Transforming Big-Step to Small-Step Semantics Using Interpreter Specialisation

Gallagher, John P.; Hermenegildo, Manuel V.; Morales, José F.; López-García, Pedro

Published in:
Logic-Based Program Synthesis and Transformation

DOI:
10.1007/978-3-031-45784-5_3

Publication date:
2023

Document Version
Peer reviewed version

Citation for published version (APA):
Gallagher, J. P., Hermenegildo, M. V., Morales, J. F., & López-García, P. (2023). Transforming Big-Step to Small-Step Semantics Using Interpreter Specialisation. In R. Glück, & B. Kafle (Eds.), *Logic-Based Program Synthesis and Transformation: 33rd International Symposium, LOPSTR 2023, Cascais, Portugal, October 23-24, 2023, Proceedings* (Vol. 14330, pp. 28-38). Springer. https://doi.org/10.1007/978-3-031-45784-5_3

Transforming Big-step to Small-step Semantics using Interpreter Specialisation[⋆]

John P. Gallagher [⋆⋆1],[2][0000−0001−6984−7419], Manuel Hermenegildo[2],[3][0000−0002−7583−323X], José Morales[2],[3][0000−0001−9782−8135], and Pedro Lopez-Garcia[2],[4][0000−0002−1092−2071]

[1] Roskilde University, Denmark
[2] IMDEA Software Institute, Madrid, Spain
[3] Universidad Politécnica de Madrid (UPM) Madrid, Spain
[4] Spanish Council for Scientific Research (CSIC), Madrid, Spain

Abstract. Natural semantics (big-step) and structural operational semantics (small-step) each have advantages, so it can be useful to produce both semantic forms for a language. Previous work has shown that big-step semantics can be transformed to small-step semantics. This is also the goal of our work, but our main contribution is to show that this can be done by specialisation of an interpreter that imposes a small-step execution on big-step transition rules. This is arguably more direct, transparent and flexible than previous methods. The paper contains two examples and further examples are available in an online repository.

Keywords: Interpreter specialisation · Operational semantics

1 Introduction

The goal of this work is to transform big-step operational semantics to small-step operational semantics. This has previously been studied [2,18,30]. The main novelty is the method, which we consider to be more direct and transparent than previous approaches. We formulate the transformation as the specialisation of a "small-step" interpreter for big-step semantic rules. Once a suitable interpreter has been written, in which the definition of a "small step" has been encoded (see Section 3), the transformation consists of partially evaluating it with respect to given big-step semantics. The specialised interpreter contains the small-step transition rules, with minor syntactic modification. We describe experiments using an off-the-shelf partial evaluator for logic programs [23] (Section 4).

2 Background

Natural semantics (NS) was proposed by Kahn [22] as a proof-theoretic view of program semantics. Structural operational semantics (SOS) was developed

[⋆] Partially funded by MICINN projects PID2019-108528RB-C21 *ProCode*, TED2021-132464B-I00 *PRODIGY*, and FJC2021-047102-I, and by the Tezos foundation.

[⋆⋆] Email. `jpg@ruc.dk`

by Plotkin [27,28]. The motivation of SOS was to define machine-like execution of programs in a syntax-directed style, omitting all unnecessary details of the machine. Both styles have their advantages, which we do not discuss here.

We use the nicknames *big-step* and *small-step* for NS and SOS respectively, as they neatly express the difference between NS and SOS. Both approaches define the behaviour of a program as runs in a transition system. The system states (or *configurations*) have the form $\langle s, \sigma \rangle$ where s is a program expression (such as a statement) and σ is a program environment (such as a store); sometimes s is omitted when it is empty or associated with a final state.

In big-step semantics, transitions are of the form $\langle s, \sigma \rangle \Longrightarrow \sigma'$, or $\langle s, \sigma \rangle \Longrightarrow s'$ depending on the language being defined, which means that s is completely evaluated in σ, terminating in final state σ' or value s'. In small-step semantics, a transition has the form $\langle s, \sigma \rangle \Rightarrow \langle s', \sigma' \rangle$, which defines a single step from s in environment σ to the *next* configuration $\langle s', \sigma' \rangle$. We may also have transitions of the form $\langle s, \sigma \rangle \Rightarrow \sigma'$ or $\langle s, \sigma \rangle \Rightarrow s'$ for the case that s terminates in one step. There is a small-step (terminating) *run* iff $(\langle s, \sigma \rangle, v)$ is in $\Rightarrow^*$, the transitive closure of $\Rightarrow$. Note that we use $\Longrightarrow$ and $\Rightarrow$ for big and small-step transitions respectively. For transition relations $\Longrightarrow$ and $\Rightarrow$ for a given language, the equivalence requirement is that $\langle s, \sigma \rangle \Longrightarrow v$ iff for some n, $\langle s, \sigma \rangle \Rightarrow^n v$, for all $\langle s, \sigma \rangle$.

Interpreter specialisation. The idea of specialising a program with respect to partial input, known as program specialisation, partial evaluation or mixed computation, originated in the 1960s and 1970s [3,7,9,24]. Specialisation of a program interpreter with respect to an object program is related to compilation [9]. When the interpreter and the object program are written in the same language, the specialisation may be viewed as a source transformation of the object program (whereas it is in fact a transformation of the interpreter). This idea was exploited to transform programs [10,12,13,14,21,29], and can result in deep changes in program structure, possibly yielding superlinear speedups [21], in contrast to partial evaluation itself, which gives only linear speedups and does not fundamentally alter program structure. A transformation technique for logic programs with the similar aim of "compiling-in" non-standard semantics, *compiling control* [5], has also been shown to be realisable as interpreter specialisation [26].

The idea of transformation by interpreter specialisation is thus well known, yet its potential has not been fully realised, probably due to the fact that effective specialisation of complex interpreters is beyond the power of general purpose program specialisers and needs further research.

Summary of the approach. Let $\mathbf{b}$ be a set of big-step rules for a language and $\langle \mathbf{s}, \sigma \rangle \Longrightarrow \mathbf{v}$ be a big-step transition derived using $\mathbf{b}$; let $\mathbf{I}$ be an interpreter for big-step rules (written in a small-step style, see Section 3) and $\mathbf{pe}$ be a partial evaluator. Following the notational conventions of Jones *et al.* [20], $[\![p]\!]$ denotes the function corresponding to program p and we have the following equations.

$$\begin{aligned}\mathbf{v} &= [\![\mathbf{b}]\!]\ \langle \mathbf{s}, \sigma \rangle \\ &= [\![\mathbf{I}]\!]\ [\mathbf{b}, \langle \mathbf{s}, \sigma \rangle] \\ &= [\![\ [\![\mathbf{pe}]\!]\ [\mathbf{I}, \mathbf{b}]\ ]\!]\ \langle \mathbf{s}, \sigma \rangle\end{aligned}$$

In the first equation, $\mathbf{b}$ is itself an evaluator (indeed, a set of big-step rules is a logic program). $[\![\mathbf{I}]\!]\ [\mathbf{b}, \langle \mathbf{s}, \sigma \rangle]$ is the result of running the interpreter on $\mathbf{b}$ and $\langle \mathbf{s}, \sigma \rangle$ and the equation expresses the assumption that the interpreter yields the same result as running $\mathbf{b}$. The expression $[\![\mathbf{pe}]\!]\ [\mathbf{I}, \mathbf{b}]$ represents the result of specialising $\mathbf{I}$ with respect to the set of big-step rules $\mathbf{b}$. This yields an interpreter specific to $\mathbf{b}$ that follows the small-step style of $\mathbf{I}$, which can be applied directly to a configuration. It contains (after some minor syntactic modification) the small-step semantic rules corresponding to $\mathbf{b}$.

Horn clause representation of semantics and interpreters. Both big-step and small-step semantics are defined using rules with premises and conclusion, typically written as follows.

$$\frac{\textit{premises}}{\textit{conclusion}} \quad \text{if } \textit{condition}$$

With a suitable encoding of syntactic objects and environments as first-order terms, this is a first-order logic implication $\textit{premises} \wedge \textit{condition} \rightarrow \textit{conclusion}$ The conclusion is an atomic formula (a big- or small-step transition) so assuming that the premises and conditions are conjunctions, it is a Horn clause.

The close connection between transition rules and Horn clauses, and hence to the logic programming language Prolog, was noticed by Kahn and his co-workers and exploited in the Typol tool [6]. Similarly, small step transition rules, together with a rule specifying a run of small-step transitions, can also be written as Horn clauses and used to execute programs.

Interpreters for logic programs can themselves be written as logic programs, where the program being interpreted is represented in some way as a data structure in the interpreter (see [16,17] for a discussion of representations).

In the following, we use Prolog syntax and teletype font for Horn clauses.

3 A Small-step Interpreter for Big-step Semantics

Rule normalisation. A big-step rule has the following form (following [25]).

$$\frac{\langle s_1, \sigma_1 \rangle \Longrightarrow \sigma'_1, \ldots, \langle s_n, \sigma_n \rangle \Longrightarrow \sigma'_n}{\langle s_0, \sigma_0 \rangle \Longrightarrow \sigma'_0} \quad \text{if } c$$

This will be written as a Horn clause, where `E1` stands for σ_1, etc.

```
bigstep(S,E0,E01):-C,bigstep(S1,E1,E11),...,bigstep(Sn,En,En1)
```

The condition `C` could in general be interspersed among the other rule premises, rather than appearing on the left.

To simplify the interpreter, we assume that rules have *at most two premises*, including the conditions c. Rules can always be transformed to this normal form, possible adding extra syntax constructors. We could incorporate the normalisation in the interpreter, but we chose to automatically pre-process the rules to conform to the two-premise form. As an example, consider the following big-step inference rule taken from the call-by-value semantics of the λ-calculus, which is not in normal form.

$$\frac{\langle e_1, \rho\rangle \Longrightarrow \mathsf{clo}(x, e, \rho') \quad \langle e_2, \rho\rangle \Longrightarrow v_2 \quad \rho'[x/v_2] = \rho'' \quad \langle e, \rho''\rangle \Longrightarrow v}{\langle \mathsf{app}(e_1, e_2), \rho\rangle \Longrightarrow v}$$

After normalising, we get three rules, with new constructors app1 and app2.

$$\frac{\langle e_1, \rho\rangle \Longrightarrow \mathsf{clo}(x, e, \rho') \quad \langle \mathsf{app1}(x, e, \rho', e_2), \rho\rangle \Longrightarrow v}{\langle \mathsf{app}(e_1, e_2), \rho\rangle \Longrightarrow v}$$

$$\frac{\langle e_2, \rho\rangle \Longrightarrow v_2 \quad \langle \mathsf{app2}(x, e, \rho', v_2), \rho\rangle \Longrightarrow v}{\langle \mathsf{app1}(x, e, \rho', e_2), \rho\rangle \Longrightarrow v} \qquad \frac{\rho'[x/v_2] = \rho'' \quad \langle e, \rho''\rangle \Longrightarrow v}{\langle \mathsf{app2}(x, e, \rho', v_2), \rho\rangle \Longrightarrow v}$$

Structure of the interpreter. The main interpreter loop is as follows.

```
run([A]) :- smallStep(A,As), run(As).
run([]).
```

The `run` predicate takes as argument a stack of `bigstep` goals. The height of the stack is at most one. At each iteration of the main loop, if the stack is not empty, the top of the stack `A` is taken and a small step is applied, that is, `smallStep(A,As)` is called, resulting in `As`, which is either `[]` or `[A1]`. The loop is repeated until the stack is empty.

Definition of a small step. We now proceed to define `smallStep`, the crucial predicate in the interpreter. Given a call `smallStep(bigstep(S,E,V),As)`, the cases of the `smallStep` procedure in Figure 1 correspond to whether the rule with conclusion `bigstep(S,E,V)` has 0, 1 or 2 premises. For base cases 0 and 1, `smallStep` terminates immediately, returning either `[]` (for 0 premises) or `[B]` (where `B` is the single big-step premise). In the third case, the premises are `B1` and `B2`. We recursively call `smallStep` on `[B1]`, yielding `[D1]`, and then construct a new big-step call for the resulting goals `[D1,B2]`. This last step is performed by the predicate `foldStack`, which is now described.

Folding the stack. Consider the general form of a big-step rule with 2 premises. The rule has one of the following forms.

$$\frac{\langle s'_1, \sigma_1\rangle \Longrightarrow \sigma'_1 \quad \langle s'_2, \sigma_2\rangle \Longrightarrow \sigma'_2}{\langle f(s_1, \ldots, s_k), \sigma_0\rangle \Longrightarrow \sigma'_0} \qquad \frac{\langle s'_1, \sigma_1\rangle \Longrightarrow \sigma'_1 \quad c}{\langle f(s_1, \ldots, s_k), \sigma_0\rangle \Longrightarrow \sigma'_0}$$

Small step evaluation of such a rule evaluates the first premise, applying small steps until the first premise is completely evaluated, and then continues to the second premise. A typical example is the big-step rule for statement composition.

$$\frac{\langle s_1, \sigma\rangle \Longrightarrow \sigma' \quad \langle s_2, \sigma'\rangle \Longrightarrow \sigma''}{\langle s_1\,;s_2, \sigma\rangle \Longrightarrow \sigma''}$$

```
smallStep(A,[]) :-            % rule with zero big step premises
      givenRule(_,A,Bs),
      evalConditions(Bs,[]).
smallStep(A,[B]) :-           % rule with one big step premise
      givenRule(_,A,Bs),
      evalConditions(Bs,[B]).
smallStep(A,As) :-            % rule with 2 premises
      rule(K,A,[B1,B2]),
      bigStepPred(B1),
      smallStep(B1,D1),
      foldStack(D1,B2,K,As).

foldStack([],B2,_,As) :-    % B1 terminated
      evalConditions([B2],As).
foldStack([D1],B2,K,[H]) :- % Make new big step H with [D1,B2]
      newBigStep(D1,B2,K,H).
```

Fig. 1. Definition of a small step.

Evaluating $s_1\,;s_2$ in small steps, the evaluation of the first premise may take several small steps: $\langle s_1,\sigma\rangle \Rightarrow \langle s_{1,1},\sigma_{1,1}\rangle \Rightarrow \ldots \Rightarrow \sigma'$, which we can rewrite as the first small step followed by a big step for the rest.

$$\langle s_1,\sigma\rangle \Rightarrow \langle s_{1,1},\sigma_{1,1}\rangle, \langle s_{1,1},\sigma_{1,1}\rangle \Longrightarrow \sigma'$$

Thus after the first small step $\langle s_1,\sigma\rangle \Rightarrow \langle s_{1,1},\sigma_{1,1}\rangle$, the remaining premises of the rule are are $\langle s_{1,1},\sigma_{1,1}\rangle \Longrightarrow \sigma', \langle s_2,\sigma'\rangle \Longrightarrow \sigma''$. These form an instance of the rule premise, and so we can *fold* them to yield a single big-step, namely the rule conclusion $\langle s_{1,1}\,;s_2,\sigma_{1,1}\rangle \Longrightarrow \sigma''$. Note that if we apply the rule to this configuration, we obtain the same premises again. Hence, we derive the following relation between small-step pairs of big-step calls.

$$\frac{(\langle s_1,\sigma\rangle \Longrightarrow \sigma') \Rightarrow (\langle s_{1,1},\sigma_{1,1}\rangle \Longrightarrow \sigma')}{(\langle s_1\,;s_2,\sigma\rangle \Longrightarrow \sigma'') \Rightarrow (\langle s_{1,1}\,;s_2,\sigma_{1,1}\rangle \Longrightarrow \sigma'')}$$

The final state variables σ' and σ'' are arbitrary and can be eliminated, yielding the following recursive small-step rule for $s_1\,;s_2$.

$$\frac{\langle s_1,\sigma\rangle \Rightarrow \langle s_{1,1},\sigma_{1,1}\rangle}{\langle s_1\,;s_2,\sigma\rangle \Rightarrow \langle s_{1,1}\,;s_2,\sigma_{1,1}\rangle}$$

Auxiliary rules and constructors. In the general case for 2-premise rules, the first premise $\langle s'_1,\sigma_1\rangle \Longrightarrow \sigma'_1$ is split into a first small step and then a big step.

$$\langle s'_1,\sigma_1\rangle \Rightarrow \langle s'_{1,1},\sigma_{1,1}\rangle, \langle s'_{1,1},\sigma_{1,1}\rangle \Longrightarrow \sigma'_1$$

After executing the small step, the remaining rule premises are thus

$$\langle s'_{1,1},\sigma_{1,1}\rangle \Longrightarrow \sigma'_1, \langle s'_2,\sigma_2\rangle \Longrightarrow \sigma'_2$$

(and similarly where the second premise is c). This is not always an instance of the original rule, and so it is not always possible to fold using the same rule, as we did with the rule for $s_1\,;s_2$. In particular, the variables s'_1, σ_1 might be reused later in the premises. In such cases, we invent a new constructor, including s'_1 and/or σ_1 if needed as arguments, and construct a new auxiliary rule for the new constructor. We illustrate with an example. Consider the (normalised) rule for app shown above.

$$\frac{\langle e_1, \rho\rangle \Longrightarrow \mathsf{clo}(x, e, \rho') \quad \langle \mathsf{app1}(x, e, \rho', e_2), \rho\rangle \Longrightarrow v}{\langle \mathsf{app}(e_1, e_2), \rho\rangle \Longrightarrow v}$$

After executing a small step $\langle e_1, \rho\rangle \Rightarrow \langle e_{1,1}, \rho_{1,1}\rangle$ the remaining premises are $\langle e_{1,1}, \rho_{1,1}\rangle \Longrightarrow \mathsf{clo}(x, e, \rho') \quad \langle \mathsf{app1}(x, e, \rho', e_2), \rho\rangle \Longrightarrow v$. Note that ρ is used again later in the premises, so it is not possible to fold them into an app construct. Instead, we make a new constructor $\mathsf{app\_aux}$, and build rules as follows.

$$\frac{\langle e_1, \rho\rangle \Rightarrow \langle e_{1,1}, \rho_{1,1}\rangle}{\langle \mathsf{app}(e_1, e_2), \rho\rangle \Rightarrow \langle \mathsf{app\_aux}(e_{1,1}, e_2, \rho), \rho_{1,1}\rangle}$$

$$\frac{\langle e_{1,1}, \rho_{1,1}\rangle \Longrightarrow \mathsf{clo}(x, e, \rho'), \langle \mathsf{app1}(x, e, \rho', e_2), \rho\rangle \Longrightarrow v}{\langle \mathsf{app\_aux}(e_{1,1}, e_2, \rho), \rho_{1,1}\rangle \Longrightarrow v}$$

The second rule is an auxiliary big-step rule for $\mathsf{app\_aux}$, which is used later in the interpreter when needed. The interpreter does not store these auxiliary rules, but rather reconstructs them as needed, as this simplifies partial evaluation.

The complete interpreter, implemented in Ciao Prolog [15] along with the annotation file for the partial evaluator (Logen [23]), and a number of examples including the ones in the paper, are available online[5].

4 Examples

Simple imperative language. Figure 2(a) shows the big-step semantics for a simple imperative language containing assignments, statement composition, if-then-else and while statements. These are already in normal form. There is a function V that evaluates expressions and conditionals in a state. Specialising our small-step interpreter (using Logen) with respect to these rules gives the output shown in Figure 2(b). The small-step rules shown in Figure 2(c) are directly extracted (eliminating the redundant "final state" arguments as shown above). The rules are similar to textbook small-step semantics for imperative constructs.

Call-by-value semantics for λ-calculus. This example is taken from Vesely and Fisher's paper [30], and we applied normalisation to obtain 2-premise form (see the rule for $\mathsf{app}(e_1, e_2)$ shown at the start of Section 3). The big-step rules are shown in Figure 3(a). The specialised clauses for `smallStep` are in Figure 3(b) and the same rules typeset in standard notation in Figure 3(c). The results are somewhat different from those in [30], where the authors use small-step transitions yielding values rather than configurations.

[5] `https://github.com/jpgallagher/Semantics4PE/tree/main/Big2Small`

$$\frac{}{\langle x := e, \sigma\rangle \Longrightarrow \sigma[x/v]} \quad \text{if } V(e,\sigma) = v$$

$$\frac{\langle s_1, \sigma\rangle \Longrightarrow \sigma' \qquad \langle s_2, \sigma'\rangle \Longrightarrow \sigma''}{\langle s_1\,;s_2, \sigma\rangle \Longrightarrow \sigma''}$$

$$\frac{}{\langle \mathsf{skip}, \sigma\rangle \Longrightarrow \sigma}$$

$$\frac{\langle s_1, \sigma\rangle \Longrightarrow \sigma'}{\langle \mathsf{if}\ (b)\ s_1\ \mathsf{else}\ s_2, \sigma\rangle \Longrightarrow \sigma'} \quad \text{if } V(b,\sigma) = \mathsf{true}$$

$$\frac{\langle s_2, \sigma\rangle \Longrightarrow \sigma'}{\langle \mathsf{if}\ (b)\ s_1\ \mathsf{else}\ s_2, \sigma\rangle \Longrightarrow \sigma'} \quad \text{if } V(b,\sigma) = \mathsf{false}$$

$$\frac{\langle \mathsf{if}\ (b)\ s; \mathsf{while}\ (b)\ s\ \mathsf{else}\ \mathsf{skip}, \sigma\rangle \Longrightarrow \sigma'}{\langle \mathsf{while}\ (b)\ s, \sigma\rangle \Longrightarrow \sigma'}$$

(a) Big-step rules for a simple imperative language

```
smallStep__1(asg(var(D),C),A,B,[]) :-
   eval__2(C,A,E,F),
   eval__3(D,F,E,B).
smallStep__1(ifthenelse(C,D,_),A,B,[bigstep(D,E,B)]) :-
   eval__2(C,A,E,1).
smallStep__1(ifthenelse(C,_,D),A,B,[bigstep(D,E,B)]) :-
   eval__2(C,A,E,0).
smallStep__1(while(C,D),A,B,[bigstep(ifthenelse(C,seq(D,while(C,D)),skip),
     A,B)]).
smallStep__1(seq(C,D),A,B,[bigstep(D,E,B)]) :-
   smallStep__1(C,A,E,[]).
smallStep__1(seq(C,D),A,B,[bigstep(seq(F,D),E,B)]) :-
   smallStep__1(C,A,G,[bigstep(F,E,G)]).
```

(b) Small-step clauses from the specialised interpreter.

$$\frac{}{\langle \mathsf{if}\ (b)\ s_1\ \mathsf{else}\ s_2, \sigma\rangle \Rightarrow \langle s_1, \sigma'\rangle} \quad \text{if } V(b,\sigma) = \mathsf{true}$$

$$\frac{}{\langle \mathsf{if}\ (b)\ s_1\ \mathsf{else}\ s_2, \sigma\rangle \Rightarrow \langle s_2, \sigma'\rangle} \quad \text{if } V(b,\sigma) = \mathsf{false}$$

$$\frac{\langle s_1, \sigma\rangle \Rightarrow \sigma'}{\langle s_1\,;s_2, \sigma\rangle \Rightarrow \langle s_2, \sigma'\rangle}$$

$$\frac{}{\langle \mathsf{while}\ (b)\ s, \sigma\rangle \Rightarrow \langle \mathsf{if}\ (b)\ s; \mathsf{while}\ (b)\ s\ \mathsf{else}\ \mathsf{skip}, \sigma\rangle}$$

$$\frac{}{\langle x := e, \sigma\rangle \Rightarrow \sigma[x/v]} \quad \text{if } V(e,\sigma) = v$$

$$\frac{}{\langle \mathsf{skip}, \sigma\rangle \Rightarrow \sigma}$$

$$\frac{\langle s_1, \sigma\rangle \Rightarrow \langle s_1', \sigma'\rangle}{\langle s_1\,;s_2, \sigma\rangle \Rightarrow \langle s_1'\,;s_2, \sigma'\rangle}$$

(c) Small-step rules from (b) rewritten in standard form.

Fig. 2. Transformation of the semantics of a simple imperative language.

$$\frac{}{\langle \mathsf{var}(x), \rho\rangle \Longrightarrow v} \quad \text{if } \rho x = v \qquad\qquad \frac{\langle e_1, \rho\rangle \Longrightarrow \mathsf{clo}(x, e, \rho) \quad \langle \mathsf{a41}(e_2, x, e, \rho'), \rho\rangle \Longrightarrow v}{\langle \mathsf{app}(e_1, e_2), \rho\rangle \Longrightarrow v}$$

$$\frac{}{\langle \mathsf{val}(v), \rho\rangle \Longrightarrow v} \qquad\qquad \frac{\langle e_2, \rho\rangle \Longrightarrow v_2 \quad \langle \mathsf{a42}(x, e, \rho', v_2), \rho\rangle \Longrightarrow v}{\langle \mathsf{a41}(e_2, x, e, \rho'), \rho\rangle \Longrightarrow v}$$

$$\frac{}{\langle \mathsf{lam}(x, e), \rho\rangle \Longrightarrow \mathsf{clo}(x, e, \rho)} \qquad\qquad \frac{\rho'[x/v_2] = \rho'' \quad \langle e, \rho''\rangle \Longrightarrow v}{\langle \mathsf{a42}(x, e, \rho', v_2), \rho\rangle \Longrightarrow v}$$

(a) Big-step rules for the call-by-value λ-calculus (from [30]), after normalising

```
smallStep1(val(A),_,A,[]).
smallStep__1(var(C),A,B,[]) :-
   eval__2(A,C,B).
smallStep__1(lam(B,C),A,clo(B,C,A),[]).
smallStep__1(app4_2(C,D,E,F),A,B,[bigstep(D,A,B)]) :-
   eval__3(C,F,E,A).
smallStep__1(app(C,D),A,B,[bigstep(app4_1(D,E,F,G),A,B)]) :-
   smallStep__1(C,A,clo(E,F,G),[]).
smallStep__1(app(C,D),A,B,[bigstep(app_aux_3(F,D,A),E,B)]) :-
   smallStep__1(C,A,clo(G,H,I),[bigstep(F,E,clo(G,H,I))]).
smallStep__1(app4_1(C,D,E,F),A,B,[bigstep(app4_2(D,E,F,G),_,B)]) :-
   smallStep__1(C,A,G,[]).
smallStep__1(app4_1(C,D,E,F),A,B,[bigstep(app4_1(H,D,E,F),G,B)]) :-
   smallStep__1(C,A,I,[bigstep(H,G,I)]).
smallStep__1(app_aux_3(C,D,E),A,B,[bigstep(app4_1(D,F,G,H),E,B)]) :-
   smallStep__1(C,A,clo(F,G,H),[]).
smallStep__1(app_aux_3(C,D,E),A,B,[bigstep(app_aux_3(G,D,E),F,B)]) :-
   smallStep__1(C,A,clo(H,I,J),[bigstep(G,F,clo(H,I,J))]).
```

(b) Small-step rules taken from the specialised interpreter

$$\frac{}{\langle \mathsf{val}(v), \rho\rangle \Rightarrow v} \qquad\qquad \frac{\langle e_1, \rho\rangle \Rightarrow \langle e_3, \rho'\rangle}{\langle \mathsf{app}(e_1, e_2), \rho\rangle \Rightarrow \langle \mathsf{app}'(e_3, e_2, \rho), \rho'\rangle}$$

$$\frac{\rho\ x = v}{\langle \mathsf{var}(x), \rho\rangle \Rightarrow v} \qquad\qquad \frac{\langle e_2, \rho\rangle \Rightarrow v_2}{\langle \mathsf{a41}(e_2, x, e, \rho'), \rho\rangle \Rightarrow \langle \mathsf{a42}(x, e, \rho', v_2), \rho''\rangle}$$

$$\frac{}{\langle \mathsf{lam}(x, e), \rho\rangle \Rightarrow \mathsf{clo}(x, e, \rho)} \qquad\qquad \frac{\langle e_2, \rho\rangle \Rightarrow \langle e_3, \rho''\rangle}{\langle \mathsf{a41}(e_2, x, e, \rho'), \rho\rangle \Rightarrow \langle \mathsf{a41}(e_3, x, e, \rho'), \rho''\rangle}$$

$$\frac{\rho'[x/v_2] = \rho''}{\langle \mathsf{a42}(x, e, v_2, \rho'), \rho''\rangle \Rightarrow \langle e, \rho''\rangle} \qquad\qquad \frac{\langle e_3, \rho'\rangle \Rightarrow \mathsf{clo}(x, e, \rho'')}{\langle \mathsf{app}'(e_3, e_2, \rho), \rho'\rangle \Rightarrow \langle \mathsf{a41}(e_2, x, e, \rho''), \rho\rangle}$$

$$\frac{\langle e_1, \rho\rangle \Rightarrow \mathsf{clo}(x, e, \rho')}{\langle \mathsf{app}(e_1, e_2), \rho\rangle \Rightarrow \langle \mathsf{a41}(e_2, x, e, \rho'), \rho\rangle} \qquad\qquad \frac{\langle e_3, \rho'\rangle \Rightarrow \langle e_4, \rho''\rangle}{\langle \mathsf{app}'(e_3, e_2, \rho), \rho'\rangle \Rightarrow \langle \mathsf{app}'(e_4, e_2, \rho), \rho''\rangle}$$

(c) Small-step rules from (b) rendered in standard notation.

Fig. 3. Transformation of big-step to small-step semantics for the call-by-value λ-calculus.

5 Related Work

Vesely and Fisher [30] start from an evaluator for big-step semantics and then transform it in (ten) stages to a small-step evaluator. While our interpreter embodies some of the same transformations (for instance, continuation-passing transformation corresponds roughly to specialisation of our `run` predicate), some of their transformations are more low-level, and are subsumed by standard features of partial evaluation (for instance, argument lifting). We also exploit the uniform representation as Horn clauses and the term representation of object programs in the interpreter to avoid explicit defunctionalisation or continuations-to-terms. When comparing our results to theirs, we note (as also observed in [2]) that they choose a version of small-step transitions that yields a term rather than a configuration. Therefore our transitions are not directly comparable in terms of the number of rules and constructors in the resulting small-step semantics.

Ambal *et al.* [2] describe a transformation that also starts from a continuation-passing transformation of a big-step evaluator. The transformation steps are certified in Coq. To eliminate the continuation stack they introduce a new syntax constructor for each continuation, much as we do for the case when direct folding cannot be applied. Overall, the transformation seems similar to ours though the method differs and we are comparing the results more closely.

Huizing *et al.* [19] describe a direct transformation of big-step rules - so their approach is not explicitly based on an interpreter. Ager [1] defines a transformation from L-attributed big-step rules to abstract machines, a related problem. His is also a direct transformation of rules rather than based on an explicit evaluator. Our procedure can also handle rules that are not L-attributed (such as Kahn's big-step semantics for Mini-ML [22]).

Discussion and future research. We claim that interpreter specialisation gives a more direct and transparent approach. Its correctness depends on validating the interpreter and the correctness of the partial evaluator, an established and well-tested tool. We do not yet have a formal proof of correctness of the interpreter. However, our interpretive approach using Horn clauses allows execution of the original big-step rules, the interpreter and its specialisation, so we have been able to perform extensive validation showing that they all produce the same results for a given configuration. This is a practical advantage of exploiting the close relationship between semantic rules and Horn clauses, though clearly not a replacement for a formal proof of correctness. Furthermore, the definition of what is a small step can be modified; for example, evaluation of conditionals in `if` statements could be done in small steps instead of as an atomic operation by a small modification of the interpreter. We are applying the method to further cases of big-step semantics; among other challenging examples we are targeting the semantics for Clight [4].

At present we generate structural operational semantics, with transitions from configurations to configurations. Other styles that could be targeted by an interpreter are transitions that yield values rather than configurations (e.g. as used by Vesely and Fisher [30]) and reduction semantics [8]. The related problem of transforming a big-step semantics to an abstract machine seems also a feasible goal for interpreter specialisation; in this case, the interpreter would incorporate implementation details that would be inherited by the abstract machine.

We also note that our previous work on translating imperative programs to Horn clauses [11] is related to the present work. Instead of performing the specialisation $[\![\mathbf{pe}]\!]\ [\mathbf{I}, \mathbf{b}]$, we would also provide the first component of the initial configuration $\langle \mathbf{s}, \sigma \rangle$; that is, we compute $[\![\mathbf{pe}]\!]\ [\mathbf{I}, \mathbf{b}, \mathbf{s}]$, obtaining Horn clauses which would compute directly on the environment σ.

Acknowledgements. Discussions with Robert Glück, Bishoksan Kafle, Morten Rhiger and Mads Rosendahl are gratefully acknowledged. The paper was improved by the suggestions of the anonymous reviewers.

References

1. Ager, M.S.: From natural semantics to abstract machines. In: Proc. LOPSTR 2004. LNCS, vol. 3573, pp. 245–261. Springer (2004). `https://doi.org/10.1007/11506676_16`
2. Ambal, G., Lenglet, S., Schmitt, A., Noûs, C.: Certified derivation of small-step from big-step skeletal semantics. In: Proc. PPDP 2022. pp. 11:1–11:48. ACM (2022). `https://doi.org/10.1145/3551357.3551384`
3. Beckman, L., Haraldson, A., Oskarsson, Ö., Sandewall, E.: A partial evaluator, and its use as a programming tool. Artif. Intell. **7**(4), 319–357 (1976). `https://doi.org/10.1016/0004-3702(76)90011-4`
4. Blazy, S., Leroy, X.: Mechanized semantics for the Clight subset of the C language. J. Autom. Reasoning **43**(3), 263–288 (2009). `https://doi.org/10.1007/s10817-009-9148-3`
5. Bruynooghe, M., De Schreye, D., Krekels, B.: Compiling control. Journal of Logic Programming **6(2–3)**, 135–162 (1989)
6. Despeyroux, T.: Executable specification of static semantics. In: Semantics of Data Types. LNCS, vol. 173, p. 215–233. Springer-Verlag (1984)
7. Ershov, A.P.: On the partial computation principle. Inf. Process. Lett. **6**(2), 38–41 (1977). `https://doi.org/10.1016/0020-0190(77)90078-3`
8. Felleisen, M., Friedman, D.P.: A reduction semantics for imperative higher-order languages. In: Proc. PARLE 87. LNCS, vol. 259, pp. 206–223. Springer (1987). `https://doi.org/10.1007/3-540-17945-3_12`
9. Futamura, Y.: Partial evaluation of computation process - an approach to a compiler-compiler. Systems, Computers, Controls **2(5)**, 45–50 (1971)
10. Gallagher, J.P.: Transforming logic programs by specialising interpreters. In: Proc. ECAI-86. pp. 109–122 (1986)
11. Gallagher, J.P., Hermenegildo, M.V., Kafle, B., Klemen, M., López-García, P., Morales, J.F.: From big-step to small-step semantics and back with interpreter specialisation. In: Proc. VPT/HCVS@ETAPS 2020. EPTCS, vol. 320, pp. 50–64 (2020). `https://doi.org/10.4204/EPTCS.320.4`
12. Giacobazzi, R., Jones, N.D., Mastroeni, I.: Obfuscation by partial evaluation of distorted interpreters. In: PEPM. pp. 63–72. ACM (2012). `https://doi.org/10.1145/2103746.2103761`

13. Glück, R.: On the generation of specializers. J. Funct. Program. **4**(4), 499–514 (1994). https://doi.org/10.1017/S0956796800001167
14. Glück, R., Jørgensen, J.: Generating transformers for deforestation and supercompilation. In: Charlier, B.L. (ed.) Proc. SAS'94. LNCS, vol. 864, pp. 432–448. Springer (1994). https://doi.org/10.1007/3-540-58485-4_57
15. Hermenegildo, M.V., Bueno, F., Carro, M., Lopez-Garcia, P., Mera, E., Morales, J., Puebla, G.: An Overview of Ciao and its Design Philosophy. Theory and Practice of Logic Programming **12**(1–2), 219–252 (2012). https://doi.org/10.1017/S1471068411000457
16. Hill, P.M., Gallagher, J.P.: Handbook of Logic in Artificial Intelligence and Logic Programming, vol. 5, chap. Meta-Programming in Logic Programming, pp. 421–498. Oxford University Press (1998)
17. Hill, P.M., Lloyd, J.W.: Analysis of meta-programs. In: Meta-Programming in Logic Programming. pp. 23–51. MIT Press (1988)
18. Huizing, C., Koymans, R., Kuiper, R.: A small step for mankind. In: Dams, D., Hannemann, U., Steffen, M. (eds.) Concurrency, Compositionality, and Correctness, Essays in Honor of Willem-Paul de Roever. LNCS, vol. 5930, pp. 66–73. Springer (2010). https://doi.org/10.1007/978-3-642-11512-7_5
19. Huizing, C., Koymans, R., Kuiper, R.: A small step for mankind. In: Concurrency, Compositionality, and Correctness. LNCS, vol. 5930, pp. 66–73. Springer (2010). https://doi.org/10.1007/978-3-642-11512-7_5
20. Jones, N.D., Gomard, C., Sestoft, P.: Partial Evaluation and Automatic Software Generation. Prentice Hall (1993). https://doi.org/10.1016/j.scico.2004.03.010
21. Jones, N.D.: Transformation by interpreter specialisation. Sci. Comput. Program. **52**, 307–339 (2004). https://doi.org/10.1016/j.scico.2004.03.010
22. Kahn, G.: Natural semantics. In: Proc. STACS 87. LNCS, vol. 247, pp. 22–39. Springer (1987). https://doi.org/10.1007/BFb0039592
23. Leuschel, M., Jørgensen, J.: Efficient specialisation in Prolog using the handwritten compiler generator LOGEN. Elec. Notes Theor. Comp. Sci. **30(2)** (1999). https://doi.org/10.1017/S1471068403001662
24. Lombardi, L.A.: Incremental computation: The preliminary design of a programming system which allows for incremental data assimilation in open-ended man-computer information systems. Adv. Comput. **8**, 247–333 (1967). https://doi.org/10.1016/S0065-2458(08)60698-1
25. Nielson, H.R., Nielson, F.: Semantics with applications - a formal introduction. Wiley professional computing, Wiley (1992)
26. Nys, V., Schreye, D.D.: Compiling control as offline partial deduction. In: Mesnard, F., Stuckey, P.J. (eds.) Proc. LOPSTR. LNCS, vol. 11408, pp. 115–131. Springer (2018). https://doi.org/10.1007/978-3-030-13838-7_7
27. Plotkin, G.D.: The origins of structural operational semantics. J. Log. Alg. Prog. **60-61**, 3–15 (2004). https://doi.org/10.1016/j.jlap.2004.03.009
28. Plotkin, G.D.: A structural approach to operational semantics. J. Log. Alg. Prog. **60-61**, 17–139 (2004)
29. Turchin, V.F.: Program transformation by supercompilation. In: Ganzinger, H., Jones, N.D. (eds.) Programs as Data Objects. LNCS, vol. 217, pp. 257–281. Springer (1985). https://doi.org/10.1007/3-540-16446-4_15
30. Vesely, F., Fisher, K.: One step at a time - A functional derivation of small-step evaluators from big-step counterparts. In: Caires, L. (ed.) Proc. ESOP 2019. LNCS, vol. 11423, pp. 205–231. Springer (2019). https://doi.org/10.1007/978-3-030-17184-1_8