An Operational Semantics Framework Supporting the Incremental Construction of Derivation Trees[*]

Allen Stoughton[†]
Department of Computing and Information Sciences
Kansas State University
Manhattan, KS 66506, USA
`allen@cis.ksu.edu`
`http://www.cis.ksu.edu/~allen/home.html`

Abstract. We describe the current state of the design and implementation of Dops, a framework for Deterministic OPerational Semantics that will support the incremental construction of derivation trees, starting from term/input pairs. This process of derivation tree expansion may terminate with either a complete derivation tree, explaining why a term/input pair evaluates to a particular output, or with a blocked incomplete derivation tree, explaining why a term/input pair fails to evaluate to an output; or the process may go on forever, yielding, in the limit, an infinite incomplete derivation tree, explaining why a term/input pair fails to evaluate to an output.

The Dops metalanguage is a typed lambda calculus in which all expressions converge. Semantic rules are specified by lambda terms involving resumptions, which are used by a rule to consume the outputs of sub-evaluations and then resume the rule's work. A rule's type describes the number and kinds of sub-evaluations that the rule can initiate, and indicates whether the rule can block. The semantics of Dops is defined in an object language-independent manner as a small-step semantics on concrete derivation trees: trees involving resumptions. These concrete derivation trees can then be abstracted into ordinary derivation trees by forgetting the resumptions.

1 The incremental construction of derivation trees

We begin by defining the operational semantics that we will use as an example throughout the rest of the paper: a big-step, environment-based semantics of the untyped, call-by-value lambda calculus.

The set Exp of *lambda expressions* is inductively defined by the rules of Figure 1, where Int is the set of all integers. We abbreviate

$$\mathsf{App}\langle \mathsf{Lam}\langle 0, \mathsf{App}\langle \mathsf{Var}\,0, \mathsf{Var}\,0\rangle\rangle, \mathsf{Lam}\langle 0, \mathsf{App}\langle \mathsf{Var}\,0, \mathsf{Var}\,0\rangle\rangle\rangle$$

to $(\lambda_0.\, v_0\, v_0)(\lambda_0.\, v_0\, v_0)$, and make use of similar abbreviations (in which v_n abbreviates $\mathsf{Var}\, n$, the nth variable) without further comment.

[*]Appears in the proceedings of the *Second Workshop on Higher-Order Operational Techniques in Semantics (HOOTS II)*, Electronic Notes in Theoretical Computer Science, vol. 10, Elsevier Science B. V., 1998.

[†]This work was partially supported by NSF/DARPA grant CCR-9633388.

Figure 1: Syntax of lambda expressions

$$\frac{n \in \mathsf{Int}}{\mathsf{Var}\, n \in \mathsf{Exp}} \qquad \frac{a, b \in \mathsf{Exp}}{\mathsf{App}\langle a, b\rangle \in \mathsf{Exp}} \qquad \frac{n \in \mathsf{Int} \quad a \in \mathsf{Exp}}{\mathsf{Lam}\langle n, a\rangle \in \mathsf{Exp}}$$

Figure 2: Values and environments

$$\frac{e \in \mathsf{Env} \quad n \in \mathsf{Int} \quad a \in \mathsf{Exp}}{\mathsf{Clos}\langle e, n, a\rangle \in \mathsf{Vlu}}$$

$$\mathsf{Nil}\langle\,\rangle \in \mathsf{Env} \qquad \frac{n \in \mathsf{Int} \quad x \in \mathsf{Vlu} \quad e \in \mathsf{Env}}{\mathsf{Cons}\langle n, x, e\rangle \in \mathsf{Env}}$$

To define the semantics of expression evaluation, we need two simple semantic spaces. The sets Vlu of *values* and Env of *environments* are defined inductively by the rules of Figure 2. For example, if x and y are values, then

$$e = \mathsf{Cons}\langle 1, x, \mathsf{Cons}\langle 3, y, \mathsf{Nil}\langle\,\rangle\rangle\rangle$$

is an environment: the one in which variable 1 has value x, and variable 3 has value y. Note that e is sorted by variable number. The functions on environments that we define will assume and preserve the sortedness of environments. We also need the familiar auxiliary functions for looking up the value of an identifier in an environment and updating an environment to reflect a new binding:

$$\begin{array}{l}\mathsf{Lookup}\colon \mathsf{Env} \to \mathsf{Int} \to (\{\langle\,\rangle\} + \mathsf{Vlu}) \\ \mathsf{Update}\colon \mathsf{Env} \to \mathsf{Int} \to \mathsf{Vlu} \to \mathsf{Env}.\end{array}$$

To understand the significance of the sum in Lookup's type, consider the environment e defined above. Then, $\mathsf{Lookup}\, e\, 2$ is $\mathsf{in}(0, \langle\,\rangle)$, the injection of the empty tuple into the 0th component of the sum, since variable 2 is not bound in e. And $\mathsf{Lookup}\, e\, 1$ is $\mathsf{in}(1, x)$, the injection of variable 1's value in e into the 1st component of the sum.

The semantics of expression evaluation can be defined as in Figure 3, where we read "$a, e \Rightarrow x$" as "expression a in environment e evaluates to value x". We consider the single premise of the variable evaluation rule to be a "side-condition", since it doesn't involve expression evaluation. The most straightforward way to view this definition is as the inductive definition of the relation $\Rightarrow \subseteq \mathsf{Exp} \times \mathsf{Env} \times \mathsf{Vlu}$, where $a, e \Rightarrow x$ abbreviates $\langle a, e, x\rangle \in \Rightarrow$. Given this interpretation, we can prove the following facts:

Figure 3: Definition of expression evaluation

$$\frac{\mathsf{Lookup}\,e\,n = \mathsf{in}(1, x)}{\mathsf{Var}\,n, e \Rightarrow x}$$

$$\mathsf{Lam}\langle n, a\rangle, e \Rightarrow \mathsf{Clos}\langle e, n, a\rangle$$

$$\frac{a, e \Rightarrow \mathsf{Clos}\langle e', n, a'\rangle \quad b, e \Rightarrow y \quad a', \mathsf{Update}\,e'\,n\,y \Rightarrow z}{\mathsf{App}\langle a, b\rangle, e \Rightarrow z}$$

Figure 4: Complete derivation

$$\begin{tikzcd}
\underset{(2)}{\lambda_0.\,v_0, \mathsf{Nil}\langle\,\rangle} \Rightarrow \underset{(3)}{x} \arrow[dr, no head] & \underset{(4)}{\lambda_0.\,v_0, \mathsf{Nil}\langle\,\rangle} \Rightarrow \underset{(5)}{x} \arrow[d, no head] & \underset{(6)}{v_0, e_{0,x}} \Rightarrow \underset{(7)}{x} \arrow[dl, no head] \\
 & \underset{(1)}{(\lambda_0.\,v_0)(\lambda_0.\,v_0), \mathsf{Nil}\langle\,\rangle} \Rightarrow \underset{(8)}{x} &
\end{tikzcd}$$

(where $x = \mathsf{Clos}\langle \mathsf{Nil}\langle\,\rangle, 0, v_0\rangle$ and $e_{0,x} = \mathsf{Cons}\langle 0, x, \mathsf{Nil}\langle\,\rangle\rangle$)

1. $\Rightarrow$ is a partial function from $\mathsf{Exp} \times \mathsf{Env}$ to Vlu.

2. $(\lambda_0.\,v_0)(\lambda_0.\,v_0), \mathsf{Nil}\langle\,\rangle \Rightarrow \mathsf{Clos}\langle \mathsf{Nil}\langle\,\rangle, 0, v_0\rangle$.

3. $(\lambda_0.\,v_1)(\lambda_0.\,v_0), \mathsf{Nil}\langle\,\rangle \Rightarrow x$ for no $x \in \mathsf{Vlu}$.

4. $(\lambda_0.\,v_0\,v_0)(\lambda_0.\,v_0\,v_0), \mathsf{Nil}\langle\,\rangle \Rightarrow x$ for no $x \in \mathsf{Vlu}$.

It is also possible to think about expression evaluation more concretely. For example, Figure 4 (ignore the labels (1)-(8) for now) consists of a derivation tree proving (providing evidence for) Fact (2). Since the leftmost and middle children of this tree are instances of the axiom for abstraction evaluation, and the rightmost child follows by the variable rule (we omit the rule's premise, since it's a side-condition), the conclusion follows by the application rule.

But, it is also possible and useful to think even more concretely, to focus on the step-by-step procedure in which derivation trees are constructed. With the tree of Figure 4, we begin, in Step (1), with the incomplete derivation tree consisting of the expression/environment

Figure 5: Blocked incomplete derivation

$$\begin{tikzcd}[column sep=small]
\begin{matrix}\lambda_0\ldotp v_1, \mathsf{Nil}\langle\,\rangle \Rightarrow x \\ (2) \qquad\quad (3)\end{matrix} \arrow[dr, no head] & \begin{matrix}\lambda_0\ldotp v_0, \mathsf{Nil}\langle\,\rangle \Rightarrow y \\ (4) \qquad\quad (5)\end{matrix} \arrow[d, no head] & \begin{matrix}v_1, e_{0,y} \Rightarrow \\ (6)\end{matrix} \arrow[dl, no head] \\
& \begin{matrix}(\lambda_0\ldotp v_1)(\lambda_0\ldotp v_0), \mathsf{Nil}\langle\,\rangle \Rightarrow \\ (1)\end{matrix} &
\end{tikzcd}$$

(where $x = \mathsf{Clos}\langle \mathsf{Nil}\langle\,\rangle, 0, v_1\rangle$, $y = \mathsf{Clos}\langle \mathsf{Nil}\langle\,\rangle, 0, v_0\rangle$ and $e_{0,y} = \mathsf{Cons}\langle 0, y, \mathsf{Nil}\langle\,\rangle\rangle$)

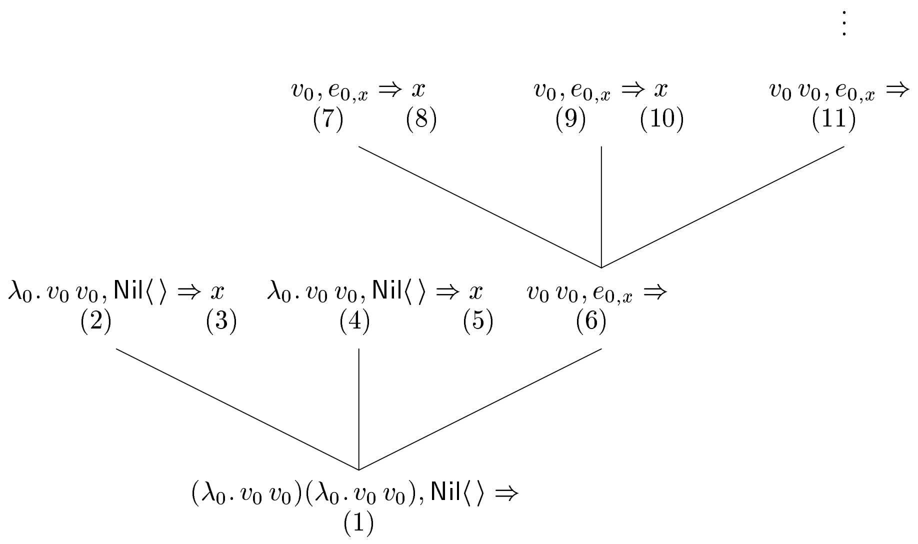
pair $(\lambda_0\ldotp v_0)(\lambda_0\ldotp v_0), \mathsf{Nil}\langle\,\rangle$. Next, since our expression is an application, we begin evaluating the left side of the application, in Step (2), and finish this evaluation, in Step (3). In Steps (4) and (5), we evaluate the right side of the application. In Steps (6) and (7), we evaluate the expression of the closure x in the environment that is formed by binding the variable of the closure in the environment of the closure to the value of the right side of the application. Finally, in Step (8), we take the result of this rightmost evaluation and make it the result of our overall evaluation, giving us a complete derivation providing evidence for Fact (2).

It is easy to prove that the tree expansion procedure that we followed above is sound and complete. If we start with an incomplete tree consisting of an expression/environment pair a, e and terminate with a complete tree whose root is $a, e \Rightarrow x$, then a, e evaluates to x. And, if a, e evaluates to x, then the procedure will turn the incomplete tree consisting of a, e into a complete tree with root $a, e \Rightarrow x$. When the procedure doesn't terminate with a complete derivation tree, it provides an explanation for why the starting expression/environment pair fails to evaluate to any value. Figure 5 gives an explanation for why Fact (3) holds: the procedure terminates with a blocked incomplete derivation tree, since variable 1 is not bound in environment $e_{0,y}$. And Figure 6 gives an explanation for why Fact (4) holds: the procedure fails to terminate, giving, in the limit, an infinite incomplete derivation tree.

2 A framework for deterministic operational semantics

We are designing and implementing a framework for Deterministic OPerational Semantics called Dops that will support the incremental construction of derivation trees, starting from object language term/input pairs. We restrict our attention to deterministic semantics for two reasons. First, one often wants a semantics to be deterministic, and so it is useful to have frameworks in which expressed semantics are guaranteed to be deterministic. Second,

Figure 6: Infinite incomplete derivation


$v_0, e_{0,x} \Rightarrow x$
(7) (8)
$v_0, e_{0,x} \Rightarrow x$
(9) (10)
$v_0\, v_0, e_{0,x} \Rightarrow$
(11)
$\lambda_0.\, v_0\, v_0, \mathsf{Nil}\langle\,\rangle \Rightarrow x$
(2) (3)
$\lambda_0.\, v_0\, v_0, \mathsf{Nil}\langle\,\rangle \Rightarrow x$
(4) (5)
$v_0\, v_0, e_{0,x} \Rightarrow$
(6)
$(\lambda_0.\, v_0\, v_0)(\lambda_0.\, v_0\, v_0), \mathsf{Nil}\langle\,\rangle \Rightarrow$
(1)

(where $x = \mathsf{Clos}\langle \mathsf{Nil}\langle\,\rangle, 0, v_0\, v_0\rangle$ and $e_{0,x} = \mathsf{Cons}\langle 0, x, \mathsf{Nil}\langle\,\rangle\rangle$)

to be useful in practice, our tree expansion procedure itself will have to be deterministic, which means that any nondeterminism would have to be reflected either in the structure of the derivation trees themselves or in a lack of monotonicity of the tree expansion process. Neither of these alternatives seems desirable.

The operational semantics of an object language is expressed in the Dops metalanguage, a typed lambda calculus with sums, products, algebraic datatypes (recursive types not directly involving function types) and primitive recursion. This lambda calculus only expresses total functions, i.e., all well-typed lambda terms converge to values. In the metalanguage, n-ary sums, products and tuples are written like $[\sigma_0, \ldots, \sigma_{n-1}]$, $\{\sigma_0, \ldots, \sigma_{n-1}\}$ and $\{x_0, \ldots, x_{n-1}\}$, respectively, so that $\{\}$ is the single value of the unit type $\{\}$. Much of the metalanguage's syntax is reminiscent of Standard ML.

The syntactic categories of an object language are defined as sorts in the Dops metalanguage, and each sort has associated with it *input* and *output types*. Figure 7 shows how the single sort Exp of our example object language, along with its associated input and output types, Env and Vlu, can be expressed in our metalanguage. Figure 8 shows how the auxiliary functions Lookup and Update can be defined in the Dops metalanguage, using primitive recursion. (Straightforward constraints are used to prohibit general recursion.)

For each constructor of a given sort (Var, Lam and App in our example object language), a corresponding semantic rule must be specified as a metalanguage term. When an object

Figure 7: Dops definitions of sorts and datatypes

```
sort Exp = Var of Int | App of {Exp, Exp} | Lam of {Int, Exp}

datatype Vlu = Clos of {Env, Int, Exp}
and      Env = Nil  of {} | Cons of {Int, Vlu, Env}
```

Figure 8: Dops definitions of auxiliary functions

```
type VluOpt = [{}, Vlu]

rec Lookup : Env -> Int -> VluOpt =
      Nil{}         => fn _ : Int => in(0, VluOpt, {})
    | Cons{l, u, e} => fn n : Int =>
        case n < l of
             True{}  => in(0, VluOpt, {})
           | False{} =>
               case n = l of
                    True{}  => u
                  | False{} => Lookup e n
               esac
        esac

rec Update : Env -> Int -> Vlu -> Env =
      Nil{}         => fn n : Int => fn v : Int => Cons{n, v, Nil{}}
    | Cons{l, u, e} => fn n : Int => fn v : Int =>
        case n < l of
             True{}  => Cons{n, v, Cons{l, u, e}}
           | False{} =>
               case n = l of
                    True{}  => Cons{l, v, e}
                  | False{} => Cons{l, u, Update e n v}
               esac
        esac
```

language semantics would ordinarily have multiple inference rules for a single constructor, e.g., for a conditional operator, the multiple rules will have to be combined into a single metalanguage term, in a process that is similar to the "left-factoring" of [7]. The rule corresponding to a constructor δ of sort σ takes in an instance p of the constructor's data and a value x of σ's input type, and describes how the term $\delta\, p$ should be evaluated with input x. This evaluation may cause various sub-evaluations to be initiated, and may eventually terminate with the production of a value of σ's output type. Resumptions are used to consume the output values of sub-evaluations and then resume the rule's work. The types of rules involve *action types*, which describe the number and kinds of sub-evaluations that an application of a rule is capable of initiating, and indicate whether an application of a rule is capable of blocking. Since the metalanguage is deterministic, and there is only one rule per constructor, only deterministic semantics can be expressed in Dops.

Figure 9 shows how the semantic rules of our example object language can be expressed in the Dops metalanguage. Consider the most complex of these rules: App. The lambda term for App takes in the left and right sides, a and b, of the application to be evaluated, along with the environment e in which the evaluation should be carried out. It then returns an element of the action type AppAct. Action types always consist of sums with two or more components. Since the 0th component of AppAct is the empty type, we know that the evaluation of an application is incapable of immediately blocking; if it had been the unit type, then immediate blocking might have been possible. And, since the 1st component of AppAct is also the empty type, we know that the evaluation of an application cannot immediately result in a value of type Vlu (the output type of our constructor's sort); if this component had been Vlu, then immediate production of an output value might have been possible. Thus the value returned by the application rule will have to consist of (the injection into the 2nd component of the sum of) a triple with type $\{\mathsf{Exp}, \mathsf{Env}, \mathsf{Vlu} \to \mathsf{AppAct1}\}$. The triple returned should be thought of as a request to initiate a sub-evaluation: to evaluate the 0th component of the triple with its 1st component as input, and then to supply the output value produced by this sub-evaluation to the resumption that is the 2nd component of the triple. The actual triple returned is thus a request to evaluate the left side a of the application in the environment e, and then to call the supplied resumption with the output value x of this sub-evaluation. The value x must be a closure, and the resumption first gives names to the components of the closure, and then initiates a second sub-evaluation: the evaluation of the right side b of the application in the environment e, where the output value y of the sub-evaluation is to be given to the supplied resumption. This resumption, when invoked, will initiate a third and final sub-evaluation: the evaluation of the expression a' of the closure in the environment that is obtained by updating the environment e' of the closure so that the variable n of the closure is bound to the value y of b, where the output value z of this sub-evaluation is to be given to the final resumption, which must produce a value of action type $\mathsf{AppAct3}$. Since $\mathsf{AppAct3}$ has only two components, and only its 1st

Figure 9: Dops definitions of semantic rules

```
type VarAct = [{}, Vlu]

rule Var : Int -> Env -> VarAct =
       fn n : Int => fn e : Env => Lookup e n

type AppAct3 = [[], Vlu]
type AppAct2 = [[], [], {Exp, Env, Vlu -> AppAct3}]
type AppAct1 = [[], [], {Exp, Env, Vlu -> AppAct2}]
type AppAct  = [[], [], {Exp, Env, Vlu -> AppAct1}]

rule App : {Exp, Exp} -> Env -> AppAct =
       fn {a, b} : {Exp, Exp} => fn e : Env =>
            in(2, AppAct,
               {a, e,
                fn x : Vlu =>
                     case x of
                          Clos{e', n, a'} =>
                            in(2, AppAct1,
                               {b, e,
                                fn y : Vlu =>
                                     in(2, AppAct2,
                                        {a', Update e' n y,
                                         fn z : Vlu =>
                                              in(1, AppAct3, z)})})
                     esac})

type LamAct = [[], Vlu]

rule Lam : {Int, Exp} -> Env -> LamAct =
       fn {n, a} : {Int, Exp} => fn e : Env =>
            in(1, LamAct, Clos{e, n, a})
```

Figure 10: Concrete derivation trees

$$\mathsf{Init}\langle a,x\rangle \in \Gamma \qquad \frac{\bar{\gamma}\in\Gamma^+}{\mathsf{Inc}\langle a,x,\bar{\gamma},f\rangle \in \Gamma} \qquad \frac{\bar{\gamma}\in\Gamma^*}{\mathsf{Comp}\langle a,x,\bar{\gamma},y\rangle \in \Gamma}$$

component is nonempty, this resumption must yield (the injection into the 1st component of $\mathsf{AppAct3}$ of) an output value. The actual value returned is, of course, z.

By examining the action types VarAct and LamAct, we can tell that evaluating variables and lambda expressions never involves the initiation of sub-evaluations. In particular, the side-condition of the variable evaluation rule is handled inside the rule. According to VarAct, variable evaluation may be capable of blocking, since its 0th component is the unit type; and, if we look at the semantic rule for variable evaluation, we will see that variable evaluation blocks when a variable is looked up in an environment where it is unbound. On the other hand, LamAct tells us that lambda expression evaluation always terminates normally.

The semantics of Dops is defined in an object language-independent manner via a small-step semantics on the set Γ of *concrete derivation trees*, which are inductively defined in Figure 10. In this figure, Γ^* denotes the set of all tuples of elements of Γ, and Γ^+ denotes the set of all nonempty tuples of elements of Γ. When evaluating a term a with input x, one starts with the initial concrete derivation tree $\mathsf{Init}\langle a,x\rangle$. After some number of tree expansion steps, one may have an incomplete concrete derivation tree of the form $\mathsf{Inc}\langle a,x,\bar{\gamma},f\rangle$. Here the elements of $\bar{\gamma}$ are the sub-derivations that have been constructed so far during the evaluation, and the resumption f is waiting for the last sub-derivation of $\bar{\gamma}$ to become complete; then the output value of this sub-derivation will be supplied to the resumption. Eventually, the tree expansion process may terminate with a complete concrete derivation tree of the form $\mathsf{Comp}\langle a,x,\bar{\gamma},y\rangle$. Here, y is the output value obtained after evaluating a with input x, and $\bar{\gamma}$ would only be the empty tuple if the complete derivation tree was formed directly from $\mathsf{Init}\langle a,x\rangle$. There is a typing system for concrete derivation trees that puts some additional constraints on these trees, requiring the types of their components to be compatible and requiring all non-final sub-derivations to be complete.

The tree expansion relation $\to \subseteq \Gamma\times\Gamma$ is inductively defined by Figure 11, where $i \geq 0$, $\Downarrow$ is the metalanguage evaluation relation, rule_δ denotes the rule (a metalanguage term) corresponding to the constructor δ, and @ appends tuples. Note that the premises of the first four rules don't involve tree expansion and so can be viewed as side-conditions.

The first two tree expansion rules show how an initial concrete derivation tree $\mathsf{Init}\langle \delta\, p, x\rangle$ is expanded. We proceed by taking the rule corresponding to the constructor δ and applying it to the constructor's data p and the input value x. If our derivation tree is well-typed, this

Figure 11: Definition of tree expansion relation

$$\frac{\mathsf{rule}_\delta\, p\, x \Downarrow \mathsf{in}(1, \sigma, y)}{\mathsf{Init}\langle \delta\, p, x\rangle \to \mathsf{Comp}\langle \delta\, p, x, \langle\,\rangle, y\rangle}$$

$$\frac{\mathsf{rule}_\delta\, p\, x \Downarrow \mathsf{in}(i+2, \sigma, \{a, y, f\})}{\mathsf{Init}\langle \delta\, p, x\rangle \to \mathsf{Inc}\langle \delta\, p, x, \langle \mathsf{Init}\langle a, y\rangle\rangle, f\rangle}$$

$$\frac{f\, z \Downarrow \mathsf{in}(1, \sigma, w)}{\begin{array}{r}\mathsf{Inc}\langle a, x, \bar{\gamma}_1 \mathbin{@} \langle \mathsf{Comp}\langle b, y, \bar{\gamma}_2, z\rangle\rangle, f\rangle \quad \to \\ \mathsf{Comp}\langle a, x, \bar{\gamma}_1 \mathbin{@} \langle \mathsf{Comp}\langle b, y, \bar{\gamma}_2, z\rangle\rangle, w\rangle\end{array}}$$

$$\frac{f\, z \Downarrow \mathsf{in}(i+2, \sigma, \{c, w, g\})}{\begin{array}{r}\mathsf{Inc}\langle a, x, \bar{\gamma}_1 \mathbin{@} \langle \mathsf{Comp}\langle b, y, \bar{\gamma}_2, z\rangle\rangle, f\rangle \qquad\qquad \to \\ \mathsf{Comp}\langle a, x, \bar{\gamma}_1 \mathbin{@} \langle \mathsf{Comp}\langle b, y, \bar{\gamma}_2, z\rangle, \mathsf{Init}\langle c, w\rangle\rangle, g\rangle\end{array}}$$

$$\frac{\gamma_1 \to \gamma_2}{\mathsf{Inc}\langle \delta\, p, x, \bar{\gamma} \mathbin{@} \langle \gamma_1\rangle, f\rangle \to \mathsf{Inc}\langle \delta\, p, x, \bar{\gamma} \mathbin{@} \langle \gamma_2\rangle, f\rangle}$$

will result in a value of the action type σ of δ's rule (metalanguage non-termination is not possible). If the action type allows for immediate blocking, then the resulting value may have the form $\mathsf{in}(0, \sigma, \{\})$, which means that object language blocking will occur. Otherwise, there are two possibilities. The resulting value may have the form $\mathsf{in}(1, \sigma, y)$, which means that the rule has immediately produced the output value y, in which case our derivation tree must be turned into a complete concrete derivation tree with output y. On the other hand, the value may have the form $\mathsf{in}(i + 2, \sigma, \{a, y, f\})$, which is a request to initiate a sub-evaluation. In this case, our derivation tree is turned into the incomplete concrete derivation tree

$$\mathsf{Inc}\langle \delta\, p, x, \langle \mathsf{Init}\langle a, y\rangle\rangle, f\rangle,$$

in which the sub-evaluation of term a with input y has been initiated, and the resumption f is waiting for the sub-derivation $\mathsf{Init}\langle a, y\rangle$ to become complete.

The next two tree expansion rules are similar, but are concerned with the expansion of incomplete concrete derivation trees whose last sub-derivations have become complete. Again, object language blocking is only possible if allowed by the action type σ of the metalanguage term that is being evaluated. Finally, the last rule is a contextual rule: it shows how the last sub-derivation of an incomplete concrete derivation tree can be expanded in place.

There is one more aspect to the semantics of Dops: the translation of concrete derivation trees into *abstract derivation trees*. Abstract derivation trees are defined as certain functions from tree paths to tree nodes consisting of either term/input pairs a, x or output values y. Then, a tree abstraction function abs can be defined in such a way that $\gamma \to \gamma'$ implies that $\mathsf{abs}\,\gamma \subseteq \mathsf{abs}\,\gamma'$. Resumptions are discarded as part of the abstraction process. Then, the meaning of a term/input pair a, x can be defined to be the abstract derivation tree

$$\bigcup\{\, \mathsf{abs}\,\gamma \mid \mathsf{Init}\langle a, x\rangle \to^* \gamma\,\}.$$

The meaning of a term/input pair will be a complete (and finite) derivation tree like the tree of Figure 4, or a blocked incomplete derivation tree like the one of Figure 5, or an infinite incomplete derivation tree like the one of Figure 6.

The implementation of Dops will allow users to construct as much of the meanings of term/input pairs as they desire. At each point in the evaluation of a given term/input pair, the user will be presented with a single node of the abstraction of the current concrete derivation tree, since the whole abstract derivation tree will typically be far too large to display in its entirety. The user will be able to navigate around the abstract derivation tree, and to view as much of the metalanguage values occurring in the tree's nodes as they wish (these values may themselves become too large to display fully). They may also opt to view the resumptions that occur in the underlying concrete derivation tree, which will normally be hidden. There will be various commands for continuing the tree expansion process, causing more of the final meaning to be constructed.

Dops specifications of the following operational semantics have been written: a typing system for the simply typed lambda calculus, substitution-based, big- and small-step semantics for the call-by-value untyped lambda-calculus, and big- and small-step semantics for a simple imperative language. When expressing a small-step semantics in Dops, one will make use of output types involving terms. It is also useful to add an extra layer to a small-step semantics that computes the transitive closure of the original small-step relation. Dops is currently being implemented in Standard ML.

3 Comparison with related work

There are various operational semantics (or logical) frameworks that allow users to evaluate object language term/input pairs [2, 1, 6, 7, 4]. Some of these frameworks support the construction of complete derivation trees in cases when term/input pairs evaluate to output values [1, 6, 4]. But, as far as I know, only D. Berry's Animator Generator [1] supports the incremental construction of derivation trees.

The Animator Generator takes in a deterministic operational semantics for a programming language and generates an animator for that language. The animator incrementally constructs derivation trees, starting from term/input pairs, and displays various views of those trees. One of these views, which Berry calls a "semantic view", displays whole derivation trees. From our point of view, however, the Animator Generator suffers from several deficiencies.

The Animator Generator doesn't do a good job of displaying derivation trees (supporting other kinds of views had much higher priority). The main problem is that it insists on displaying an entire derivation tree in a single window; when the tree becomes large, this makes it very difficult to navigate around the tree. The problem is compounded by the fact that semantic values are also displayed in their entirety. We hope that our approach to displaying derivation trees will work better in practice.

In the Animator Generator's metalanguage, auxiliary tests and functions must be expressed as separate sets of rules. Unfortunately, this means that the side-conditions and auxiliary operations of rules may fail to be total (i.e., may diverge), which can lead to apparent rule blocking that won't be detected by the system. This deficiency is shared by all of the frameworks referred to above, but is avoided in Dops by employing a metalanguage in which all terms converge.

Finally, the tree expansion model of the Animator Generator is much more complicated than our definition of tree expansion (Figure 11) via a small-step semantics on concrete derivation trees.

The incremental construction of derivation trees has also been advocated by Gunter and Rémy [3]. They gave a definition of "partial proofs" in the context of a big-step semantics of a simple programming language. However, they only gave an informal description of how one partial proof can be transformed into another, by “resolving or extending subgoals” in the first.

4 Acknowledgments

Dave Schmidt's interest in viewing the semantics of divergent programs as infinite big-step derivation trees led to this work. (My suggestion to Dave that derivation trees be incrementally constructed using a small-step semantics on derivation trees contributed to the definition of the tree expansion relation in [5].) It is a pleasure to acknowledge stimulating conversations regarding this work with Dave, Brian Howard, Husain Ibraheem and Alan Jeffrey. Thanks are also due to Karen Bernstein, Bob Harper and Flemming Nielson for referring me to related research, and to Andy Pitts for helpful advice on terminology.

References

[1] D. Berry. *Generating Program Animators from Programming Language Semantics*. PhD thesis, Department of Computer Science, University of Edinburgh, 1991. Report number ECS-LFCS-91-163.

[2] P. Borras, D. Clément, T. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. CENTAUR: the system. In *ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 14–24. ACM Press, 1988.

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

[4] P. H. Hartel. LATOS—a Lightweight Animation Tool for Operational Semantics. Technical Report DSSE-TR-97-1, Declarative Systems and Software Engineering Group, Department of Electronics and Computer Science, University of Southampton, 1997.

[5] H. I. Ibraheem and D. A. Schmidt. Partial evaluation of higher-order natural-semantics derivations. In M. Leuschel, editor, *Workshop on Specialization of Declarative Programs and its Applications*, pages 17–28, Port Jefferson, Long Island, NY, 1997.

[6] S. Michaylov and F. Pfenning. Natural semantics and some of its meta-theory in Elf. In *Second International Workshop on Extensions of Logic Programming*, number 596 in Lecture Notes in Artificial Intelligence, pages 299–344, 1991.

[7] M. Pettersson. A compiler for natural semantics. In *Sixth Conference on Compiler Construction*, volume 1060 of *Lecture Notes in Computer Science*, pages 177–191. Springer-Verlag, 1996.