A Logical Correspondence between Natural Semantics and Abstract Machines

Robert J. Simmons
Carnegie Mellon University
Email: rjsimmon@cs.cmu.edu

Ian Zerny
Aarhus University
Email: zerny@cs.au.dk

***Abstract*—We present a logical correspondence between natural semantics and abstract machines. This correspondence enables the mechanical and fully-correct construction of an abstract machine from a natural semantics. Our *logical correspondence* mirrors the Reynolds *functional correspondence* but places it in a logical setting, as both semantics are encoded in a substructural logical framework.**

I. Introduction

The literature contains numerous semantic specifications and therefore many proposals for relating them. These relations are stated using a diversity of methods and methodologies. To the best of the authors' knowledge, only one, the Reynolds functional correspondence [1], [2] has seen repeated use outside the work of its inventors [3], [4]. Our goal [5], [6] is to develop a logical counterpart. We want it to be formal, mechanizable and, like the functional correspondence, widely applicable.

In this paper, we describe a method for relating natural semantics with abstract machines within a common logical framework. We take advantage of the fact that *substructural logic* provides an adequate specification language for different types of specifications.

We motivate our work by recalling how divergence and failure interact with natural semantics and abstract machine semantics. Expressions are λ-terms in addition to an extra nonsense term, here denoted by junk. As usual, syntactic values are λ-abstractions, and contexts are a list of application frames terminated by the empty frame halt:

$$\begin{aligned} e &::= x \mid \lambda x.e \mid e_1\, e_2 \mid \mathsf{junk} \\ v &::= \lambda x.e \\ k &::= \mathsf{halt} \mid k; \square\, e_2 \mid k; (\lambda x.e)\, \square \end{aligned}$$

We give two semantics for call-by-value (CBV) evaluation: Figure 1 defines a big-step semantics in the form of a natural semantics [7], [8]; Figure 2 defines a small-step semantics in the form of an abstract machine. Common to these specifications is the appearance of being specified by the same logical tool: *inductive definitions*. Despite this, the two specifications have a very different character.

Consider the term $\omega = (\lambda x.x\,x)\,(\lambda x.x\,x)$. The abstract machine can characterize developments of ω. The natural semantics cannot find a v such that $\omega \Downarrow v$ is derivable. As a small-step semantics the abstract machine can characterize how $((\lambda x.x\,x)\,\mathsf{junk})$ goes wrong. As a big-step semantics the natural semantics cannot find a v such that $((\lambda x.x\,x)\,\mathsf{junk}) \Downarrow v$ is derivable. Thus, working with the natural semantics, we cannot distinguish safe programs that do not terminate from programs that go wrong. This is a known obstacle to proving type soundness with natural-semantics specifications.

$$\frac{}{\lambda x.e \Downarrow \lambda x.e} \qquad \frac{e_1 \Downarrow \lambda x.e \quad e_2 \Downarrow v_2 \quad e[v_2/x] \Downarrow v}{e_1\, e_2 \Downarrow v}$$

Fig. 1. A natural semantics for CBV evaluation

$$\frac{}{k \triangleright \lambda x.e \mapsto k \triangleleft \lambda x.e} \qquad \frac{}{k \triangleright e_1\, e_2 \mapsto (k; \square\, e_2) \triangleright e_1}$$

$$\frac{}{(k; \square\, e_2) \triangleleft \lambda x.e \mapsto (k; (\lambda x.e)\, \square) \triangleright e_2}$$

$$\frac{}{(k; (\lambda x.e)\, \square) \triangleleft v_2 \mapsto k \triangleright e[v_2/x]}$$

Fig. 2. An abstract machine semantics for CBV evaluation

II. Natural semantics as logic programs

Our approach to this problem reaches back to some of the original work on natural semantics, the TYPOL compiler that translated natural semantics specifications to logic programs in Prolog [7]. The operational interpretation introduced by this compilation process is only implicitly present in the original natural semantics presentation. We derive operational meaning from Figure 1 by systematically describing a search procedure that attempts to find an expression v and derivation $e \Downarrow v$ given an expression e.

- If $e = \lambda x.e'$, derive $\lambda x.e' \Downarrow \lambda x.e'$ with the first rule.
- If $e = e_1\, e_2$, attempt to derive $e_1\, e_2 \Downarrow v$ using the second rule:
  1) Search for a v_1 such that $e_1 \Downarrow v_1$ is derivable.
  2) Assert that $v_1 = \lambda x.e'$ for some e'; fail if it is not.
  3) Search for a v_2 such that $e_2 \Downarrow v_2$ is derivable.
  4) Let $e'' = e'[v_2/x]$
  5) Search for a v such that $e'' \Downarrow v$ is derivable.
  6) If we succeed, derive $e_1\, e_2 \Downarrow v$ with the second rule.

Our operationalization transformation makes this implicit search process explicit. It is applicable to a significant fragment of Horn clause logic programs (those with a reasonable input-output interpretation, the so-called *well-moded* programs).

Two similar lines of work by Hannan and Miller [9] and Ager [10] also derive abstract machines by representing a

natural semantics as a logical specification, in λProlog and L-attributed grammars respectively, and then applying logical transformations. Our work follows this tradition of assigning operational behavior by means of proof search.

III. Abstract machines as logic programs

Given as input a standard *judgments as types* encoding of the natural semantics in Figure 1 [11], our operationalization transformation produces an encoding of the abstract machine semantics in Figure 2. This encoding of Figure 2 is not in the standard judgments as types encoding, however. Instead, we get a *substructural operational semantics*, an encoding of the transition system as rewriting rules in ordered logic [12].

The states of a substructural operational semantics are ordered sequences of propositions Δ, contexts in ordered linear logic. We interpret the ordered logic proposition $\mathsf{a} \bullet \mathsf{b} \rightarrowtail \mathsf{c} \bullet \mathsf{d}$ as a local rewriting rule that allows us to transition from a state $\Delta_1\ \mathsf{a}\ \mathsf{b}\ \Delta_2$ to a state $\Delta_2\ \mathsf{c}\ \mathsf{d}\ \Delta_2$. (The connective $P \bullet Q$ is conjunction in ordered logic, and the connective $P \rightarrowtail Q$ is implication.) We say there is a *trace* $\Delta \rightsquigarrow^* \Delta'$ if we can rewrite Δ to Δ' with a series of transitions.

The general idea behind operationalization is that a trace $(\mathsf{eval}(e)\ \Delta \rightsquigarrow^* \mathsf{retn}(v)\ \Delta)$ indicates the presence of a derivation $e \Downarrow v$, so the left-most proposition in Δ, if any, represents a continuation that spawned the evaluation of e and needs to receive a v to continue. In general, operationalization takes the encoding of a single judgment like $e \Downarrow v$ and defines (at minimum) an evaluation predicate $\mathsf{eval}(e)$ and a return predicate $\mathsf{retn}(v)$.

Describing the proof search behavior when $e = \lambda x.e'$ is simple. We start in the state $\mathsf{eval}(\lambda x.e)\ \Delta$. Because $\lambda x.e \Downarrow \lambda x.e$ is immediately derivable, we can step immediately to the state $\mathsf{retn}(\lambda x.e)\ \Delta$. This is captured by the rule evlam:

evlam: $\mathsf{eval}(\lambda x.e) \rightarrowtail \mathsf{retn}(\lambda x.e)$.

(For brevity's sake, we are omitting a careful treatment of the term language; see [5] for details.)

Dealing with proof search $e = e_1\, e_2$ requires extra machinery. If we are in a state $\mathsf{eval}(e_1\, e_2)\ \Delta$, then the search procedure from Section II indicates that we first should search for a v_1 such that $e_1 \Downarrow v_1$. This means picking a Δ' and trying to find a trace $\mathsf{eval}(e_1)\ \Delta' \rightsquigarrow^* \mathsf{retn}(v_1)\ \Delta'$. We introduce a new predicate $\mathsf{cont1}(e_2)$ that stores e_2 in on the top of the continuation stack, letting $\Delta' = \mathsf{cont1}(e_2)\ \Delta$, while we attempt to evaluate e_1 to a value.

evapp: $\mathsf{eval}(e_1\, e_2) \rightarrowtail \mathsf{eval}(e_1) \bullet \mathsf{cont1}(e_2)$.

If we ever complete a trace of this form:

$$\mathsf{eval}(e_1)\ \mathsf{cont1}(e_2)\ \Delta \rightsquigarrow^* \mathsf{retn}(v_1)\ \mathsf{cont1}(e_2)\ \Delta$$

then we know there is a proof of $e_1 \Downarrow v_1$. Then we proceed to check that v_1 has the form $\lambda x.e$ and evaluate e_2 to a value, storing the body of the function $\lambda x.e$ in another new predicate $\mathsf{cont2}(\lambda x.e)$.

evapp1: $\mathsf{retn}(\lambda x.e) \bullet \mathsf{cont1}(e_2) \rightarrowtail \mathsf{eval}(e_2) \bullet \mathsf{cont2}(\lambda x.e)$.

Finally, once a value v_2 returns to the left of the proposition $\mathsf{cont2}(\lambda x.e)$, we know that in order to prove $e_1\, e_2 \Downarrow v$ it suffices to prove $e[v_2/x] \Downarrow v$.

evapp2: $\mathsf{retn}(v_2) \bullet \mathsf{cont2}(\lambda x.e) \rightarrowtail \mathsf{eval}([v_2/x]e)$.

Thus, we can encode the search procedure from Section II as a transition system in ordered logic. The connection between this four-rule specification and the transition relation in Figure 2 is witnessed by a translation from states $k \triangleleft v$ and $k \triangleright e$ to ordered contexts. The stack frames $\square\, e_2$ and $(\lambda x.e)\, \square$ are, respectively, associated with the propositions $\mathsf{cont1}(e_2)$ and $\mathsf{cont2}(\lambda x.e)$. This interpretation treats Figure 2 not as an inductive definition but as a direct encoding of a transition system in ordered logic.

IV. Conclusion

We have shown how to take a natural semantics and make its implicit operational interpretation explicit as a substructural operational semantics. This instance of the general operationalization transformation also formally connects a natural semantics specification (Figure 1) and an abstract machine semantics specification (Figure 2). The general correctness proof for this operationalization establishes that $\mathsf{eval}(e) \rightsquigarrow^* \mathsf{retn}(v)$ if and only if $e \Downarrow v$ [5].

We have implemented operationalization in SML and have successfully applied the transformation to a variety of examples, including but not limited to a number of larger natural semantics specifications.

References

[1] J. C. Reynolds, "Definitional interpreters for higher-order programming languages," in *Proc. of 25th ACM National Conference*, 1972, pp. 717–740.

[2] M. S. Ager, D. Biernacki, O. Danvy, and J. Midtgaard, "A functional correspondence between evaluators and abstract machines," in *Proc. of PPDP'03*, D. Miller, Ed. ACM Press, Aug. 2003, pp. 8–19.

[3] I. Sergey and D. Clarke, "A correspondence between type checking via reduction and type checking via evaluation," *IPL*, vol. 112, no. 13-20, pp. 13–20, 2011.

[4] K. Anton and P. Thiemann, "Typing coroutines," in *Trends in Functional Programming*, ser. LNCS, R. Page, Z. Horvth, and V. Zsk, Eds. Springer Berlin Heidelberg, 2011, vol. 6546, pp. 16–30.

[5] R. J. Simmons, "Substructural logical specifications," Ph.D. dissertation, School of Computer Science, Computer Science Department, Carnegie Mellon University, Nov. 2012.

[6] I. Zerny, Ph.D. dissertation, Dept. of Comp. Sci., Aarhus University, forthcoming.

[7] D. Clément, J. Despeyroux, T. Despeyroux, L. Hascoet, and G. Kahn, "Natural semantics on the computer," INRIA, Tech. Rep. 416, Jun. 1985.

[8] G. Kahn, "Natural semantics," in *Proc. of STACS'87*, ser. LNCS, F.-J. Brandenburg, G. Vidal-Naquet, and M. Wirsing, Eds., no. 247. Springer-Verlag, Feb. 1987, pp. 22–39.

[9] J. Hannan and D. Miller, "From operational semantics to abstract machines," in *Special issue on the 1990 ACM Conference on Lisp and Functional Programming*, ser. Mathematical Structures in Computer Science, Vol. 2, No. 4, M. Wand, Ed. Cambridge University Press, Dec. 1992, pp. 415–459.

[10] M. S. Ager, "From natural semantics to abstract machines," in *LOPSTR 2004, revised selected papers*, ser. LNCS, S. Etalle, Ed., no. 3573. Springer, Aug. 2004, pp. 245–261.

[11] R. Harper, F. Honsell, and G. Plotkin, "A framework for defining logics," *Journal of the ACM*, vol. 40, no. 1, pp. 143–184, 1993.

[12] F. Pfenning and R. J. Simmons, "Substructural operational semantics as ordered logic programming," in *Proc. of LICS'09*, 2009, pp. 101–110.