Adapting Big-Step Semantics to Small-Step Style: Coinductive Interpretations and "Higher-Order" Derivations

Husain Ibraheem and David A. Schmidt
Kansas State University*

February 25, 1998

Abstract

We adapt Kahn-style ("big-step") natural semantics to take on desirable aspects of small-step and denotational semantics forms, more precisely: *(i)* the ability to express divergent computations; *(ii)* the ability to reason about the (length of a) computation of a derivation; and *(iii)* the ability to compute upon and reason about higher-order values. To accomplish these results, we extend the classical, inductive interpretation of natural semantics with coinduction mechanisms and use "negative" rules to express divergence. A simple reformatting of the syntax of derivations allows a simple description of the "length" of a derivation. Finally, the recoding of closure values into denotational-semantics-like functions lets one embed derivations within closures that embed within derivations; in this sense, the semantics becomes "higher order." Examples are given to support the definitional developments.

Kahn-style ("big-step") natural semantics [9] captures a program's entire computation within a single derivation. This representation is more compact than what one obtains from state-transition, "small-step," and denotational definitions; in addition, natural semantics provides several additional attractions:

- When the natural semantics is defined in "environment form," that is, the semantics' propositions are ternary relations of the form, $\rho \vdash e \Downarrow v$, where $\rho \in Environment$, $e \in Expression$, and $v \in Value$, the resulting derivations are (almost always) *semi-compositional.* That is, when one draws a physical derivation tree starting from an initial source program e_0 and initial (empty) environment ρ_0, then every node within the tree has the form $\rho' \vdash e' \Downarrow v'$, where e' is a subphrase of e_0. See Figure 1 for an example.
- One can prove concisely properties of convergent derivations by induction on the height of the derivation tree.
- Static analyses are simple to formalize. The key concept, a program's *collecting semantics* [2, 13, 15], is defined as a simple function of the program's derivation: Given a

*Computing and Information Sciences Department, 234 Nichols Hall, Manhattan, KS 66506 USA. Ph: +1-785-532-6350. `schmidt@cis.ksu.edu`. Supported by NSF CCR-9633388. A reformatted version of this paper appears in *Proc. 2d Workshop on Higher-Order Techniques in Operational Semantics (HOOTS2)*, Stanford, Dec. 1997, Elsevier ENTCS 10 (1998).

Let $\rho_3 = \rho \oplus \{x \mapsto 3\}$ in the following:

$$\begin{tikzcd}
 & & \rho \vdash (\lambda x.x)(1+2) \Downarrow 3 \arrow[dll] \arrow[d] \arrow[drr] & & \\
\rho \vdash (\lambda x.x) \Downarrow \langle \rho, x, x \rangle & & \rho \vdash 1+2 \Downarrow 3 \arrow[dl] \arrow[dr] & & \rho_3 \vdash x \Downarrow 3 \\
 & \rho \vdash 1 \Downarrow 1 & & \rho \vdash 2 \Downarrow 2 &
\end{tikzcd}$$

Figure 1: A semicompositional derivation tree

program, e_0, its initial environment, ρ_0, and its resulting derivation tree, d, define the collecting semantics of subphrase e of e_0 in d to be $Collecting(d, e) = \{\rho \mid \rho \vdash e \Downarrow v$ is a node in $d\}$.

Alas, natural semantics is imperfect: One cannot reason about divergent programs, and its inherent first-ordered-ness makes natural semantics a poor choice for analysis of "open" or "modular" programs. Both denotational and small-step semantics hold advantages in these regards.

In this paper, we modify natural semantics to take on attractive aspects of small-step and denotational semantics without losing its attractive features. The features we wish to gain are

- One should be able to express divergent computations.
- One should be able to reason in terms of the length of a computation, whether the computation is convergent or divergent.
- One should be able to reason about "higher-order" values, e.g., closures.

We achieve our goals for a subset of natural semantics definitions that we call L-attributed—that is, left-to-right processing. L-attributed natural semantics definitions have a natural notion of sequential computation from left-to-right that resembles the computations done with a small-step semantics or with a Prolog interpreter applied to the natural semantics rule schemes [4]. In addition, we can discuss the "length" of a computation.

To gain divergent computations, we generate sets of "positive" (convergent) and "negative" (divergent) rules from the natural semantics rule schemes [3, 14, 13] and apply coinduction definition techniques to the negative rules. Finally, we modify the structure of higher-order values—closures—so that instead of being inert values, closures become "alive" by containing partial derivations. This allows denotational-semantics-like, extensional-style reasoning and opens the door to the application of partial evaluation techniques [5, 8] to natural semantics definitions [7].

1 L-attributed Natural Semantics

Recall that an L-attributed attribute grammar [10] is a Knuthian attribute grammar where information flows from left to right; thus, the attributes for a parse tree can be calculated in a single, left-to-right, tree traversal.

Analogously, we define an *L-attributed*, environment-based, natural-semantics rule scheme as follows. A scheme of the form

$$\frac{\rho_1 \vdash e'_1 \Downarrow v_1 \quad \rho_2 \vdash e'_2 \Downarrow v_2 \quad \cdots \quad \rho_m \vdash e'_m \Downarrow v_m}{\rho_0 \vdash op(e_1, e_2, ..., e_n) \Downarrow v_{m+1}}$$

is L-attributed if

- the value of each ρ_i is a function of $e_1, e_2, ..., e_n$, and those ρ_j and v_j such that $j < i$
- the value of each e'_i is a function of $e_1, e_2, ..., e_n$, and those ρ_j such that $j \leq i$ and those v_j such that $j < i$
- the value of v_{m+1} is a function of $e_1, e_2, ..., e_n$, and those ρ_j and v_j such that $j < m+1$

Again, the intuition is that a derivation with an L-attributed rule can be formulated by a left-to-right tree traversal, where the ρ_i-values are the "inherited attributes" and the v_i-values are the "synthesized attributes." But the rule scheme is a true attribute-grammar rule only if $m = n$ and $e'_i = e_i$, for all $1 \leq i \leq m$. This need not be the case—the standard rule for function application is a counterexample:

$$\frac{\rho \vdash e_1 \Downarrow \langle \rho', x', e' \rangle \quad \rho \vdash e_2 \Downarrow v_2 \quad \rho' \oplus \{x' \mapsto v_2\} \vdash e' \Downarrow v}{\rho \vdash e_1\ e_2 \Downarrow v}$$

Virtually every natural semantics definition in common use is L-attributed.

Assuming that no ρ_i or v_{m+1} manufactures new source program syntax, it is straightforward to prove that a set of L-attributed rule schemes forms a semicompositional definition. (Indeed, L-attribution is unimportant to the proof.)

2 Positive and Negative Rules

Although it is traditional to take an inductive interpretation of natural semantics rule schemes, we desire derivations for divergent programs, so we interpret each L-attributed rule scheme as inducing *positive* and *negative* rules, in the sense of Cousot and Cousot [3, 13]: An L-attributed rule scheme induces a family of *positive* rules of the form

$$+: \frac{\rho_1 \vdash e'_1 \Downarrow v_1 \quad \rho_2 \vdash e'_2 \Downarrow v_2 \quad \cdots \quad \rho_m \vdash e'_m \Downarrow v_m}{\rho_0 \vdash op(e_1, e_2, ..., e_n) \Downarrow v_{m+1}}$$

and m families of *negative* rules of the form

$$-: \frac{\rho_1 \vdash e'_1 \Downarrow v_1 \quad \rho_{k-1} \vdash e'_{k-1} \Downarrow v_{k-1} \quad \rho_k \vdash e_k \Uparrow}{\rho_0 \vdash op(e_1, ..., e_n) \Uparrow} \qquad \text{for all } k \in 1..m$$

If desired, you can read the above as generating one positive rule scheme and m negative rule schemes, but it is traditional to generate the set of substitution instances (*rules*) of a rule scheme, hence we speak of one family of positive rules and m families of negative rules.

The positive rules derive convergent computations and the negative rules derive divergent ones. We define the *convergent derivations* to be the inductive (least-fixed point) interpretation of the positive rules.[1] Next, we define the *divergent derivations* by using the

[1] The L-attributed positive rules are continuous.

Let $\rho_3 = \rho \oplus \{x \mapsto 3\}$ in the following:

$$\begin{tikzcd}
 & & \rho \vdash (\lambda x.x)(1+2) \arrow[dll] \arrow[d] \arrow[drr] \arrow[drrr] & & & \\
\rho \vdash (\lambda x.x) \arrow[d] & & \rho \vdash 1+2 \arrow[dl] \arrow[d] \arrow[dr] & & \rho_3 \vdash x \arrow[d] & 3 \\
\langle \rho, x, x \rangle & \rho \vdash 1 \arrow[d] & \rho \vdash 2 \arrow[d] & 3 & 3 & \\
 & 1 & 2 & & &
\end{tikzcd}$$

Figure 2: Sample derivation in modified syntax

convergent derivations plus a coinductive (greatest fixed-point) interpretation of the negative rules.[2] The set of well-formed derivations, *wfd*, of the natural semantics is the union of the convergent and divergent derivations.

With the inductive interpretation of the positive rules comes inductive reasoning principles, most notably induction on the height of the convergent derivations, and with the coinductive interpretation comes coinductive reasoning.

3 Modifying the Rules into a "Smaller-Step" Format

A user of a natural semantics starts with an initial environment, program pair, ρ_0, e_0, and wants to draw a derivation. When she starts, the user has no clue whether a convergent derivation, $\rho_0 \vdash e_0 \Downarrow v$, or a divergent derivation, $\rho_0 \vdash e_0 \Uparrow$, will develop. The usual hack is to employ a Prolog-style "logical variable," V, to denote either of $\Downarrow v$ or $\Uparrow$, and start writing a derivation with a root of form $\rho_0 \vdash e_0 \Downarrow V$. But this is an imperfect start and fails to accurately deal with divergence—a formalization should do better.

We make the big-step, natural semantics into a "smaller step" one—based on the intuition that the triple, $\rho_0 \vdash e_0 \Downarrow v$, asserts the existence of a finite derivation of small steps, $(\rho_0 \vdash e_0) \rightarrow^* v$, we propose this new grammar for "raw derivations":

$$\begin{array}{l}
d \in \mathit{Derivation} \\
s \in \mathit{Sequent} \;\; (\rho \vdash e) \\
v \in \mathit{Value} \text{ (Note: assume } \mathit{Sequent} \cap \mathit{Value} = \emptyset)
\end{array}$$

$$d \; ::= \; v \;\; | \;\; \begin{tikzcd}[column sep=small, row sep=small] & s \arrow[dl] \arrow[dr] & \\ d_1 & \cdots & d_n \end{tikzcd}, \; n \geq 0$$

That is, a derivation[3] is a value (e.g., an integer or a closure), or a tree whose root is a sequent, s (that is, $\rho \vdash e$), and whose subtrees are derivations.[4] If a sequent, s, has a sequence of computation steps that converge at v, then the corresponding derivation will have s at the root and have as its rightmost child subtree the leaf, v. Figure 2 displays a small example: Such trees bear a relationship to the ones derived with Plotkin-style, "small-step," structural-operational semantics rules [11], once one notes that the arrows from sequent to

[2] The L-attributed negative rules are cocontinuous.

[3] Starting with a universe of finitely branching trees of at most countably infinite depth, we take the coinductive definition of the grammar rule for *Derivation*.

[4] We will also write this in the linear form, $s(d_1, ..., d_n)$.

sequent represent "congruence" steps whereas the arrows from sequent to value represent "$\beta\delta$" steps. Indeed, based on this intuition, one can mechanically reformat the derivation tree into a Plotkin-style derivation by traversing and disassembling the derivation from left to right. This idea provides the intuition for the development in the next section.

Importantly, a divergent program will have a derivation with an infinite path; specifically, the derivation's rightmost path will be infinite, because the negative rules were formulated from L-attributed rule schemes.

From this point onwards, we work with the syntax of derivations defined above, and we use $\mathit{wfd} \subseteq \mathit{Derivation}$ to denote the set of well-formed derivations under the new syntax.

Here is some additional useful metanotation: We use $\rho \vdash e \downarrow v$ and $\rho \vdash e \uparrow$ as follows:

- $\rho \vdash e \downarrow v$ denotes a derivation, d, such that $\mathit{root}(d) = \rho \vdash e$ and the rightmost subtree of d is the leaf, v
- $\rho \vdash e \uparrow$ denotes a derivation, d, such that $\mathit{root}(d) = \rho \vdash e$ and the rightmost subtree of d is *not* a value.

This metanotation gives us a trivial representation of the new formats of positive and negative rules. Given this natural semantics rule scheme,

$$\frac{\rho_1 \vdash e'_1 \Downarrow v_1 \quad \rho_2 \vdash e'_2 \Downarrow v_2 \quad \cdots \quad \rho_m \vdash e'_m \Downarrow v_m}{\rho_0 \vdash op(e_1, e_2, ..., e_n) \Downarrow v_{m+1}}$$

the positive rules we induce have the form

$$+ : \frac{\rho_1 \vdash e'_1 \downarrow v_1 \quad \rho_2 \vdash e'_2 \downarrow v_2 \quad \cdots \quad \rho_m \vdash e'_m \downarrow v_m \quad v_{m+1}}{\rho_0 \vdash op(e_1, e_2, ..., e_n)}$$

and the negative rules now appear

$$- : \frac{\rho_1 \vdash e'_1 \downarrow v_1 \quad \rho_{k-1} \vdash e'_{k-1} \downarrow v_{k-1} \quad \rho_k \vdash e_k \uparrow}{\rho_0 \vdash op(e_1, ..., e_n)}$$

for all $k \in 1..m$. Remember that $\rho \vdash e \downarrow v$ and $\rho \vdash e \uparrow$ describe derivations (not just sequents), therefore the rules resemble Prawitz-style natural deduction rules [12], where antecedents are derivation trees and succedents are sequents/propositions.

It is easy to prove that the set of derivations of the modified rules is isomorphic to the set of derivations of the original rules.

4 Derivation Discovery

The previous two sections defined the "semantics of natural semantics," namely, how the original natural semantics rule schemes induce families of positive and negative rules and how the rules are used to define *wfd*, the set of well-formed derivations.

But users use rule schemes to *compute* derivations—starting from an initial sequent, $\rho_0 \vdash e_0$, a user computes, step by step, a derivation tree. Using the new syntax of derivation trees, defined in the previous section, the process of derivation discovery becomes simple and natural: One uses the original, L-attributed natural semantics rule schemes to draw a

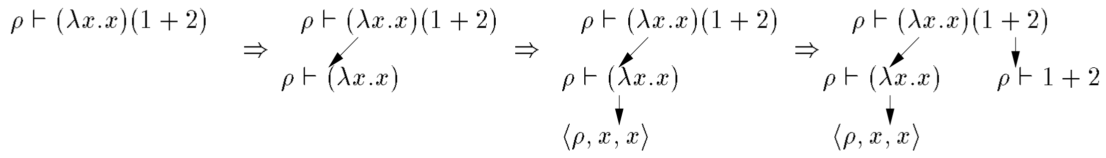


Figure 3: The first four stages of a development

derivation in incremental steps as a left-to-right tree traversal—there is no need for Prolog-style "logical variables." To state this more precisely, an original, L-attributed rule scheme,

$$\frac{\rho_1 \vdash e'_1 \Downarrow v_1 \quad \rho_2 \vdash e'_2 \Downarrow v_2 \quad \cdots \quad \rho_m \vdash e'_m \Downarrow v_m}{\rho_0 \vdash op(e_1, e_2, ..., e_n) \Downarrow v_{m+1}}$$

directs the traversal as follows:

1. Starting from the "goal sequent," $\rho_0 \vdash op(e_1, e_2, ..., e_n)$, generate an arc to the "subgoal," $\rho_1 \vdash e'_1$; attempt to "satisfy" the subgoal by drawing a convergent derivation, $\rho_1 \vdash e'_1 \downarrow v_1$.

2. Once the subgoal is achieved, repeatedly generate and attempt to satisfy subgoals $\rho_i \vdash e'_i$, for $1 < i \leq m$.

3. When all subgoals are achieved, compute v_{m+1} and attach this to the derivation as the rightmost child of the root, $\rho_0 \vdash op(e_1, e_2, ..., e_n)$.

For example, the discovery of the derivation in Figure 2 would start with the steps drawn in Figure 3 and would proceed in similar increments until the tree in Figure 2 results. We call a sequence of such partial derivation trees a *development*. Informally stated, a development is *complete* if while working from left to right, one reaches a derivation that belongs to *wfd*. A finite development might not be complete, and a complete development might not be finite: In the first case, an error in the source program, e.g., $\rho_0 \vdash 2 + true$, can cause an unwanted termination in the development.

In the second case, this rule scheme, $\frac{\rho \vdash loop \Downarrow v}{\rho \vdash loop \Downarrow v}$, would of course generate this positive and this negative rule:

$$+ : \frac{\rho \vdash loop \downarrow v \quad v}{\rho \vdash loop} \qquad - : \frac{\rho \vdash loop \uparrow}{\rho \vdash loop}$$

Using these rules, we obtain from the starting sequent, $\rho_0 \vdash loop + 1$, this infinite derivation

$$\rho_0 \vdash loop + 1 \longrightarrow \rho_0 \vdash loop \longrightarrow \rho_0 \vdash loop \longrightarrow \cdots$$

The derivation is well formed because the rule scheme for arithmetic addition, namely, $\frac{\rho \vdash e_1 \Downarrow v_1 \quad \rho \vdash e_2 \Downarrow v_2}{\rho \vdash e_1 + e_2 \Downarrow plus(v_1, v_2)}$, generates positive rules of this form

$$+ : \frac{\rho \vdash e_1 \downarrow v_1 \quad \rho \vdash e_2 \downarrow v_2 \quad plus(v_1, v_2)}{\rho \vdash e_1 + e_2}$$

$isPartial(v)$
$isPartial(s(d_1, ..., d_k))$ iff $0 \le k \le m$ and there exists
$\pm : \dfrac{d_1 \cdots d_k \quad d_{k+1} \cdots d_m}{s}$
such that for all $0 < j < k$, $isComplete^+(d_j)$, and $isPartial(d_k)$
(Take greatest solution of this definition.)

$isComplete^+(v)$
$isComplete^+(s(d_1, ..., d_m))$ iff there exists $+ : \dfrac{d_1 \cdots d_m}{s}$
such that for all $0 < j \le m$, $isComplete^+(d_j)$
(Take least solution of this definition.)

$isComplete(v)$
$isComplete(s(d_1, ..., d_m))$ iff there exists $\pm : \dfrac{d_1 \cdots d_m}{s}$
such that for all $0 < j < m$, $isComplete^+(d_j)$, and $isComplete(d_m)$
(Take greatest solution of this definition.)

Figure 4: Definitions of partially and completely developed derivations

and negative rules of these two forms

$$- : \frac{\rho \vdash e_1 \uparrow}{\rho \vdash e_1 + e_2} \qquad - : \frac{\rho \vdash e_1 \downarrow v_1 \quad \rho \vdash e_2 \uparrow}{\rho \vdash e_1 + e_2}$$

and the first form of negative rule justifies that the derivation is well formed, that is, it is a member of *wfd*.

Figure 4 gives formalizations of when a raw derivation is partially developed and is completely developed. An important consequence of the definitions is that left-to-right derivation discovery generates a sequence of partially developed trees. When the sequence completes, the result is a complete development.

4.1 Semantics of Developments

Figure 4 defines the crucial aspects of developments, and several easily proved consequences follow. First, here is a simple semantics of developments: Given a development, $d_0 \Rightarrow d_1 \Rightarrow \cdots \Rightarrow d_i \Rightarrow \cdots$, define the set of derivations denoted by a partial derivation, d_i, to be $S_i = \{d \in wfd \mid d \text{ extends } d_i\}$. That is, each d_i defines the set of those well-formed derivations whose prefix is d_i. For all $j > i$, $S_j \subseteq S_i$. The semantics of a complete development is $S_\omega = \bigcap_{i \ge 0} S_i$.

With this semantics, we have easy proofs of

- *soundness:* every completely developed derivation is well formed, that is, it is a member of *wfd*
- *adequacy:* every $d \in wfd$ has a complete development

Soundness follows because the limit of a complete development is a derivation that belongs to S_ω; adequacy holds because the natural semantics is L-attributed, hence every well-formed

$$\rho \vdash x \Downarrow \rho_x \qquad\qquad \rho \vdash \lambda x.\, e \Downarrow \boldsymbol{\lambda} A.\, (\rho \oplus \{x \mapsto A\} \vdash e)^*$$

where s^* denotes a derivation, d, such that $root(d) = s$ and *isPartial(d)*

$$\frac{\rho \vdash e_1 \Downarrow \boldsymbol{\lambda} A.\, d \quad \rho \vdash e_2 \Downarrow v_2 \quad [v_2/A]d \Downarrow v}{\rho \vdash e_1\ e_2 \Downarrow v}$$

Figure 5: Natural semantics rule schemes for higher-order closures

derivation must have a complete development that is a sequence of partial derivations drawn in a left-to-right traversal.

Hence the set of well-formed derivations, *wfd*, is characterized by the set of complete developments. As a result, one obtains a small-step-semantics-like induction principle: induction on the length of a complete development.

5 Higher-Order Natural Semantics

Traditional natural semantics is "too first ordered"—values like closures, $\langle \rho, x, e \rangle$, are inert, "dumb," data structures. In this section, we make closures alive and "smart" by modifying them so that they contain raw derivations—a closure, $\langle \rho, x, e \rangle$ is reformatted into $\boldsymbol{\lambda} A.\ \rho \oplus \{x \mapsto A\} \vdash e$, that is, a mapping from a value, A, to a sequent that uses A. Of course, the sequent is a raw derivation, so we generalize the idea and allow a closure to hold a partially developed derivation, d, that is, $\boldsymbol{\lambda} A.\ d$, such that $root(d) = \rho \oplus \{x \mapsto A\} \vdash e$.

Using this formulation, we can develop the bodies of closures and we can apply techniques akin to those from denotational semantics to reason about the behavior of a closure prior to its application.

The syntax we use goes as follows:

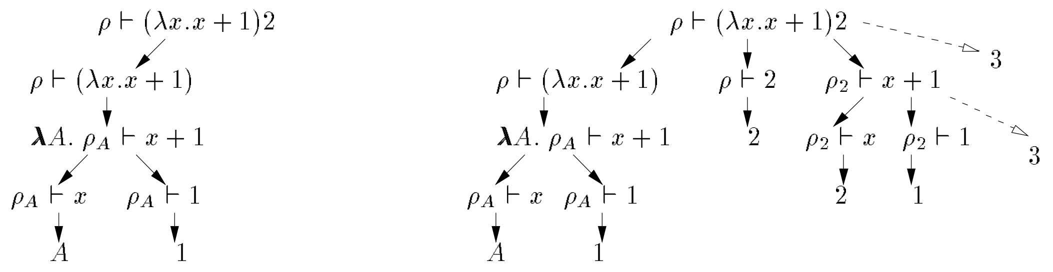
$d \in Derivation$
$s \in Sequent\ \ (\rho \vdash e)$
$v \in Value$ (Note: assume $Sequent \cap Value = \emptyset$)
$A \in VariableValue$
$p \in PrimitiveFirstOrderValue$ (e.g., integer)

$$\begin{array}{rcl} d & ::= & v \mid \overset{s}{d_1 \cdots d_n},\ n \geq 0 \\ v & ::= & p \mid A \mid \boldsymbol{\lambda} A.\ d \end{array}$$

This syntax is "higher order" in the sense that derivations can contain values that can contain derivations.

Figure 5 presents a set of natural semantics rule schemes for the new form of closures. The rule schemes in Figure 5 permit a development to proceed within a closure value; we call this a *speculative development*. For example, Figure 6(a) shows a partial development of a program where the value part for $\lambda x.\ x{+}1$ is a closure that contains speculative development. This speeds up the the remainder of the development, because the speculative development can be used with the substitution of 2 for A, as seen in Figure 6(b); this quickly leads to completion of the derivation, as indicated by the two dotted arrows in Figure 6(b).

Let $\rho_\alpha = \rho \oplus \{x \mapsto \alpha\}$ in the following:



(a) (b)

Figure 6: An example with speculative development

Of course, a serious application of speculative development would not limit itself to a strict left-to-right strategy; examples like $\rho \vdash \lambda x.\ (x+1)+(2+3)$ and $\rho \vdash \lambda x.\ \mathtt{if}\ x\ (\lambda y.\ x)\ (\lambda x.\ x)$ can benefit from a non-leftmost speculative development.

Indeed, partial evaluation [5, 8] applied to natural semantics derivations is aggressive speculative development, and as an ongoing project, we have been applying partial evaluation to the variant of natural semantics seen in this paper to uncover the feasibility of automated, static analysis of open ("modular") programs [7].

6 Applications, Related Work, and Conclusions

As noted above, the modifications of natural semantics derivations let one work easily with partial derivations and derivations of open programs (source programs containing unbound variables). Partial evaluation techniques now apply directly to natural semantics framework, as demonstrated by the ongoing Ph.D. research of Ibraheem [7], where a supercompilation algorithm by Glück and Sørensen has been adapted to operate on partial derivations in place of so-called "process trees" [5]. This line of research aims to show that static analysis of program modules can be performed with the same partial evaluation machinery used for complete programs.

Of course, the intuition that a natural semantics derivation can be computed by a Prolog interpreter is a standard one. Deransart and Ferrand [4] give a careful formulation of how a goal tree can be computed by a Prolog interpretater in a left-to-right, incremental fashion. Gunter and Remy [6] adapt Prolog goal search to natural semantics definitions; they define a typed notation, RAVL, for their presentation and show how one might prove a type-safety property in their framework. And Berry's Ph.D. thesis describes a suite of tools for drawing "animations" of the incremental generation of partial derivations [1]. Most recently, Stoughton has studied the formalizaton of partial generation of derivations within resumption semantics [16].

Based on the investigations in this paper, one must conclude that the relationships between big-step, small-step, and denotational semantics are closer, in a formal sense, than what has been demonstrated so far in the literature. Although it is premature to speculate exactly what class of semantic definition is simultaneously both small-step- and big-step-like, it is clear that there is a subclass of operational semantics definitions that form an intersection between the two formats. And, the connections between denotational and operational semantics need closer examination as well.

References

[1] D. Berry. *Generating Program Animators from Programming Language Semantics.* PhD thesis, Edinburgh University, 1991.

[2] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs. In *Proc. 4th ACM Symp. on Principles of Programming Languages*, pages 238–252. ACM Press, 1977.

[3] P. Cousot and R. Cousot. Inductive definitions, semantics, and abstract interpretation. In *Proc. 19th ACM Symp. on Principles of Programming Languages*, pages 83–94. ACM Press, 1992.

[4] P. Deransart and G. Ferrand. An operational formal definition of PROLOG. In *Proc. IEEE Symp. on Logic Programming*, pages 162–172. IEEE Press, 1987.

[5] R. Glück and M. H. Sørensen. A roadmap to metacomputation by supercompilation. In O. Danvy, R. Glück, and P. Thiemann, editors, *Partial Evaluation*, number 1110 in Lecture Notes in Computer Science, pages 137–160. Springer-Verlag, 1996.

[6] C. Gunter and D. Rémy. A proof-theoretic assessment of runtime type errors. Technical memo 11261-921230-43TM, AT&T Bell Laboratories, Murray Hill, PA, 1993.

[7] H. Ibraheem and D. Schmidt. Partial evaluation of higher-order natural-semantics derivations. In M. Leuschel, editor, *Proc. Workshop on Specialization of Declarative Programs and its Applications, Port Jefferson, NY.* `http://www.cis.ksu.edu/~schmidt/papers/driving.ps.Z`, 1997.

[8] N.D. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice-Hall, 1993.

[9] G. Kahn. Natural semantics. In *Proc. STACS '87*, pages 22–39. Lecture Notes in Computer Science 247, Springer, Berlin, 1987.

[10] P. Lewis, D. Rosenkrantz, and R. Stearns. *Compiler Design Theory.* Addison-Wesley, 1978.

[11] Gordon D. Plotkin. A structural approach to operational semantics. Technical Report FN-19, Computer Science Department, Aarhus University, Aarhus, Denmark, September 1981.

[12] D. Prawitz. *Natural Deduction: A Proof-Theoretical Study.* Almqvist and Wiksell, Stockholm, 1965.

[13] D.A. Schmidt. Trace-based abstract interpretation of operational semantics. *J. Lisp and Symbolic Computation.* In press. Available from `www.cis.ksu.edu/~schmidt/papers/aiosh.ps.Z`

[14] D.A. Schmidt. Natural-semantics-based abstract interpretation. In A. Mycroft, editor, *Static Analysis Symposium*, number 983 in Lecture Notes in Computer Science, pages 1–18. Springer-Verlag, 1995.

[15] D.A. Schmidt. Data-flow analysis is model checking of abstract interpretations. In *Proc. 25th ACM Symp. on Principles of Prog. Languages.* ACM Press, 1998.

[16] A. Stoughton. An operational semantics framework supporting the incremental construction of derivation trees. In C. Talcott, editor, *Proceedings 2nd Workshop on Higher-Order Operational Techniques in Operational Semantics, Stanford*, volume 10 of *Electronic Notes in Theoretical Computer Science.* Elsevier Science Publishers, 1997.