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

With rule-based operational semantics, one has the choice of looking at reduction and evaluation abstractly or concretely. With the abstract approach, one is only interested in outcomes, e.g., that term a evaluates to value v. But one can also work more concretely, looking at the derivation trees that explain, using the semantic rules, how evaluations and reductions arise. Unfortunately, when reduction or evaluation fails, ordinary derivation trees can't be used to explain the failure. E.g., in a transition (single-step) semantics, there is no tree-based explanation for why a term a is blocked. And, in an evaluation (big-step) semantics, there is no explanation of this kind for why a term b diverges. Of course, one might informally employ derivation tree-like notation to explain why a is blocked or b diverges. Such trees would have rightmost branches whose nodes were incomplete (they would lack results). The tree for a would be finite, whereas the tree for b would be infinite.

Formal explanations of term/configuration blocking and divergence (as well as convergence) can be given using possibly infinite *abstract derivation trees* γ of the following kinds, where $\bar{\gamma}$ is a finite sequence of such derivation trees:

$$\mathrm{Comp}(a, x, \bar{\gamma}, y) \qquad \mathrm{Inc}(a, x, \bar{\gamma})$$

A (finite) *completed* tree $\mathrm{Comp}(a, x, \bar{\gamma}, y)$ consists of a term a, an input value x of the kind needed to process a, the sequence of (completed and finite) sub-derivations $\bar{\gamma}$ that were created during the processing of a, and the output y that was finally produced when a's processing terminated. An (possibly infinite) *incomplete* tree $\mathrm{Inc}(a, x, \bar{\gamma})$ is similar, expect that the last sub-derivation in $\bar{\gamma}$ is allowed to be incomplete, and no final value y is given.

Dops is a framework for Deterministic OPerational Semantics that supports the incremental construction of finite approximations to abstract derivation trees. The framework is based on a typed lambda calculus with inductive types, sums and products. For each syntactic construct of a language being modeled, a corresponding rule is specified by giving a λ-term that takes in the sub-terms of an instance of the construct plus the input data that is needed to process the instance, and returns one of the following kinds of values:

(a) an indication that blocking has occurred;

(b) the final value y produced by processing the instance; or

(c) a subgoal of the form (b, z), consisting of a term plus its input data, together with a resumption r that says how the processing of the instance should proceed once (or if!) the sub-derivation consisting of the processing of b with input z becomes complete (the resumption r will consume the output of this sub-derivation).

Abstract derivation trees are constructed using a transition (small-step) semantics involving (finite) *concrete derivation trees* of the following kinds:

$$\mathrm{Init}(a, x) \qquad \mathrm{Comp}(a, x, \bar{\gamma}, y) \qquad \mathrm{Inc}(a, x, \bar{\gamma}, r)$$

One begins with an *initial* tree of the form $\mathrm{Init}(a, x)$, where a is the term to be processed, and x is its input data. The transition semantics then uses the rule for the outermost syntactic construct of a to (a) indicate that blocking has occurred, or (b) transform $\mathrm{Init}(a, x)$ into the complete derivation tree $\mathrm{Comp}(a, x, \langle\rangle, y)$, where the term y is as above, or (c) transform $\mathrm{Init}(a, x)$ into the incomplete tree $\mathrm{Inc}(a, x, \langle \mathrm{Init}(b, z)\rangle, r)$, where the term b, input data z and resumption r are as above. There is also a conditional rule that allows the final sub-derivation of an incomplete tree (e.g., $\mathrm{Init}(b, z)$, above) to be simplified, as well as a rule that invokes a resumption on the output produced by a complete sub-derivation; this can lead to blocking, to the overall tree becoming complete, or to the processing of a new sub-goal, whose output will be consumed by a new resumption.

Because the typed lambda calculus employed by the framework can only express total functions, a derivation tree can only become blocked when blocking has occurred at the object language level, and the possibility of such blocking can always be predicted by examining the types of the λ-term encodings of an operational semantic's rules. When an initial derivation tree $\mathrm{Init}(a, x)$ leads to an infinite sequence of derivation trees, these trees can be put together to form an infinite derivation tree.

I am currently working on an implementation of Dops, and expect to be ready to demonstrate the framework at the workshop. The framework will allow the user to build and view as much of the abstract derivation tree corresponding to an object language term and associated input as the user desires to see. I've already written Dops specifications of the following operational semantics: a typing system for the simply typed lambda calculus, evaluation semantics (both substitution- and environment-based) and transition semantics for the call-by-value, untyped lambda calculus, and evaluation and transition semantics for a simple imperative language.

Acknowledgment

It was Dave Schmidt's interest in viewing the semantics of diverging programs as infinite big-step derivation trees that led to this research.