# Towards Small-step Compilation Schemas for SOS

Ferdinand Vesely

Department of Computer Science, Swansea University, Swansea SA2 8PP, UK
`csfvesely@swansea.ac.uk`

**Abstract**

We present work in progress on a method of compiling programs based on SOS specifications. The idea is to compile programs using SOS rules by translation into labelled blocks with explicit exit points, which implement a valid computation in the LTS of the program. Under this approach, a correct compiler can be constructed in a systematic way, based on an SOS specification.

## 1 Introduction and Background

Small-step SOS is a popular framework for specifying semantics of programming and specification languages. A collection of SOS rules together define the transitions of a labelled transition system (LTS). For programming languages, the states of an LTS usually contain a program term along with auxiliary entities (stores, environments) and labels may contain emitted signals or output streams. Table 1 contains a small example specification. Under such a specification, each program is represented by a concrete LTS. We present a compilation method which can be understood as the translation of the LTS to a corresponding control-flow graph (CFG). The nodes of the CFG are sequences of instructions with behaviour that should be equivalent to the states in the LTS.

**Atomic Blocks** Our method produces a collection of labelled *atomic blocks* (AB) containing instructions for a virtual machine. Each AB corresponds to a state in the LTS of the program, and is essentially a *basic block*: a sequence of instructions with one entry (at the beginning) and one exit point (at the end) [1]. However, we relax the second condition and allow multiple exit points at the end of the block, while requiring that an AB executes atomically as a single unit.

**Target Machine Language** We are targeting a simple register machine, with an unlimited supply of temporaries (registers). In this regard it is similar to LLVM [3], which we intend to

$$\frac{\rho \vdash s_1 \xrightarrow{l} s_1'}{\rho \vdash \mathsf{let}(i, s_1, s_2) \xrightarrow{l} \mathsf{let}(i, s_1', s_2)} \quad (1) \qquad \frac{\mathsf{Value}\, v_1 \quad \rho[i \mapsto v_1] \vdash s_2 \xrightarrow{l} s_2'}{\rho \vdash \mathsf{let}(i, v_1, s_2) \xrightarrow{l} \mathsf{let}(i, v_1, s_2')} \quad (2)$$

$$\frac{\mathsf{Value}\, v_2}{\rho \vdash \mathsf{let}(i, v_1, v_2) \xrightarrow{\tau} v_2} \quad (3) \qquad \frac{\rho(i) = v}{\rho \vdash \mathsf{bound}(i) \xrightarrow{\tau} v} \quad (4)$$

$$\frac{\rho \vdash s \xrightarrow{l} s'}{\rho \vdash \mathsf{print}(s) \xrightarrow{l} \mathsf{print}(s')} \quad (5) \qquad \frac{\mathsf{Value}\, v}{\rho \vdash \mathsf{print}(v) \xrightarrow{\mathbf{out}\ v} \mathsf{skip}} \quad (6)$$

Table 1: Example language specification. '$\mathsf{Value}\, s$' asserts that '$\langle \rho, s\rangle$' is a value (terminal) state for any $\rho$. As usual, $\rho \vdash s \xrightarrow{l} s'$ is a shorthand for $\langle \rho, s\rangle \xrightarrow{l} \langle \rho, s'\rangle$.

use as the ultimate target. There is no program counter, instead the program is stored as a set of labelled code blocks $\beta$. Each block is (just) a sequence of instructions '$\iota_1 \cdot \iota_2 \cdot \ldots$'. The basic control-flow instructions are `halt` for stopping the machine immediately, and `jump` for unconditionally jumping to a labelled AB.

$$\frac{}{\beta \vdash \texttt{halt} \cdot t \xrightarrow{\tau} \texttt{halt}} \qquad \frac{\langle l, t_2 \rangle \in \beta}{\beta \vdash \texttt{jump}\ \ l \cdot t_1 \xrightarrow{\tau} t_2}$$

Further instructions will be mentioned in our translation example in the next section.

## 2 A Small-step Compilation Schema

Let's take a simple construct like **print**. If there is a sequence of $n$ transitions starting from term $t$, then the computation starting from **print**$(t)$ will look as follows:

$$\begin{array}{ccccccccccc} t & \xrightarrow{L_1} & t_1 & \xrightarrow{L_2} & \cdots & \xrightarrow{L_{n-1}} & t_{n-1} & \xrightarrow{L_n} & v & & \\ \textbf{print}(t) & \xrightarrow{L_1} & \textbf{print}(t_1) & \xrightarrow{L_2} & \cdots & \xrightarrow{L_{n-1}} & \textbf{print}(t_{n-1}) & \xrightarrow{L_n} & \textbf{print}(v) & \xrightarrow{\{\textbf{out}=v,\ldots\}} & \textbf{skip} \end{array}$$

The ABs for **print**$(t)$ should each corresponds to a term (state) in the lower part of the above sequence. We construct a *translator* which will generate code blocks that implement the steps of the construct. A translator for construct $f$, $\mathsf{tr}_f$, is a structure of operations $\mathsf{next}$, $\mathsf{code}$, and $\mathsf{label}$. A *translator state* is constructed by applying $\mathsf{tr}_f$ to translator states for arguments of $f$. We also write $[\![f(t_1, \ldots, t_n)]\!]$ for $\mathsf{tr}_f([\![t_1]\!], \ldots, [\![t_n]\!])$, where $n$ is the arity of $f$. For a translator $tr$, $\mathsf{next}\,tr$ is the next translator state, $\mathsf{code}\,tr$ is a code block corresponding to the current state, and $\mathsf{label}\,tr$ assigns a name to the state. The name can be used as a label for the atomic block or as the name of a temporary holding the computed value. The value of $\mathsf{next}\,tr$ can be $\mathsf{none}$ if the current state is final. In that case, the instructions in $\mathsf{code}\,tr$ must store a value in the named temporary $\mathsf{label}\,tr$. The translator for **print**, $\mathsf{tr}_{\textbf{print}}$, can be defined as follows:

$$\mathsf{code}[\![\textbf{print}(t)]\!] = \begin{cases} \mathsf{code}[\![t]\!] & \text{if } \mathsf{next}[\![t]\!] \neq \mathsf{none} \\ \mathsf{code}[\![t]\!] \cdot \texttt{out}\ \ temp & \text{otherwise, } temp = \mathsf{label}[\![t]\!] \end{cases} \tag{7}$$

$$\mathsf{next}[\![\textbf{print}(t)]\!] = \begin{cases} \mathsf{tr}_{\textbf{print}}(\mathsf{next}[\![t]\!]) & \text{if } \mathsf{next}[\![t]\!] \neq \mathsf{none} \\ [\![\textbf{skip}]\!] & \text{otherwise} \end{cases} \tag{8}$$

A translator for a value $v$ just has to put (a representation of) the value to a temporary, for which we introduce an instruction `ldval`.

$$\mathsf{code}[\![v]\!] = \texttt{ldval}\ \ temp\ \ v \qquad \mathsf{next}[\![v]\!] = \mathsf{none} \qquad \mathsf{label}[\![v]\!] = temp \tag{9}$$
$$\text{(where } temp \text{ is a fresh temporary name)}$$

The role of a top-level translator $\mathsf{tr}_{\mathsf{top}}$ is to take the code block for each step and turn it into an atomic block by appending an explicit exit point (`jump` or `halt`):

$$\mathsf{code}[\![t]\!]_{\mathsf{top}} = \begin{cases} \mathsf{code}[\![t]\!] \cdot \texttt{jump}\ \ l & \text{if } \mathsf{next}[\![t]\!] \neq \mathsf{none} \text{ and } l = \mathsf{label}(\mathsf{next}[\![t]\!]) \\ \mathsf{code}[\![t]\!] \cdot \texttt{halt} & \text{otherwise} \end{cases} \tag{10}$$

$$\mathsf{next}[\![t]\!]_{\mathsf{top}} = \mathsf{next}[\![t]\!] \qquad \mathsf{label}[\![t]\!]_{\mathsf{top}} = \mathsf{label}[\![t]\!] \tag{11}$$

The main compilation function just collects all the atomic blocks for a term, and returns them together with the initial block label:

$$\mathcal{C}(s) = \{\langle \mathsf{label}[\![t]\!]_{\mathsf{top}}, \mathsf{fold}_{\mathsf{tr}}[\![t]\!]_{\mathsf{top}}\rangle\} \tag{12}$$

where

$$\mathsf{fold}_{\mathsf{tr}}\ tr = \begin{cases} \{\langle \mathsf{label}\ tr, \mathsf{code}\ tr\rangle\} \cup \mathsf{fold}_{\mathsf{tr}}(\mathsf{next}\ tr) & \text{if } tr \neq \mathsf{none} \\ \emptyset & \text{otherwise} \end{cases} \tag{13}$$

As a further illustration, we look at a definition of code for **let**. The construct uses an updated context in the premise of the rule in Eq. (2). The resulting code block also has to provide a corresponding context for the sub-block. This context has to be explicitly constructed at the beginning of the premise transition and cleaned up at the end. In this definition we assume a value translator for sets of mappings '$\{i \mapsto v\}$' and machine operations for manipulating environments.

$$\mathsf{code}[\![\mathbf{let}(i, t_1, t_2)]\!] = \begin{cases} \mathsf{code}[\![t_1]\!] & \text{if } \mathsf{next}[\![t_1]\!] \neq \mathsf{none} \\ \mathsf{code}\ tr_{iv} \cdot \texttt{push\_env}\ \ tmp \cdot \mathsf{code}[\![t_2]\!] \cdot \texttt{pop\_env} & \begin{array}{l}\text{if } \mathsf{next}[\![t_1]\!] = \mathsf{none}, \\ \mathsf{next}[\![t_2]\!] \neq \mathsf{none}, \\ tr_{iv} = [\![\{i \mapsto t_2\}]\!], \\ tmp = \mathsf{label}(tr_{iv})\end{array} \\ \mathsf{code}[\![t_2]\!] & \text{otherwise} \end{cases} \tag{14}$$

# 3 Conclusion

We have illustrated a schema for small-step compilation on a few simple programming constructs. For lack of space we didn't illustrate translations for, e.g., conditional, iterative, or non-deterministic constructs. The approach could be used with a suitable notion of bisimulation: to prove its correctness, to develop a compiler calculation method (following [2]), and to explore (semi-) automatic compiler generation based on SOS rules. To this end, we intend to work with Modular SOS [4], a modular variant of SOS, which places all auxiliary entities into labels of transitions, and the corresponding notions of bisimulation [5]. To deal with inherent inefficiencies (e.g., construction and destruction of contexts in atomic blocks), common optimisation methods, such as peephole optimisation, could be applied to the resulting translations.

# References


[1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition).* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[2] P. Bahr and G. Hutton. Calculating Correct Compilers. To appear in J. Fun. Program., 2015.

[3] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. CGO '04, Washington, DC, USA, 2004. IEEE Computer Society.

[4] P. D. Mosses. Modular structural operational semantics. *J. Log. Algebr. Program.*, 60-61:195–228, 2004.

[5] P. D. Mosses and F. Vesely. Weak bisimulation as a congruence in MSOS. In N. Martí-Oliet, P. C. Ölveczky, and C. Talcott, editors, *Logic, Rewriting, and Concurrency*, volume 9200 of *LNCS*, pages 519–538. Springer, 2015.