Partial Evaluation of Higher-Order Natural-Semantics Derivations

Husain I. Ibraheem and David A. Schmidt
Kansas State University*

Abstract. We use on-line partial-evaluation technology to generate partial derivations from natural (big-step) semantics definitions. First, we reformat the big-step rules into a small-step form, and we extend big-step semantics to be *higher-order*, that is, we allow data structures like closures to contain partial derivations. Next, we apply a supercompilation algorithm to the transformed semantics, thereby producing partial derivations for programs and exposing the programs' control structures. The resulting system can perform static analysis on incomplete or "modular" programs and can do double duty as a link-time static analyzer; these are significant steps towards modular program analysis.
Keywords: on-line partial evaluation, natural semantics, supercompilation, static analysis, modular program analysis.

1 Introduction

Perhaps the most general form of declarative language is a deductive operational semantics framework, since the semantics of other languages can be encoded within deductive operational semantics in side-effect-free fashion. Big-step (*natural* [8, 18]) semantics definitions are particularly attractive, because their "semicompositional structure" lends itself to compact, understandable derivations [17]. Such derivations are facilitated even more by the inherent two-dimensional nature of big-step semantics derivations and its use of both inherited (downwards flowing) and synthesized (upwards flowing) attributes. However, the latter characteristics do not suit on-line partial evaluation[1] algorithms that are oriented towards a term-rewriting (small-step) framework, e.g. [10, 12, 13, 20, 27, 28, 31, 33]. And simulations of two-dimensional big-step derivations by term rewriting on one-dimensional representations destroy the beauty and many of the advantages of the big-step framework [15, 23].

In this paper, we develop an on-line partial evaluator that lets a big-step semantics be evaluated in two-dimensional, small-step style, in the spirit of Jones [16] and Stoughton [29]. Also, we extend big-step semantics so that data-structure values (e.g., closures and

*Computing and Information Sciences Department, 234 Nichols Hall, Manhattan, KS 66506 USA. schmidt@cis.ksu.edu. Phone: +1-913-532-6350; Fax: +1-913-532-7353. Ibraheem is supported by a Kuwait University scholarship; Schmidt is supported by an NSF CCR-9633388.

[1]We use "partial evaluation" in a general sense to denote techniques achieving the effect of specialization, e.g. supercompilation, deforestation, partial deduction, etc.

continuations) can hold partial subderivations. The result is a term-rewriting, second-order, big-step semantics variant that is well suited for on-line partial evaluation.

We demonstrate the transformation techniques on the big-step semantics of a higher-order functional language and we use Glück and Sørensen's supercompilation algorithm [14] to partially evaluate derivations of functional programs. Indeed, an extension of the example language and partial evaluator have been implemented in Sictus Prolog and have been applied to a variety of standard examples.

Our motivations for these efforts are two-fold:

- We desire a semantically sound and general technique for exposing the semantic control-flow graph of a program, much like that employed by Gallagher and Lafave [11], so that static analysis (especially abstract interpretation [4, 5, 16, 24] and model checking [2, 6, 9]) can be conducted on top of the resulting graph.
- We desire a foolproof methodology for incremental static analysis, along the lines of Codish, Debray, and Giacobazzi [3], where incomplete or "modular" programs can be analyzed as best as possible, and the residuals of the partially analyzed programs can be combined and further analyzed at "link-time" stages.

Only the combination of a program-point-based operational semantics and an on-line partial-evaluation-based analysis engine appears to be general enough to meet our needs.

In the remainder of the paper, we describe our transformations on big-step semantics definitions, we review fundamentals of process-tree-based supercompilation, and we present an example of supercompilation of semantic derivations. We finish with a few remarks regarding the status of the implementation.

2 Big-Step Semantics in Small-Step Format

The usual reading of a Kahn-style, big-step (*natural*) semantics definition [18] states that formulas are triples, $\rho \vdash e \downarrow v$, where $\rho \in \mathit{Environment}$, $e \in \mathit{Expression}$, and $v \in \mathit{Value}$. A derivation is an inductively defined "tree" of such triples. Unfortunately, this interpretation prohibits reasoning about divergent computations, and it also limits the application of on-line partial-evaluation algorithms which are oriented towards term-rewriting or "small-step" systems [19, 22] rather than attribute-grammar-like big-step systems.

In this paper, we view the triple, $\rho \vdash e \downarrow v$, intuitively, as it was once intended: The pair, $\rho \vdash e$, converges in some sequence of rewriting steps to v, that is, $(\rho \vdash e) \rightarrow^* v$. Therefore, a derivation starting from $\rho \vdash e$ consists of multiple small steps that either converge to v or diverge.

Given this view, we see that a unique aspect of big-step semantics is the two-dimensional topology in which the small steps are arranged. For example, the computation of $\rho \vdash \texttt{1} + (\texttt{2} + \texttt{3})$ into 6 goes:

$$\begin{tikzcd}[column sep=small]
 & & \rho \vdash \texttt{1} + (\texttt{2} + \texttt{3}) \arrow[dll] \arrow[d] \arrow[drr] & & \\
\rho \vdash \texttt{1} \arrow[d] & & \rho \vdash (\texttt{2} + \texttt{3}) \arrow[dl] \arrow[dr] \arrow[drr] & & 6 \\
1 & \rho \vdash \texttt{2} \arrow[d] & & \rho \vdash \texttt{3} \arrow[d] & 5 \\
 & 2 & & 3 &
\end{tikzcd}$$

Underneath the parent expression, `1+(2+3)`, one finds both the steps for the "structural" derivations for `1` and `2+3` as well as the "value", `6`.

Those arrows of form, $\rightarrow v$, for $v \in \mathit{Value}$ (e.g., $\rightarrow 6$), are called *value arrows.* Frankly, it does not matter how the value arrows are positioned, so we draw them branching downwards from their parent sequents—thus, the rightmost arrow of a convergent derivation must be a value arrow.

Our job is to generate on-line (with partial evaluation) derivations like the one above; we do so incrementally, adding new nodes of the forms, $\rho \vdash e$ and v, to the derivation tree, step by step. A big-step inference rule, such as

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

encodes two rewriting rules, which we might write diagrammatically as follows:

$$\begin{tikzcd}[column sep=small]
\rho \vdash e_1 + e_2 \arrow[r, mapsto] & & \rho \vdash e_1 + e_2 \arrow[dl] \arrow[dr] & \\
 & \rho \vdash e_1 & & \rho \vdash e_2
\end{tikzcd}
\qquad
\begin{tikzcd}[column sep=small]
 & \rho \vdash e_1 + e_2 \arrow[dl] \arrow[dr] & \arrow[r, mapsto] & & \rho \vdash e_1 + e_2 \arrow[dl] \arrow[d] \arrow[dr] & \\
d_1 \downarrow v_1 & & d_2 \downarrow v_2 & d_1 \downarrow v_1 & d_2 \downarrow v_2 & \mathit{plus}(v_1, v_2)
\end{tikzcd}$$

These two rules are rewriting rules upon *trees*, rather than sequents, hence the tree structure must be explicit [29]. The notation, $d \downarrow v$, asserts that derivation d converges and the value arrow attached to its root points to v. These rules were used to derive the tree for $\rho \vdash \texttt{1} + (\texttt{2} + \texttt{3})$ seen earlier. The resemblance of these rules to the reduction strategies used by Horn-clause programming [21] and Typol/Centaur [1, 8] is obvious, but note the handling of synthesized attributes in the second rule—there are no substitutions or unifications—and once a node is added to the derivation it stays unchanged.

3 Higher-Order Derivations

Big-step semantics is inherently "first-order": Closure and continuation values are inert data structures, rather than enabled, potential derivations. For example, it is impossible to employ the usual big-step rule for lambda abstraction,

$$\rho \vdash \lambda x.\, e \downarrow \langle \rho, x, e \rangle$$

to derive $\rho \vdash \lambda\texttt{x}.\texttt{x} + (\texttt{1} + \texttt{2}) \downarrow \langle \rho, \texttt{x}, \texttt{x} + 3 \rangle$, because the closure is "inert." To alleviate this problem, we allow derivations to appear within data-structure values like closures.

Given that the pair, $\rho \vdash e$, is a simple example of (an incomplete) derivation, we revise the above semantics rule to the following:

$$\rho \vdash \lambda x.\, e \downarrow \boldsymbol{\lambda} A.\, (\rho + (x \mapsto A) \vdash e)$$

That is, a closure is a mapping from an argument, A, to a (possibly incomplete) derivation.[2]

Figure 1 gives the syntax and big-step rules we use for a small functional language with recursive lambda-abstractions. Remember that the notation, $d \downarrow v$, should be

$$
\begin{array}{lrcl}
e \in \mathit{Expression} & e & ::= & x \;\mid\; \mathtt{rec}\, f\, x.e \;\mid\; e_1 @ e_2 \\
x, f \in \mathit{Identifier} & & & \\
d \in \mathit{Derivation} & d & ::= & v \;\mid\; \begin{array}{c} \rho \vdash e \\ \swarrow \quad \searrow \\ d_1 \;\cdots\; d_n \end{array}, \quad n \geq 0 \\
\rho \in \mathit{Environment} & \rho & ::= & [x \mapsto v_x]_{x \in \mathit{Identifier}} \\
v \in \mathit{Value} & v & ::= & \boldsymbol{\lambda} A.\, d \;\mid\; A \\
A \in \mathit{MetaVariable} & & &
\end{array}
$$

$$\rho \vdash x \downarrow \rho_x \qquad \rho \vdash \mathtt{rec}\, f\, x.e \downarrow c\ell, \text{ where } c\ell = \boldsymbol{\lambda} A.\, \rho + (f \mapsto c\ell) + (x \mapsto A) \vdash e$$

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

Figure 1: Big-step semantics for functional language

read as "derivation d converges and its value arrow points to v." The inference rule for function application, $e_1 @ e_2$, encodes these three rewriting rules:

$$
\rho \vdash e_1 @ e_2 \;\longmapsto\; \begin{array}{c} \rho \vdash e_1 @ e_2 \\ \swarrow \qquad \searrow \\ \rho \vdash e_1 \qquad \rho \vdash e_2 \end{array}
\qquad\qquad
\begin{array}{c} \rho \vdash e_1 @ e_2 \\ \swarrow \qquad \searrow \\ d_1 \downarrow \boldsymbol{\lambda} A.\, d' \qquad d_2 \downarrow v_2 \end{array} \;\longmapsto\; \begin{array}{c} \rho \vdash e_1 @ e_2 \\ \swarrow \quad \downarrow \quad \searrow \\ d_1 \downarrow \boldsymbol{\lambda} A.\, d' \quad d_2 \downarrow v_2 \quad [v_2/A]d' \end{array}
$$

$$
\begin{array}{c} \rho \vdash e_1 @ e_2 \\ \swarrow \quad \downarrow \quad \searrow \\ d_1 \downarrow v_1 \quad d_2 \downarrow v_2 \quad d \downarrow v \end{array} \;\longmapsto\; \begin{array}{c} \rho \vdash e_1 @ e_2 \\ \swarrow \quad \downarrow \quad \downarrow \quad \searrow \\ d_1 \downarrow v_1 \quad d_2 \downarrow v_2 \quad d \downarrow v \quad v \end{array}
$$

The usual implementation of the semantics in Figure 1 evaluates a derivation to "weak normal form"—bodies of closures are not evaluated until an argument is available. This is shown in Figure 2(a). But the mechanisms now exist to evaluate closure bodies, if desired; this is seen in Figure 2(b), where the closure body, $\rho_1 \vdash \mathtt{f@x}$, is developed. The latter derivation is potentially infinite, but the overall derivation has a value, nonetheless, and the rewriting rules are confluent.

Since we can conduct small-step-style computations of big-step derivations and we can evaluate closure bodies, we next apply a partial-evaluation methodology to our system.

[2] We have proved a congruence theorem that shows that derivations in the "higher-order" semantics are the same as those in the classical semantics up to the forgetting of the subderivations within the closure bodies.

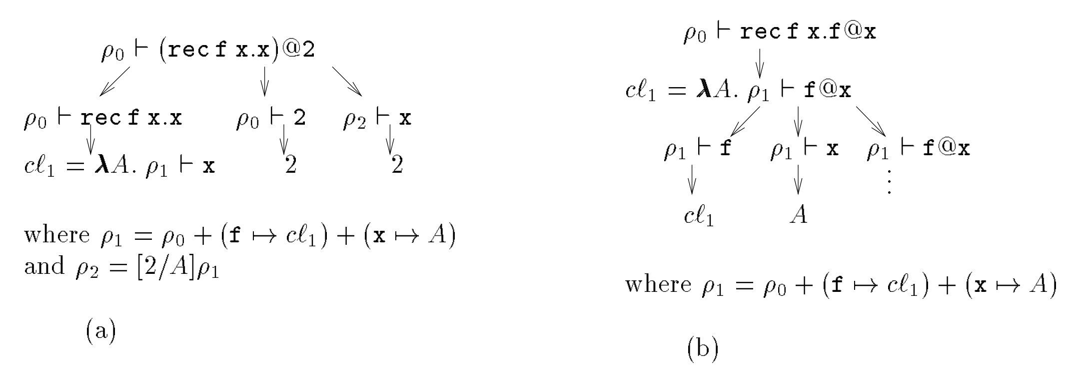

Figure 2: Example derivations

4 Partial deductions using supercompilation

As hinted by Figure 2(b), partial evaluation might be used to generate nontrivial derivation-tree structure in the presence of unknowns. We have found Glück and Sørensen's formulation [14] of Turchin's supercompilation algorithms [30, 31, 32] useful, because it performs specialization and deforestation simultaneously *on-line* [25, 14, 28]. Here is short review of the framework as presented in [14]: A source program consists of a set of first-order, equationally defined functions and an expression to be simplified. The supercompilation algorithm expands the expression by unfolding and specializing until there is danger of an infinite expansion. Then, a *fold*, a *generalization*, or a *split* is performed to prevent divergence.

To monitor the expression's expansion, a trace, called a *process tree*, is maintained. Figure 3(a) shows an incomplete process tree for a small program. If a newly created

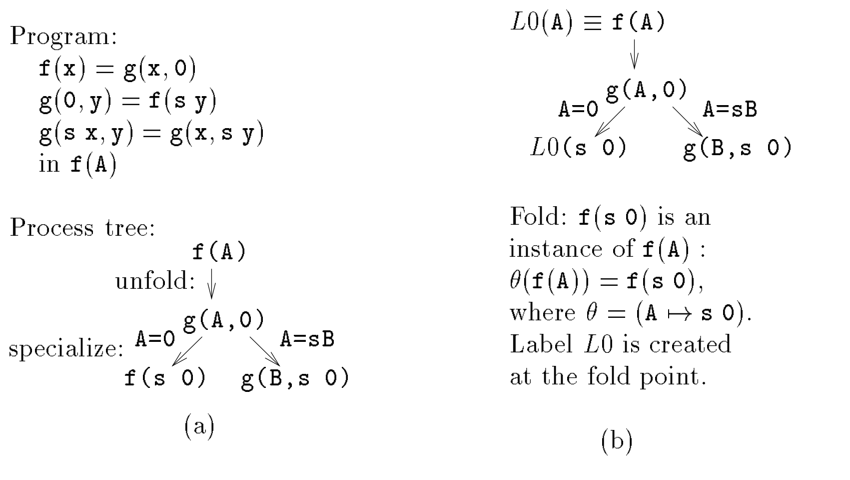

Figure 3: Process tree and a fold step

leaf is an instance of a node seen earlier in the path to the leaf, then a fold must occur. Figure 3(b) shows a fold. If a node in a path does not instantiate to but is nonetheless embedded within a newly created leaf, then a generalization is necessary. Figure 4(a) shows a generalization—notice how part of the process tree is removed and replaced by the *let*-node. Finally, a split can arise if a node embeds into a leaf in such a way that

$$L0(\mathtt{A}) \equiv \mathtt{f(A)}$$
$$\downarrow$$
$$\mathit{let}\ \mathtt{C} = \mathtt{0}\ \mathit{in}\ \mathtt{g}(\mathtt{A},\mathtt{C})$$

Generalize: $\mathtt{g}(\mathtt{A},\mathtt{0})$ embeds in $\mathtt{g}(\mathtt{B},\mathtt{s\ 0})$.Choose $\mathtt{g}(\mathtt{A},\mathtt{C})$ as the generalization, because
$\theta_1(\mathtt{g}(\mathtt{A},\mathtt{C})) = \mathtt{g}(\mathtt{A},\mathtt{0})$,
where $\theta_1 = (\mathtt{C} \mapsto 0)$;
and $\theta_2(\mathtt{g}(\mathtt{A},\mathtt{C})) = \mathtt{g}(\mathtt{B},\mathtt{s\ 0})$,
where $\theta_2 = (\mathtt{A} \mapsto \mathtt{B}, \mathtt{C} \mapsto \mathtt{s\ 0})$

(a)

$$L0(\mathtt{A}) \equiv \mathtt{f(A)}$$
$$\downarrow$$
$$\mathit{let}\ \mathtt{C} = \mathtt{0}\ \mathit{in}\ L1(\mathtt{A},\mathtt{C}) \equiv \mathtt{g}(\mathtt{A},\mathtt{C})$$
$$\swarrow_{\mathtt{A=0}} \qquad \searrow^{\mathtt{A=sD}}$$
$$L0(\mathtt{s\ C}) \qquad L1(\mathtt{D,s\ C})$$

Residual program:
$\mathtt{f(A)}$ is $L0(\mathtt{A})$,
where $L0(\mathtt{A}) = \mathit{let}\ \mathtt{C} = 0\ \mathit{in}\ L1(\mathtt{A},\mathtt{C})$
$L1(\mathtt{0},\mathtt{C}) = L0(\mathtt{s\ C})$
$L1(\mathtt{s\ D},\mathtt{C}) = L1(\mathtt{D},\mathtt{s\ C})$

(b)

Figure 4: Generalization step and results of supercompilation

there is no common superstructure. Since we do not use splits in our system, we will not develop them here.

The process tree that results from the small example is displayed in Figure 4(b): unfolding steps are collapsed, and specializations, folds, and generalizations generate residual code. Since the example is deliberately pathological, the residual program is a mere copy of the original.

When an embedding arises, one says that a "whistle" has blown. The whistle temporarily suspends computation while the process tree is repaired with a fold or generalization. Glück and Sørensen proved that the repairs motivated by the whistle ensure termination of process tree creation. The key technical result is due to Kruskal, who observed that any infinite term rewriting sequence, $t_0 \to t_1 \to \cdots \to t_i \to \cdots$, must contain two terms, t_j, t_k, $j < k$, such that t_j embeds in t_k, written $t_j \trianglelefteq t_k$ [7]. The definition of term embedding is given in Figure 5.

Variables:	Diving:	Coupling:
$A \trianglelefteq A'$	$\dfrac{s \lhd t_i}{s \trianglelefteq h(t_1, ..., t_i, ..., t_n)}$	$\dfrac{s_1 \lhd t_1, \cdots, s_n \lhd t_n}{h(s_1, ..., s_n) \trianglelefteq h(t_1, ..., t_n)}$

Figure 5: Definition of term embedding

Although a process tree looks like a big-step semantics derivation, it differs because it is a history of all the program configurations that appeared starting from the initial one. Therefore, a process tree for a big-step semantics derivation should be a tree of derivation trees. But such a process tree would be too ungainly to implement, and it ignores the crucial feature that a big-step derivation grows "monotonically," that is, nodes are added to the derivation, and none are ever rearranged or removed. Because of the latter property, the derivation itself can serve as the process tree and the whistle criterion can be applied to the derivation's nodes, namely, sequents of the form $\rho \vdash e$.

Yet another benefit of big-step semantics definitions is that they are *semicompositional*,[3] that is, the e-component of every sequent, $\rho \vdash e$, is a subphrase of the program, e_0, that appears in the root node sequent, $\rho_0 \vdash e_0$. Hence, there are at most finitely many distinct such e, and it follows that the whistle need "blow" only when one encounters a leaf, $\rho_i \vdash e_i$, and in the path that leads to the leaf, there exists a node, $\rho \vdash e_i$ such that $\rho \trianglelefteq \rho_i$.[4]

5 An Example

We now apply the supercompilation methodology on a small program. Figure 6 provides semantics rules for unary arithmetic (where v' is the successor of v). Given these rules,

$$\frac{\rho \vdash e \downarrow v}{\rho \vdash \mathtt{s}\ e \downarrow v'} \qquad \frac{\rho \vdash e_0 \downarrow 0 \quad \rho \vdash e_1 \downarrow v}{\rho \vdash \mathtt{if0}\ e_0\ e_1\ e_2 \downarrow v} \qquad \frac{\rho \vdash e_0 \downarrow v' \quad \rho \vdash e_2 \downarrow v}{\rho \vdash \mathtt{if0}\ e_0\ e_1\ e_2 \downarrow v}$$

Figure 6: Rules for successor and conditional

the process tree/derivation for the program `rec f x.if0 (x) (x) (f@(s x))` begins as seen in Figure 7(a).

We employ the equivalent of leftmost-outermost evaluation, hence we attempt to evaluate the body of the closure, $c\ell_1$. Since the test cannot be decided, we could halt at this point with an uninteresting residual derivation. But it proves more useful to construct a residual `if`-node, denoted by $\underline{\mathit{if0}}$, and *drive* into the arms of the conditional. Thus, the small-step rules are enriched with residuals, for each of the operators in the source, as intermediate values. The result appears in Figure 7(b), where a fold point, named $L0$, has been introduced.

Next, consider the simple partial derivation of $\mathtt{f@0}'$ in Figure 8(a). One might specialize the leftmost leaf, but the result, displayed in Figure 8(b), introduces only a residual substitution. When we instantiate F in Figure 8(b) by the residual derivation from Figure 7(b), we obtain the derivation in Figure 9.

Now that the two programs are linked together, partial evaluation can proceed further: The residual, $\underline{\mathit{if0}}$, can be *executed*; it selects the *else*-arm, and the fold point, $L0(0'')$,

[3]This term is due to Neil Jones [17].

[4]The definition of $\trianglelefteq$ is extended pointwise for environments, which is safe because the number of identifiers in a given program must be finite.

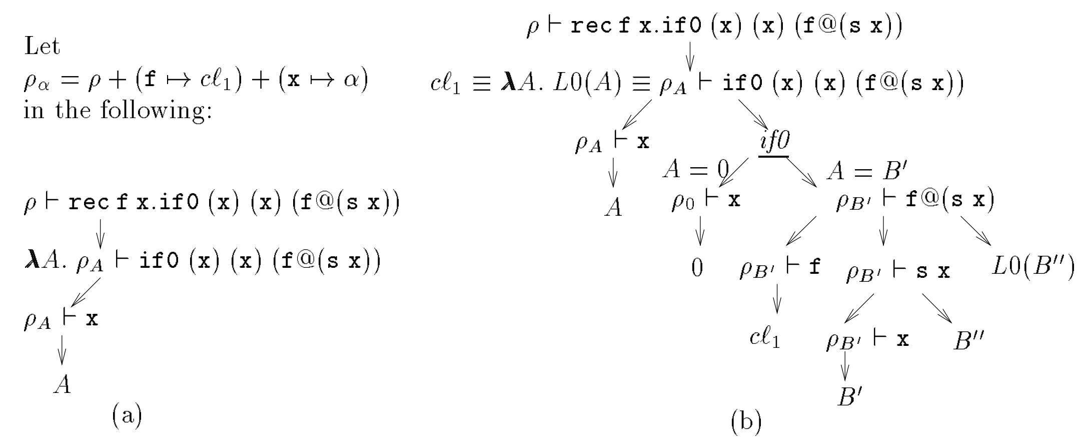

Figure 7: Partial evaluation of $\mathtt{rec\,f\,x.if0\,(x)\,(x)\,(f@(s\,x))}$

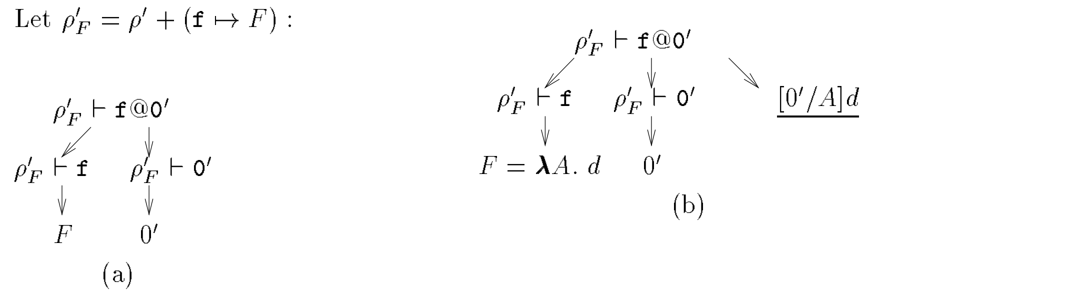

Figure 8: Partial evaluation of $\mathtt{f@0'}$.

immediately unfolds to $\rho_{0''} \vdash \mathtt{if0\,(x)\,(x)\,(f@(s\,x))}$, which blows the whistle, since this new leaf can be generalized with its ancestor node $\rho_{0'} \vdash \mathtt{if0\,(x)\,(x)\,(f@(s\,x))}$. The subtree whose root is the latter sequent is removed and replaced by the generalization node, *let* $C = 0$ *in* $\rho_{C'} \vdash \mathtt{if0\,(x)\,(x)\,(f@(s\,x))}$. Driving restarts from the generalization node. The unfolding that follows builds the tree in Figure 10, where a new fold point, $L1$, terminates the driving. The final derivation exposes program's control structure without compromising its data flow and indicates that only the *else*-arm of the conditional is utilized.

6 Implementation

We have implemented the variant of the Glück and Sørensen supercompilation algorithm specifically for the big-step semantics definition of a higher-order functional language with arithmetic and conditionals. The system is implemented as 600 lines of Sicstus Prolog. A more general-purpose system, one that accepts as input a big-step semantics

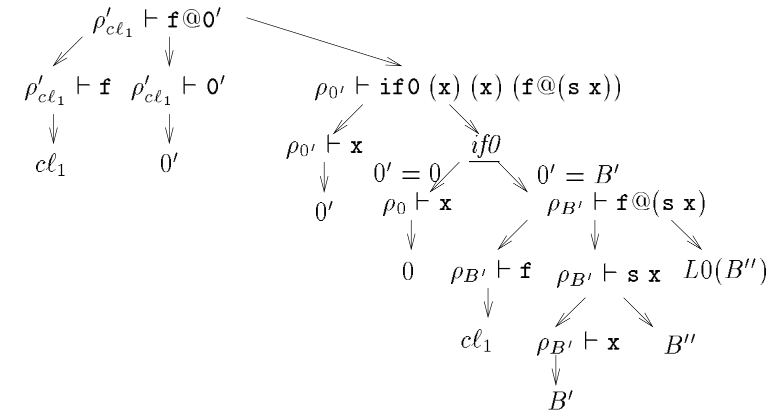

Figure 9: Linking the two residual derivations

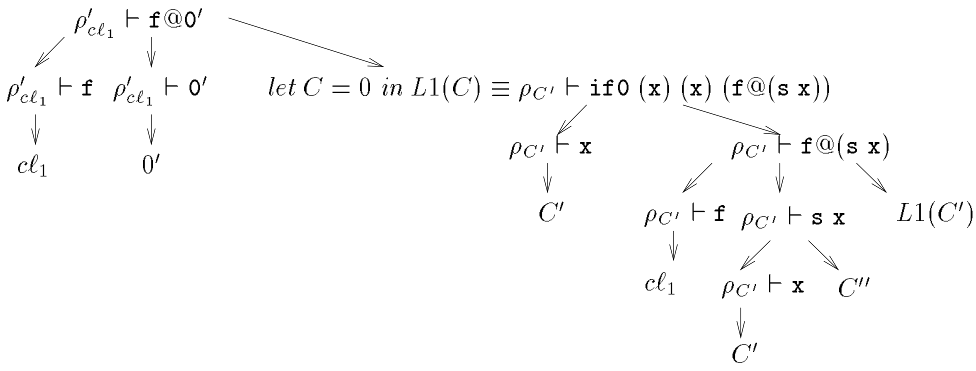

Figure 10: The derivation for the linked program

definition for an arbitrary language, is being planned.

For simplicity, nodes in the derivation tree are generated in a leftmost-outermost fashion. As they are processed, nodes are marked as "fresh," "incompletely processed," or "processed completely" (convergent), and value attributes are passed up the tree as if they were synthesized attributes in an attribute grammar. This simplifies the selection of the rewriting rules while preserving the methodology described in this paper.

As seen in the example in the previous section, if a closure body contains a partial derivation and the closure is applied to an argument, the closure body is copied, instantiated, and driving resumes at the leaves of the copied closure body.

Embedding (whistle) checks are done only when an expression, e, repeats in two sequents, $\rho_1 \vdash e$ and $\rho_2 \vdash e$, along a path. The embedding check on environments is done pointwise, and if an embedding check of some $\rho_1(x) = \boldsymbol{\lambda} A_1.\ d_1$ must be done with $\rho_2(x) = \boldsymbol{\lambda} A_2.\ d_2$, then A_1 and A_2 are treated as α-equivalent [26], and *only the* *respective root nodes of the two derivations, d_1 and d_2, are compared for embedding.* This abbreviated comparison works safely because we restrict our semantics definitions to deterministic languages, that is, for a starting sequent, $\rho \vdash e$, there exists at most one maximal, well-formed derivation.

7 Conclusion

Not surprisingly, the partial derivation trees generated by our system are huge. The next stage of the research is developing compact representations of residual derivations. The value arrows of the derivations likely hold the key here; i.e. whenever $(\rho \vdash e) \to^* v$, the node $\rho \vdash e$ can be replaced by v.

The compact residual derivations should also facilitate linkage to other partial derivations. Then, the linkable compact residuals can be used in two ways. First, they serve as modules for incremental static analysis. This speeds up the analysis but more importantly realizes modular program analysis.

Second, the control flow graph of a residual derivation serves as skeleton for *automating* static analysis. As elucidated in [23], one can use standard methods of abstract interpretation in order to lift a concrete analysis (e.g. evaluation) into an abstract static analysis. Combining the two objectives, we arrive at *modular automated program analysis.*

8 Acknowledgements

Allen Stoughton provided substantial input on the small-step aspects of big-step semantics derivations; John Hatcliff and Morten Heine Sørensen gave useful advice. The anonymous referees contributed significant feedback in many aspects. Thanks are also due to Michael Leuschel for his organizational efforts.

References

[1] P. Borras, D. Clément, T. Despeyroux, J. Incerpi, G. Kahn, and B. Lang. Centaur: the system. In *Proc. SOGSOFT '88: Third Ann. Symp. on Software Development Environments.* ACM Press, 1988.

[2] E.M. Clarke, O. Grumberg, and D.E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, 1994.

[3] M. Codish, S. Debray, and R. Giacobazzi. Compositional analysis of modular logic programs. In *Proc. 20th ACM Symp. on Principles of Programming Languages*, pages 451–464. ACM Press, 1993.

[4] 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.

[5] 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.

[6] D. Dams, R. Gerth, and O. Grumberg. Abstract interpretation of reactive systems. *ACM TOPLAS*, 19:253–291, 1997.

[7] N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In J. vanLeeuwan, editor, *Handbook of Theoretical Computer Science*, pages 244–320. Elsevier, 1992.

[8] Th. Despeyroux. Executable specification of static semantics. In G. Kahn, D.B. MacQueen, and G. Plotkin, editors, *Semantics of Data Types*, pages 215–234. Lecture Notes in Computer Science 173, Springer-Verlag, 1984.

[9] M. Dwyer and D. Schmidt. Limiting state explosion with filter-based refinement. In Annalisa Bossi, editor, *International Workshop on Verification, Model Checking and Abstract Interpretation, Port Jefferson, Long Island, N.Y.*, `http://www.cis.ksu.edu/~schmidt/papers/filter.ps.Z`, 1997.

[10] J. Gallagher. Tutorial in specialisation of logic programs. In *ACM Symp. on Partial Evaluation and Semantics-Based Program Manipulation*, pages 88–98. ACM Press, 1993.

[11] J.P. Gallagher and L. Lafave. Regular approximation of computation paths in logic and functional la gnuages. In O. Danvy, R. Glück, and P. Thiemann, editors, *Partial Evaluation*, number 1110 in Lecture Notes in Computer Science, pages 115–136. Springer-Verlag, 1996.

[12] R. Glück and A. V. Klimov. Occam's razor in metacomputation: the notion of a perfect process tree. In G. Filè P.Cousot, M.Falaschi and A. Rauzy, editors, *Static Analysis. Proceedings*, volume 724 of *Lecture Notes in Computer Science*, pages 112–123. Springer-Verlag, 1993.

[13] R. Glück and M. H. Sørensen. Partial deduction and driving are equivalent. In M. Hermenegildo and J. Penjam, editors, *Programming Languages: Implementations, Logics and Programs*, volume 844 of *Lecture Notes in Computer Science*, pages 165–181. Springer-Verlag, 1994.

[14] R. Glück and M. H. Sørensen. A roadmap to metacomputation by supercompilation. In O. Danvy, R. Glück, and P. Thiemann, editors, *Partial Evaluation*, volume 1110 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.

[15] V. Gouranton and D. LeMétayer. Derivation of static analysers of functional programs from path properties of a natural semantics. Technical Report Research Report 2607, INRIA, 1995.

[16] N.D. Jones. The essence of program transformation by partial evaluation and driving. In N.D. Jones, N. Hagiya, and S. Masahiko, editors, *Logic, Language, and Computation: a Festscrift in Honor of Satoru Takasu*, pages 206–224. Lecture Notes in Computer Science 792, Springer-Verlag, 1994.

[17] Neil D. Jones. What not to do when writing an interpreter for specialisation. In Olivier Danvy, Robert Glück, and Peter Thiemann, editors, *Partial Evaluation*, volume 1110 of *Lecture Notes in Computer Science*, pages 216–237. Springer-Verlag, 1996.

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

[19] J.W. Klop. Term rewriting systems. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2, pages 2–117. Oxford University Press, 1992.

[20] J. Komorowski. An introduction to partial deduction. In A. Pettorossi, editor, *Meta-Programming in Logic*, volume 649 of *Lecture Notes in Computer Science*, pages 49–69, 1992.

[21] R. Kowalski. *Logic for Problem Solving*. North-Holland, 1979.

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

[23] 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.

[24] D.A. Schmidt. Trace-based abstract interpretation in the operational semantics hierarchy. Technical Report CIS-97-6, `http://www.cis.ksu.edu/~schmidt/papers/aiosh.ps.Z`, Kansas State University, Manhattan, Kansas, 1997.

[25] M. H. Sørensen. Turchin's supercompiler revisited. Master's thesis, Department of Computer Science, University of Copenhagen, 1994. DIKU-rapport 94/17.

[26] M. H. Sørensen. Properties of infinite reduction paths in untyped λ-calculus. In *Tbilisi Symposium on Language, Logic, and Computation*, CLSI Lecture Notes, 1996. to appear.

[27] M. H. Sørensen and R. Glück. An algorithm of generalization in positive supercompilation. In J.W. Lloyd, editor, *Logic Programming: Proceedings of the 1995 International Sympo sium*, pages 465–479. MIT Press, 1995.

[28] M. H. Sørensen, R. Glück, and N. D. Jones. Towards unifying deforestation, supercompilation, partial evaluation, and generalized partial computation. In D. Sannella, editor, *European Symposium on Programming*, volume 788 of *Lecture Notes in Computer Science*, pages 485–500. Springer-Verlag, 1994.

[29] A. Stoughton. Treating finite and infinite operational semantics derivation trees as objects. Technical Report 97-12, Computing Science, Kansas State University, 1997.

[30] V.F. Turchin. The concept of a supercompiler. *ACM Transactions on Programming Languages and Systems*, 8(3):292–325, 1986.

[31] V.F. Turchin. Program transformation by supercompilation. In H. Ganzinger and Neil D. Jones, editors, *Program as Data Objects, Copenhagen, Denmark*, volume 217 of *Lecture Notes in Computer Science*, pages 257–281, 1986.

[32] V.F. Turchin. The algorithm of generalization in the supercompiler. In D. Bjørner, A.P. Ershov, and N.D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 531–549. North-Holland, 1988.

[33] P.L. Wadler. Deforestation: Transforming programs to eliminate intermediate trees. *Theoretical Computer Science*, 73:231–248, 1990.