use another domain in which functions expect elements of Ep as arguments. It should be emphasized that this is purely to preserve the symmetry of the semantic equations; we will show in section 7 how the domains correspond. Thus:

$$D = Bas + (Ep \rightarrow D).$$

The domain changes induce only three changes in the semantic equations for $\mathcal{E}$: we must *evaluate* the value obtained when looking up a variable in the bound variable environment, and in two places (function application and equation groups) we must *not evaluate* the expression and environment pair before using it as an argument or putting it into the environment. The equation for function abstraction ($\lambda x.e$), in fact, requires no change, although the argument v changes type from D_{Std} to Ep.

New Standard Semantic Functions

$$\begin{aligned} \mathcal{K}_{\mathcal{E}} &: Con \to D \\ \mathcal{E} &: Ep \to D \end{aligned}$$

$$\begin{aligned} \mathcal{E}\ [\![c]\!]env &= \mathcal{K}_{\mathcal{E}}[\![c]\!] \\ \mathcal{E}\ [\![x]\!]env &= \mathcal{E}\ env[\![x]\!] \\ \mathcal{E}\ [\![e_1\ e_2]\!]env &= (\mathcal{E}\ [\![e_1]\!]env)\ ([\![e_2]\!]env) \\ \mathcal{E}\ [\![\lambda x.e]\!]env &= \lambda v.\ \mathcal{E}\ [\![e]\!]env[v/x] \\ \mathcal{E}\ [\![\textbf{letrec}\ \ f_i = e_i\ \textbf{in}\ \ e]\!]env &= \mathcal{E}\ [\![e]\!]env' \\ & \quad \textbf{whererec}\ \ env' = env[\ [\![e_i]\!]env'/f_i\ ] \end{aligned}$$

We must also change the primitive functions to reflect the fact that functions now take as arguments elements of Ep rather than of D. Since $\mathcal{E}$ maps from Ep to D, this is not difficult to do: we simply call $\mathcal{E}$ on each argument. For example:

$$\begin{aligned} \mathcal{K}_{\mathcal{E}}\ [\![+]\!] &= \lambda ab.\ \mathcal{E}\ a + \mathcal{E}\ b \\ \mathcal{K}_{\mathcal{E}}\ [\![if]\!] &= \lambda pca.\ \textbf{if}\ \ \mathcal{E}\ p\ \textbf{then}\ \ \mathcal{E}\ c\ \textbf{else}\ \ \mathcal{E}\ a \end{aligned}$$

6 Example

To demonstrate how generic environments work in practice, we now present an exact strictness analysis, which calls the new standard semantics. This presentation is a minor variation of [6] (to which the reader is referred for more details); the main difference is that this analysis is "exact" – it uses the standard semantics of the predicate of a conditional to choose between the arms of the conditional. (It is not clear how desirable this analysis is; it is mainly for expository purposes.) Our value domain is a set of variables (those which will certainly be evaluated) together with a higher-order behavior:

$$Sp = \mathcal{P}(Id) \times (Ep \to Sp);$$

when $sp = \langle s, f \rangle \in Sp$, we write $(sp)_{set} = s$ and $(sp)_{fn} = f$. The analysis, $\mathcal{S} : Ep \rightarrow Sp$, is defined by:

$$\begin{array}{rcl}
\mathcal{S}\ [\![c]\!]env & = & \mathcal{K}_{\mathcal{S}}[\![c]\!] \\
\mathcal{S}\ [\![x]\!]env & = & \langle \{x\}, err \rangle \cup_{<} (\mathcal{S}\ env[\![x]\!]) \\
\mathcal{S}\ [\![e_1\ e_2]\!]env & = & Apply_{\mathcal{S}}\ (\mathcal{S}\ [\![e_1]\!]env)\ ([\![e_2]\!]env) \\
\mathcal{S}\ [\![\lambda x.e]\!]env & = & \langle \phi, \lambda v.\ \mathcal{S}\ [\![e]\!]env[v/x] \rangle \\
\mathcal{S}\ [\![\textbf{letrec}\ \ f_i = e_i\ \textbf{in}\ \ e]\!]env & = & \mathcal{S}\ [\![e]\!]env' \\
 & & \textbf{whererec}\ \ env' = env[\ [\![e_i]\!]env'/f_i\ ]
\end{array}$$

The primitives for this analysis are:

$$\begin{array}{rcl}
\mathcal{K}_{\mathcal{S}}\ [\![+]\!] & = & \langle \phi, \lambda a.\ \langle (\mathcal{S}\ a)_{set}, \lambda b.\ (\mathcal{S}\ b)_{set}, error \rangle \rangle \\
\mathcal{K}_{\mathcal{S}}\ [\![if]\!] & = & \langle \phi, \lambda p.\ \langle (\mathcal{S}\ p), \lambda c.\ \textbf{if}\ \ (\mathcal{E}\ p)\ \textbf{then} \\
 & & \qquad \langle (\mathcal{S}\ c), \lambda a.\ \perp_{Sp} \rangle \\
 & & \quad \textbf{else}\ \ \langle \phi, \lambda a.\ (\mathcal{S}\ a) \rangle \rangle \rangle
\end{array}$$

For comparison, an inexact (abstracted) strictness analysis would instead use:

$$\mathcal{K}_{\mathcal{S}}\ [\![if]\!] \ = \ \langle \phi, \lambda p.\ \langle (\mathcal{S}\ p), \lambda c.\ \langle \phi, (\lambda a.\ (\mathcal{S}\ c) \cap (\mathcal{S}\ a)) \rangle \rangle \rangle \rangle.$$

$Apply_{\mathcal{S}}$, $\cup_{<}$, and $\cap$ are defined as follows:

$$\begin{array}{rcl}
Apply_{\mathcal{S}}\ sp\ ep & = & sp \cup_{<} ((sp)_{fn}\ ep), \\
\langle s_1, f_1 \rangle \cup_{<} \langle s_2, f_2 \rangle & = & \langle s_1 \cup s_2, f_2 \rangle, \text{ and} \\
\langle s_1, f_1 \rangle \cap \langle s_2, f_2 \rangle & = & \langle s_1 \cap s_2, \lambda x.\ (f_1\ x) \cap (f_2\ x) \rangle.
\end{array}$$

Note how this composite analysis uses only the one generic environment, which is also passed to the standard semantics. If we had not used a generic environment, we would have had to either pass around two environments (one for $\mathcal{S}$ and one for $\mathcal{E}$) or use a strictness domain which included the value domain, such as

$$Sp^{+} = D \times \mathcal{P}(Id) \times (Ep \rightarrow Sp^{+}).$$

In either case, the values from the standard domain would need to be computed along with the strictness values, and several of the semantic equations would need to be changed. On the other hand, we have implemented this analysis without significant deviation from the above equations.

7 Domain Conversion

In this section, we show a correspondence between (certain) elements of D_{Std} and D.

We need the assumption that elements of our domains are ***extensionally equivalent***; that is, if $d_1, d_2 \in D$ and for all $ep \in Ep$ we have $d_1(ep) = d_2(ep)$, then we also have $d_1 = d_2$ (and similarly for D_{Std}).

We also need some definitions. A value $d_{Std} \in D_{Std}$ is *denotable* if there exists an expression e such that $d_{Std} = \mathcal{E}_{Std}\ [\![e]\!]nullenv$ (where $nullenv$ is the empty environment). Similarly, a value $d \in D$ is denotable if there exists an expression e such that $d = \mathcal{E}\ [\![e]\!]nullenv$. We write $\hat{D}$ for the denotable values in D. Note that there is a function $\mathcal{U} : \hat{D}_{Std} \rightarrow Ep$ which for any denotable d returns some $[\![e]\!]nullenv \in Ep$ which evaluates to d. Note that $\mathcal{E}(\mathcal{U}(d)) = d$ for all $d \in \hat{D}$. Although $\mathcal{E}$ and $\mathcal{U}$ are not strictly inverses, we do have the following interesting properties:

Lemma: If $\mathcal{E}(ep) = \mathcal{E}(ep')$, then for any denotable $d \in Ep \rightarrow D$, we have $d(ep) = d(ep')$.

Proof: (omitted).

Corollary: $d(\mathcal{U}(\mathcal{E}(ep))) = d(ep)$ for all denotable $d \in D$ and $ep \in Ep$.

Proof: $\mathcal{E}(\mathcal{U}(\mathcal{E}(ep))) = \mathcal{E}(ep)$.

We also need to define the *finite* elements of our domains. Let

$$\begin{aligned} D^0_{Std} &= Bas \\ D^i_{Std} &= Bas + (D^{i-1}_{Std} \rightarrow D^{i-1}_{Std}) \end{aligned}$$

$$\begin{aligned} D^0 &= Bas \\ D^i &= Bas + (Ep^{i-1} \rightarrow D_{i-1}) \\ Ep^i &= \{ep | \mathcal{E}(ep) \in D^i\} \end{aligned}$$

Observe $D^i_{Std} \subset D_{Std}$ and $D^i \subset D$ for all i. We say that an element $d \in D$ is *finite* if $d \in D^i$ for some i (similarly for D_{Std}), in which case we say that the *depth* of d is i.

Let $In : D_{Std} \rightarrow D$ and $Out : D \rightarrow D_{Std}$ be defined (by mutual recursion) as follows:

$$
\begin{array}{rcl}
In(d_{Std}) & = & \textbf{if } d_{Std} \in Bas \textbf{ then } d_{Std} \\
 & & \textbf{else } \lambda ep.\ In(d_{Std}(Out(\mathcal{E}(ep)))) \\
Out(d) & = & \textbf{if } d \in Bas \textbf{ then } d \\
 & & \textbf{else } \lambda d_{Std}.\ Out(d(\mathcal{U}(In(d_{Std}))))
\end{array}
$$

Theorem: For all finite, denotable d_{Std}, $Out(In(d_{Std})) = d_{Std}$. Also, for all finite, denotable d, $In(Out(d)) = d$.

Proof: By simultaneous induction on the depth of d_{Std} and d. The base case is trivial. Assume the equations hold for all elements of $\hat{D}^i_{Std}$ and $\hat{D}^i$. Then for $d_{Std} \in \hat{D}^i_{Std} \to \hat{D}^i_{Std}$ and $d \in \hat{D}^i \to \hat{D}^i$, we have

$$
\begin{array}{rcl}
Out(In(d_{Std}) & = & Out(\lambda ep.\ In(d_{Std}(Out(\mathcal{E}(ep))))) \\
 & = & \lambda d'_{Std}.\ Out(\ (\lambda ep.\ In(d_{Std}(Out(\mathcal{E}(ep))))))\ (\mathcal{U}(In(d'_{Std})))) \\
 & = & \lambda d'_{Std}.\ Out(In(d_{Std}(Out(\mathcal{E}(\mathcal{U}(In(d'_{Std}))))))) \\
 & = & \lambda d'_{Std}.\ Out(In(d_{Std}(Out(In(d'_{Std})))))) \\
 & = & \lambda d'_{Std}.\ d_{Std}(d'_{Std})) \\
 & = & d_{Std}
\end{array}
$$

$$
\begin{array}{rcl}
In(Out(d) & = & In(\lambda d_{Std}.\ Out(d(\mathcal{U}(In(d_{Std})))))) \\
 & = & \lambda ep.\ In(\ (\lambda d_{Std}.\ Out(d(\mathcal{U}(In(d_{Std})))))\ (Out(\mathcal{E}(ep)))) \\
 & = & \lambda ep.\ In(Out(d(\mathcal{U}(In(Out(\mathcal{E}(ep))))))) \\
 & = & \lambda ep.\ d(\mathcal{U}(\mathcal{E}(ep))) \\
 & = & \lambda ep.\ d(ep) \\
 & = & d
\end{array}
$$

8 Conclusion

We have presented a new technique, *generic environments*, for writing semantics. Because generic environments are unevaluated, they can be used in any semantic analysis where an evaluated environment would previously have been used. We showed how to write both a standard semantics and a *composite semantics* using this technique, and we showed how the domains which arise correspond to the usual domains.