From Small-Step Semantics to Big-Step Semantics, Automatically*

Ştefan Ciobâcă

Faculty of Computer Science, University "Alexandru Ioan Cuza"
Iaşi, Romania
stefan.ciobaca@info.uaic.ro

Abstract. Small-step semantics and big-step semantics are two styles for operationally defining the meaning of programming languages. Small-step semantics are given as a relation between program configurations that denotes one computational step; big-step semantics are given as a relation directly associating to each program configuration the corresponding final configuration. Small-step semantics are useful for making precise reasonings about programs, but reasoning in big-step semantics is easier and more intuitive. When both small-step and big-step semantics are needed for the same language, a proof of the fact that the two semantics are equivalent should also be provided in order to trust that they both define the same language. We show that the big-step semantics can be automatically obtained from the small-step semantics when the small-step semantics are given by inference rules satisfying certain assumptions that we identify. The transformation that we propose is very simple and we show that when the identified assumptions are met, it is sound and complete in the sense that the two semantics are equivalent. For a strict subset of the identified assumptions, we show that the resulting big-step semantics is sound but not necessarily complete. We discuss our transformation on a number of examples.

1 Introduction

In order to reason about programs, a formal semantics of the programming language is needed. There exist a number of styles in which the semantics can be given: denotational [1] (which associates to a program a mathematical object taken to be its meaning), axiomatic [2,3] (where the meaning of a program is exactly what can proven about it from the set of axioms) and operational [4,5,6] (which describes how the program executes on an abstract machine). Each semantic style has its own advantages and disadvantages. Sometimes, two or more semantics in different styles are needed for a programming language. In such a case, the equivalence of the different semantics must be proven in order to be sure that they really describe the same language.

In this article, we focus on two (sub)styles of operational semantics: small-step structural operational semantics [5] and big-step structural operational semantics [6] (also called natural semantics). Small-step semantics are given as

* This work was supported by Contract 161/15.06.2010, SMISCSNR 602-12516 (DAK).

a relation ($\rightarrow$) between program configurations modelling code and state. The small-step relation is usually defined inductively (based on the structure of the abstract syntax of the programming language – therefore the name of *structural* operational semantics). Configurations in which programs are expected to end are called *final configurations* (e.g., configurations in which there is no more code to execute). The transitive-reflexive closure $\rightarrow^*$ of the small-step rewrite relation $\rightarrow$ is taken to model the execution of a program. Configurations that cannot take a small step but which are not final configurations are *stuck* (e.g., when the program is about to cause a runtime error such as dividing by zero).

In contrast, big-step (structural operational) semantics describe the meaning of a programming language by an inductively defined predicate $\Downarrow$ which links a (starting) program configuration directly to a *final* configuration. Therefore the big-step semantics of a programming language is in some sense similar to the transitive closure of the small-step semantics.

Small-step semantics are especially useful in modeling systems with a high degree of non-determinism such as concurrent programming languages and in proofs of soundness for type systems, since small-step semantics can distinguish between programs that go wrong (by trying to perform an illegal operation such as adding an integer to a boolean) and programs which do not terminate because they enter an infinite loop. In contrast, a big-step semantics has the disadvantage that it cannot distinguish between a program that does not terminate (such as the program $\mu x.x$ in a lambda calculus extended with the fix-point operator μ) and a program that performs an illegal operation (such as the program 1 2 – the program which tries to apply the natural number 1 (which is not a function and therefore cannot be applied to anything) to the natural number 2 – in a lambda calculus extended with integers). The reason for which the big-step semantics cannot distinguish between these is that in both cases there does not exist a final configuration M such that $\mu x.x \Downarrow M$ or $1\ 2 \Downarrow M$. Furthermore, big-step semantics cannot be used to model non-determinism accurately: consider a language with a C-like *add-and-assign* operator += and the following statement: $x := (x \mathrel{+}= x + 1) + ((x \mathrel{+}= x + 2) + (x \mathrel{+}= x + 3))$. The *add-and-assign* operator evaluates the right-hand side, adds the result to the variable on the left-hand side and returns the new value of the variable. If the language has a non-deterministic + operator, then the evaluation of the three *add-and-assign* expressions can happen in any order. However, the following (slightly simplified) big-step rules which seem to naturally model the non-determinism of +:

$$\frac{}{V_1 + V_2 \Downarrow V}\ V = V_1 +_{Int} V_2 \qquad \frac{M \Downarrow V_1 \quad V_1 + N \Downarrow V}{M + N \Downarrow V} \qquad \frac{N \Downarrow V_2 \quad M + V_2 \Downarrow V}{M + N \Downarrow V}$$

will fail to capture all behaviors of the statement, since the side-effects of $x \mathrel{+}= x{+}1$ will never be taken into account *in the middle of the evaluation* of $((x \mathrel{+}= x + 2) + (x \mathrel{+}= x + 3))$. This is a known inherent limitation of big-step semantics which prevents it from being used to reason about concurrent systems. Known workarounds [7] that allow a certain degree of non-determinism in big-step semantics require adding additional syntax to the language (required only for the evaluation), which makes the semantics less structural.

Big-step semantics do have however a few notable advantages. First of all, they can be used to produce efficient interpreters and compilers [8] since there is no need to search for the redex to be reduced – instead, the result of a program is directly computed from the results of smaller programs. In contrast, an interpreter based on small-step semantics has to search at each step for a possible redex and then perform the update. Secondly, reasoning about programs and about the correctness of program transformations with a big-step semantics is easier [9,10].

Therefore, because both small-step semantics and big-step semantics each have their own set of advantages, it is desirable to have both types of semantics for a programming language. However, when both the small-step semantics and the big-step semantics are given for a language, the equivalence of the two semantics needs to be proven in order to be sure that the two semantics define the same language. We would like to obtain the advantages of both small-step semantics and big-step semantics, but without having to do this proof (or at least, not having to redo it for every programming languages being defined).

Therefore, we propose and investigate a transformation to *automatically* obtain the big-step semantics of a language from its small-step semantics. This allows in principle to enjoy both the advantages of the small-step semantics and those of the big-step semantics without having to manually maintain the two semantics and their proof of equivalence. Of course, this automation does not come without costs: in order for the transformation to yield an equivalent semantics, a number of assumptions must hold for the small-step semantics.

Our motivation for transforming small-step semantics into big-step semantics comes from our research on the K Semantic Framework [11], which is a framework for defining programming languages based on rewriting logic [12]. The K framework can be seen as a methodological fragment of rewriting logic with rewrite rules that describe small-step semantics and heating and cooling rules which describe under which contexts the rules can apply. We intend to generate standalone compilers for efficient execution and mechanized formal specifications for proof assistants in order to perform machine-assisted reasoning about programs. This transformation could serve as a starting point for both these directions.

In Section 2, we describe the meta-language we use for the small-step and big-step semantics. In Section 3, we formalize our transformation and present all of the assumptions under which it is sound and complete. In Section 4 we present a number of examples and in Section 5 we present related work. Section 6 contains a discussion and directions for further work.

2 Preliminaries

Before formalizing our transformation from small-step semantics to big-step semantics, we need a precise mathematical language (the meta-language) to describe such semantics. As previously discussed, the small-step semantics of a programming language is a binary relation $\rightarrow$ between program configurations. In the following, we model ground program configurations by an arbitrary algebra $\mathcal{A}$ over a signature Σ. Abstract configurations (i.e. configurations with variables, simply "configurations" from here on) are built from the signature Σ and a countably infinite set of variables $\mathcal{X}$ as expected. Substitutions σ are defined as expected and substitution application is written in suffix form. We use the letters M, N, P and their decorated counterparts (M_i, M^i, M_i^j, etc) as *meta-variables* in the meta-language; i.e., they can denote any particular program configuration.

Example 1. To define the untyped lambda calculus, we consider the signature $\Sigma = \{\mathsf{app}, \mathsf{fun}, x_0, \ldots, x_n, \ldots\}$. The constants $x_0, \ldots, x_n, \ldots$ denote the variables of lambda calculus, the binary symbol fun denotes functional abstraction and the binary symbol app denotes application. We follow the usual notations in lambda calculi and we write applications $\mathsf{app}(M, N)$ as juxtapositions MN and functional abstractions $\mathsf{fun}(x_i, M)$ as lambda-abstractions $\lambda x_i.M$. The algebra $\mathcal{A}$ is then the initial algebra of Σ. We assume that $\mathcal{A}$ is sorted such that the first argument of fun is always a constant x_i $(i \in N)$. □

We let $\mathcal{P}$ be a set of predicates. We assume that $\mathcal{P}$ contains the distinguished binary predicates $\rightarrow$ and $\Downarrow$ and the distinguished unary predicate $\downarrow$. The predicates $\rightarrow$ and respectively $\Downarrow$ are used in infix notation and the predicate $\downarrow$ is used in suffix notation. The predicate $\rightarrow$ is reserved for the small-step transition relation, the predicate $\Downarrow$ is used for the big-step relation and $\downarrow$ is used for denoting final configurations.

Example 2. Continuing the previous example, for call-by-value lambda calculus (CBV lambda calculus), the predicate $\downarrow$ (denoting final configurations) is defined to be true only for configurations that are either variables or lambda-abstractions:

$$M\downarrow \textit{ iff } \begin{cases} M = x_i & \textit{or} \\ M = \lambda x_i.N \end{cases} \quad \textit{(for some } i \in \mathbb{N},\ N \in \mathcal{A}\textit{)}.$$

In the context of lambda calculi, we also consider a predicate $\mathit{Subst}(M, x, N, P)$ which is true when P is a lambda-term obtained by substituting N for the variable x in M while avoiding name-capture. □

We model the small-step semantics as a set of inference rules R of the form

$$R = \frac{M_1 \rightarrow N_1, \ldots, M_n \rightarrow N_n}{M \rightarrow N}\ Q_1(\tilde{P}_1), \ldots, Q_m(\tilde{P}_m),$$

where $M, N, M_1, \ldots, M_n, N_1, \ldots, N_n$ are configurations, $\tilde{P}_1, \ldots, \tilde{P}_m$ are sequences of configurations and $Q_1, \ldots, Q_m \in \mathcal{P} \setminus \{\rightarrow, \Downarrow\}$ are predicates. The transition relation $\rightarrow$ associated to such a set of inference rules is the smallest relation which is closed by each inference rule, i.e., for each rule R as above and each substitution σ grounding for R, we have that if $M_1\sigma \rightarrow N_1\sigma, \ldots, M_n\sigma \rightarrow N_n\sigma$ and $Q_1(\tilde{P}_1\sigma), \ldots, Q_m(\tilde{P}_m\sigma)$ then $M\sigma \rightarrow N\sigma$.

Example 3. Continuing the previous example, the small-step semantics of call-by-value lambda calculus can be defined by the following set $S = \{R_1, R_2, R_3\}$ of inference rules:

$$(R_1)\ \frac{X \to X'}{XY \to X'Y} \qquad (R_2)\ \frac{Y \to Y'}{XY \to XY'}\ \mathrm{X}\!\downarrow \qquad (R_3)\ \frac{}{(\lambda x.X)Y \to Z}\ Y\!\downarrow, \mathit{Subst}(X, x, Y, Z)$$

Note that in the above rules, $x, X, X', Y, Y', Z \in \mathcal{X}$ are variables; we assume $x \in \mathcal{X}$ is sorted to be instantiated only with lambda-calculus variables $x_i \in \Sigma$. □

As a sanity check for the definition of small-step semantics, it is expected that $M\!\downarrow$ implies $M \not\to N$ for any N (i.e. configurations that are considered final cannot take any step). The reverse implication is not expected, since a configuration such as $x_0 x_1$ (application of the variable x_0 to the variable x_1) is *stuck* and cannot advance even if it not a final configuration. Similarly to small-step semantics, big-step semantics are modeled as a set of inference rules R of the form

$$R = \frac{M_1 \Downarrow N_1, \ldots, M_n \Downarrow N_n}{M \Downarrow N}\ Q_1(\tilde{P}_1), \ldots, Q_m(\tilde{P}_m),$$

where $M, N, M_1, \ldots, M_n, N_1, \ldots, N_n$ are configurations, $\tilde{P}_1, \ldots, \tilde{P}_m$ are sequences of configurations and $Q_1, \ldots, Q_m \in \mathcal{P} \setminus \{\to, \Downarrow\}$ are predicates. The relation $\Downarrow$ associated to such a set of inference rules is the smallest relation which is closed by each inference rule, i.e., for each rule R as above and each substitution σ grounding for R, we have that if $M_1\sigma \Downarrow N_1\sigma, \ldots, M_n\sigma \Downarrow N_n\sigma$ and $Q_1(\tilde{P}_1\sigma), \ldots, Q_m(\tilde{P}_m\sigma)$ then $M\sigma \Downarrow N\sigma$. Note that syntactically there is no difference between inference rules for big-step semantics and small-step semantics. The only difference is that in the small-step semantics, $\to$ is expected to denote one computation step while in the big-step semantics, $\Downarrow$ is expected to relate configurations to their associated final configuration.

Example 4. Continuing the previous examples, we consider the following big-step semantics $B = \{T_1, T_2\}$ for the call-by-value lambda calculus:

$$(T_1)\ \frac{}{X \Downarrow X}\ \mathrm{X}\!\downarrow \qquad (T_2)\ \frac{X \Downarrow \lambda x.X',\ Y \Downarrow Y',\ Z \Downarrow V}{XY \Downarrow V}\ \mathit{Subst}(X', x, Y', Z)$$

As for the small-step semantics, in the above rules $x, X, X', Y, Y', Z, V \in \mathcal{X}$ are variables and $x \in \mathcal{X}$ is sorted to be instantiated only with lambda-calculus variables $x_i \in \Sigma$. □

As a sanity check, it is expected for any big-step semantics that $M \Downarrow N$ implies $N\!\downarrow$. This will indeed be the case for the big-step semantics that are obtained by the algorithm that we describe next. In the following, we will write $\to^*$ for the reflexive-transitive closure of $\to$. The following definition captures the fact that a small-step semantics and a big-step semantics define the same programming language.

Definition 1. *A small-step semantics $\to$ and a big-step semantics $\Downarrow$ are* equivalent *when $M \to^* N$ and $N\!\downarrow$ hold if and only if $M \Downarrow N$ holds.*

The two semantics that we have defined above for call-by-value lambda calculus are equivalent (see, e.g., [13]):

Theorem 1. *The small-step semantics $\rightarrow$ defined by S in Example 3 is equivalent to the big-step semantics $\Downarrow$ defined by B in Example 4.*

Ideally, the big-step semantics and small-step semantics of a language should be equivalent in the sense of the definition above. However, producing a big-step semantics completely equivalent to the small-step semantics is sometimes impossible because, e.g., of non-determinism which can be described by small-step semantics but cannot be handled by big-step semantics (see discussion of non-determinism in Section 1). In such cases, it is desirable to have a slightly weaker link between the big-step semantics and the small-step semantics:

Definition 2. *A big-step semantics $\Downarrow$ is* sound *for a small-step semantics $\rightarrow$ if $M{\Downarrow}N$ implies $M{\rightarrow}^*N$ and $N{\downarrow}$.*

Note that if a small-step semantics $\rightarrow$ is equivalent to a big-step semantics $\Downarrow$, then it immediately follows that $\Downarrow$ is sound for $\rightarrow$. In all of the examples that we discuss in the rest of the paper, we will obtain big-step semantics that are fully equivalent to the initial small-step semantics. However, as discussed in Section 1, this cannot be the case for all languages. In such cases, it is desirable to establish that the two semantics satisfy the link in Definition 2.

3 From Small-Step Semantics to Big-Step Semantics

This section describes the transformation from small-step semantics to big-step semantics. We also give the class of small-step semantics for which our transformation is sound and complete in the sense of obtaining big-step semantics equivalent to the original small-step semantics.

3.1 The Transformation

The first idea that comes to mind when transforming a small-step semantics into a big-step semantics is to just add an explicit Transitivity-like inference rule. However, this defeats the purpose of having big-step semantics in the first case, since the $\rightarrow$ relation still explicitly appears in the inference system and must be reasoned about. Therefore, another approach is desirable.

Let $S = \{R_1, \ldots, R_k\}$ be a set of small-step inference rules $R_1, \ldots, R_k$ defining a small-step semantics. To S we associate the set $B(S) = \{R, R'_1, \ldots, R'_k\}$, where

$$R = \frac{}{V{\Downarrow}V}\; V{\downarrow}$$

and where

$$R'_i = \frac{M_1{\Downarrow}N_1, \ldots, M_n{\Downarrow}N_n, N{\Downarrow}V}{M{\Downarrow}V}\; Q_1(\tilde{P}_1), \ldots, Q_m(\tilde{P}_m)$$

for every inference rule

$$R_i = \frac{M_1 \rightarrow N_1, \ldots, M_n \rightarrow N_n}{M \rightarrow N} \; Q_1(\tilde{P}_1), \ldots, Q_m(\tilde{P}_m)$$

in S ($1 \leq i \leq k$). In the above rules, $V \in \mathcal{X}$ is a variable, $M, N, M_1, \ldots, M_n$, $N_1, \ldots, N_n$ are configurations, $\tilde{P}_1, \ldots, \tilde{P}_m$ are sequences of configurations and $Q_1, \ldots, Q_m$ are predicates.

Example 5. Continuing Example 3, we have that $B(S) = \{R, R'_1, R'_2, R'_3\}$ is:

$$R = \frac{}{V \Downarrow V} \; V\!\downarrow \qquad R'_1 = \frac{X \Downarrow X', X'Y \Downarrow V}{XY \Downarrow V} \qquad R'_2 = \frac{Y \Downarrow Y', XY' \Downarrow V}{XY \Downarrow V} \; X\!\downarrow$$

$$R'_3 = \frac{Z \Downarrow V}{(\lambda x.X)Y \Downarrow V} \; Y\!\downarrow, \mathit{Subst}(X, x, Y, Z)$$

□

Note that for the call-by-value lambda calculus that we have used as a running example, the big-step semantics $B(S) = \{R, R'_1, R'_2, R'_3\}$ obtained automatically from the small-step semantics $S = \{R_1, R_2, R_3\}$ by the transformation described above is slightly different from the manually designed big-step semantics $B = \{T_1, T_2\}$ (defined in Example 4). The difference is that the automatically generated rules R'_1, R'_2, R'_3 are synthesized into a single rule T_2 in the manually designed big-step semantics. It is not a surprise that the manually designed rules are slightly simpler than the automatically generated rules since the automated rules must be more generic. Note however that there is no redundancy in the generated rules and that the implementations of interpreters based on the two sets of rules would look very similar. We speculate that simplification rules could reduce the gap between the generated rules and the manually designed rules but we leave this as an open problem for further study.

3.2 The Assumptions

In order for the automatic derivation of the big-step semantics from the small-step semantics to produce a completely *equivalent semantics*, we require that the inference system S satisfies four assumptions. The big-step semantics are *sound* for the small-step semantics in the sense of Definition 2 when one of the four assumptions holds. In order to state the four assumptions, we need to notions of *star-soundness* and *star-completeness*, defined below.

Definition 3 (Star-sound Inference Rule). *A small-step inference rule*

$$R = \frac{M_1 \rightarrow N_1, \ldots, M_n \rightarrow N_n}{M \rightarrow N} \; Q_1(\tilde{P}_1), \ldots, Q_m(\tilde{P}_m)$$

is star-sound *if it still holds when $\rightarrow$ is replaced by $\rightarrow^*$, i.e. for any σ such that $M_1\sigma \rightarrow^* N_1\sigma, \ldots, M_n\sigma \rightarrow^* N_n\sigma$ and $Q_1(\tilde{P}_1\sigma), \ldots, Q_m(\tilde{P}_m\sigma)$ we have $M\sigma \rightarrow^* N\sigma$.*

Intuitively, star-soundness means that if one can take zero or more steps to reach N_i from M_i (for all $1 \leq i \leq n$), then M can also be reached from N in zero or more steps.

Definition 4 (Star-complete Inference Rule). *We say that a small-step inference rule*

$$R = \frac{M_1 \to N_1, \dots, M_n \to N_n}{M \to N} \, Q_1(\tilde{P}_1), \dots, Q_m(\tilde{P}_m)$$

is star-complete *w.r.t to a small-step semantics* $\to$ *if for every substitution* σ *such that* $M_1\sigma \to N_1\sigma, \dots, M_n\sigma \to N_n\sigma, Q_1(\tilde{P}_1\sigma), \dots, Q_m(\tilde{P}_m\sigma)$ *and* $N\sigma \to^* V$ *for some ground configuration* V *with* $V\!\downarrow$*, we have that there exists a substitution* σ' *which agrees with* σ *on* $\mathcal{V}ar(M, M_1, \dots, M_n)$ *such that* $N_1\sigma \to^* N_1\sigma', \dots, N_n\sigma \to^* N_n\sigma', Q_1(\tilde{P}_1\sigma'), \dots, Q_m(\tilde{P}_m\sigma')$ *and* $N_1\sigma'\!\downarrow, \dots, N_n\sigma'\!\downarrow$ *where the number of steps in each of the derivations* $N_i\sigma \to^* N_i\sigma'$ *(*$1 \le i \le n$*) is strictly smaller than the number of rewrite steps in the derivation* $N\sigma \to^* V$*.*

Intuitively, star-completeness means that if the rule can be used to start a terminating computation, then one can find terminating computations starting with M_i $(1 \le i \le n)$ as well. We are now ready to state our assumptions.

Assumption 1 (Ground Confluency). *The relation* $\to$ *induced by* S *is ground confluent: If* $M \to^* N_1$ *and* $M \to^* N_2$ *for some ground configurations* M, N_1, N_2*, then there exists a ground configuration* P *such that* $N_1 \to^* P$ *and* $N_2 \to^* P$*.*

Assumption 2. *For any ground configuration* M*, we have that* $M\!\downarrow$ *implies* $M \not\to N$ *for any ground configuration* N*.*

Assumption 3 (Star-soundness). *Any inference rule* $R \in S$ *is* star-sound *with respect to the rewrite relation* $\to$ *induced by* S*.*

Assumption 4 (Star-completeness). *Any inference rule* $R \in S$ *is* star-complete *with respect to the rewrite relation* $\to$ *induced by* S*.*

Our next theorem states that the transformation that we have presented is sound and complete in the sense that the resulting big-step semantics is equivalent to the original small-step semantics whenever S satisfies the above assumptions.

Theorem 2. *Let* $\to$ *be the small-step relation defined by* S *and let* $\Downarrow$ *be the big-step relation defined by* $B(S)$*. If* S *satisfies Assumptions 1, 2, 3, 4 defined in Subsection 3.2, then* $\to$ *and* $\Downarrow$ *are equivalent.*

Proof (Sketch). In one direction, the proof follows by induction on the number of small-steps taken and in the reverse direction by induction on the big-step proof tree.

The CBV lambda calculus in Example 3 satisfies the above assumptions:

Lemma 1. *The small-step semantics* $S = \{R_1, R_2, R_3\}$ *defined in Example 3 satisfies Assumptions 1, 2, 3, 4.*

Proof (Sketch.). It is well known (e.g., starting with the seminal result of Plotkin [13]) that Assumption 1 (confluence or "the Church-Rosser" property) holds for various (extensions of) lambda-calculi. Assumptions 2, 3, 4 follow by case analysis and induction.

Therefore we obtain immediately from Lemma 1 and Theorem 2:

Corollary 1. *The big-step semantics $\Downarrow$ defined by $B(S)$ in Example 5 is equivalent to the small-step semantics $\to$ defined by S in Example 3.*

It might seem that Assumptions 1, 2, 3, 4 are excessive. However, note that having these assumptions establishes a very strong link between the two semantics. If the big-step semantics should just be a sound approximation of the small-step semantics (i.e., when some behaviors of the small-step semantics can be discarded), then only Assumption 3 (star-soundness) is needed:

Theorem 3. *Let $\to$ be the small-step relation defined by S and let $\Downarrow$ be the big-step relation defined by $B(S)$. If S satisfies Assumption 3 defined in Subsection 3.2, then $\Downarrow$ is sound for $\to$ in the sense of Definition 2.*

Proof (sketch). By induction on the big-step proof tree.

4 Examples

4.1 Call-by-Name Lambda Calculus

We have already shown how our transformation works for CBV lambda calculus as a running example in Section 3. We consider the same signature as for CBV lambda calculus and the following set $S = \{R_1, R_2\}$ of small-step inference rules modeling call-by-name lambda calculus (CBN lambda calculus):

$$(R_1)\ \frac{X \to X'}{XY \to X'Y} \qquad\qquad (R_2)\ \frac{}{(\lambda x.X)Y \to Z}\ \mathit{Subst}(X, x, Y, Z)$$

The CBN big-step semantics obtained from the above definition is the set $B(S) = \{R, R'_1, R'_2\}$, where:

$$(R'_1)\ \frac{X \Downarrow X',\, X'Y \Downarrow V}{XY \Downarrow V}\ V{\downarrow} \qquad\qquad (R'_2)\ \frac{Z \Downarrow V}{(\lambda x.X)Y \Downarrow V}\ \mathit{Subst}(X, x, Y, Z)$$

It can be shown that the CBN lambda-calculus defined above satisfies Assumptions 1, 2, 3 and 4 and therefore the above transformation is sound and complete.

4.2 Call-by-Value Mini-ML

Mini-ML is a folklore language used for teaching purposes which extends lambda-calculus with some features like numbers, booleans, pairs, let-bindings or fix-points in order to obtain a language similar to (Standard) ML. We use a variant of Mini-ML where the abstract syntax is:

$$\begin{array}{lr}
\mathsf{Var} ::= & \text{variables} \\
\quad \mid x_0 \mid \ldots \mid x_n \mid \ldots & \\
\mathsf{Exp} ::= & \text{expressions} \\
\quad \mid \mathsf{Var} & \text{variable} \\
\quad \mid 0 \mid 1 \mid \ldots \mid n \mid \ldots & \text{natural number} \\
\quad \mid \mathsf{Exp} + \mathsf{Exp} & \text{arithmetic sum} \\
\quad \mid \lambda\mathsf{Var}.\mathsf{Exp} & \text{function definition} \\
\quad \mid \mu\mathsf{Var}.\mathsf{Exp} & \text{recursive definition} \\
\quad \mid \mathsf{Exp}\ \mathsf{Exp} & \text{function application} \\
\quad \mid \mathsf{let}\ \mathsf{Var} = \mathsf{Exp}\ \mathsf{in}\ \mathsf{Exp} & \text{let binding}
\end{array}$$

Here Exp and Var are sorts in the signature Σ, with Var being a subsort of Exp. The additional syntax can of course be desugared into pure lambda calculus, but we prefer to give its semantics directly in order to show how our transformation works. We define configurations to consist of expressions Exp and final configurations to be natural numbers $0, 1, \ldots, n, \ldots$ or function definitions $\lambda\mathsf{Var}.\mathsf{Exp}$. We consider the predicate $+(M, N, P)$ which holds when M, N and P are integers such that P is the sum of M and N. We define the small-step semantics of the language to be $S = \{R_1, \ldots, R_{10}\}$, where:

$$R_1 = \frac{X \to X'}{X + Y \to X' + Y} \qquad R_2 = \frac{Y \to Y'}{X + Y \to X + Y'}\ X{\downarrow} \qquad R_3 = \frac{}{X + Y \to Z}\ +(X, Y, Z)$$

$$R_4 = \frac{}{\mu x.X \to Z}\ \mathit{Subst}(X, x, \mu x.X, Z) \qquad R_5 = \frac{X \to X'}{XY \to X'Y} \qquad R_6 = \frac{Y \to Y'}{XY \to XY'}\ X{\downarrow}$$

$$R_7 = \frac{}{(\lambda x.X)Y \to Z}\ Y{\downarrow}, \mathit{Subst}(X, x, Y, Z) \qquad R_8 = \frac{X \to X'}{\mathsf{let}\ x = X\ \mathsf{in}\ Y \to \mathsf{let}\ x = X'\ \mathsf{in}\ Y}$$

$$R_9 = \frac{Y \to Y'}{\mathsf{let}\ x = X\ \mathsf{in}\ Y \to \mathsf{let}\ x = X\ \mathsf{in}\ Y'}\ X{\downarrow}$$

$$R_{10} = \frac{}{\mathsf{let}\ x = X\ \mathsf{in}\ Y \to Z}\ X{\downarrow}, Y{\downarrow}, \mathit{Subst}(Y, x, X, Z)$$

Rules R_1, R_2, R_3 describe integer arithmetic where the arguments to the plus operator are evaluated in order. Rule R_4 describes recursive definitions. The term $\mu x.M$ reduces to M where x is replaced by $\mu x.M$. This allows the definition of recursive functions. Note that $\mu x.M$ is not a final configuration. The next rules R_5, R_6, R_7 are those from the standard call-by-value lambda calculus and handle function application. Finally, rules R_8, R_9, R_{10} handle let bindings. Not surprisingly, the small-step semantics of Mini-ML satisfies Assumptions 1, 2, 3 and 4 as well. Therefore, by our result, the big-step semantics obtained by our transformation is equivalent to the small-step semantics.

4.3 IMP

Much like Mini-ML, IMP is a simple language used for teaching purposes. However, IMP is imperative and it usually features arithmetic and boolean expressions, variables, and statements such as assignment, conditionals and while-loops.

We define a variant of IMP with the following abstract syntax:

Var ::=	variables	
	$\mid x_0 \mid \ldots \mid x_n \mid \ldots$	
Exp ::=	expressions	
	$\mid$ Var	variable
	$\mid 0 \mid 1 \mid \ldots \mid n \mid \ldots$	natural number
	$\mid$ Exp + Exp	arithmetic sum
	$\mid$ Exp $\leq$ Exp	comparison
Seq ::=	sequence of statements	
	$\mid$ emp	empty sequence
	$\mid$ Var := Exp; Seq	assignment
	$\mid$ if Exp then Seq else Seq; Seq	conditional
	$\mid$ while Exp do Seq; Seq	loop
Pgm ::=	programs	
	$\mid$ Seq; Exp	execute statement return expression

For simplicity, we do not model booleans and we assume a C-like interpretation of naturals as booleans: any non-zero value is interpreted as truth and the comparison operator $\leq$ returns 0 (representing false) or 1 (representing true). We will define therefore the predicates *Zero*(n) and *NonZero*(n) which hold when n is a natural number equal to 0 (for *Zero*) and respectively when n is a natural number different from 0 (for *NonZero*).

Programs are described by terms of sort Pgm. As opposed to the previous examples of programming languages (all based on lambda-calculus), IMP programs do not run standalone; an IMP program runs in the presence of an *environment* which maps variables to natural numbers. Therefore the small-step relation will relate *configurations* which consist of a program and an environment:

Env ::=	environment (list of bindings)	
	$\mid \emptyset$	empty
	$\mid$ Var $\mapsto$ Nat, Env	non-empty
Cfg ::=	configuration	
	$\mid$ (Pgm, Env)	program + environment

As environments are essentially defined to be lists of pairs $x \mapsto n$ (for variables x and naturals n), there is no stopping a variable from appearing twice in an environment (making the environment map the same variable to potentially different natural numbers). We break ties by making the assumption that the binding which appears first in a list for a given variable is the right one. We define the predicate *Lookup*(e, x, n) to hold exactly when the variable x is mapped to the integer n by the environment e (breaking ties as described above). The predicate *Update*(e, x, n, e') holds when e' is the environment obtained from e by letting x map to n.

A final configuration (at which the computation stops) is a configuration in which the program is the empty sequence of statements (emp) followed by a fully evaluated expression (i.e., a natural number). Therefore the predicate $\downarrow$ will be true exactly of configurations of the form $(\mathsf{emp}; n, e)$ where e is any environment and n is a natural number. To simplify notations, when the sequence of statements is empty, we also write (N, e) instead of $(\mathsf{emp}; N, e)$.

To define our semantics, we also consider a predicate $Nat(N)$ which holds exactly when N is a natural number, a predicate $+(M, N, P)$ which relates any two natural number M and N to their sum P, a predicate $\leq (M, N, P)$ which is true when M, N are natural numbers and $M \leq N$ implies $P = 1$ and $M > N$ implies $P = 0$. Then the small-step semantics of IMP is given by $S = \{R_1, \ldots, R_{10}\}$, where the rules $R_1, \ldots R_{10}$ are described below. The rules for evaluating expressions are the following (recall that (M, e) is short for $(\mathsf{emp}; M, e)$):

$$R_1 = \frac{(X, e) \to (X', e)}{(X + Y, e) \to (X' + Y, e)} \qquad R_2 = \frac{(Y, e) \to (Y', e)}{(X + Y, e) \to (X + Y', e)}\ Nat(X)$$

$$R_3 = \frac{}{(X + Y, e) \to (Z, e)}\ +(X, Y, Z) \qquad R_4 = \frac{}{(x, e) \to (Y, e)}\ Lookup(e, x, Y)$$

Assignments work by first evaluating the expression and then updating the environment:

$$R_5 = \frac{(X, e) \to (X', e)}{((x := X); Z; Y, e) \to ((x := X'); Z; Y, e)}$$

$$R_6 = \frac{}{((x := X); Z; Y, e) \to (Z; Y, e')}\ Nat(X),\ Update(e, x, X, e')$$

Note that, in the last two rules above, Z matches the remaining sequence of statements while Y matches the expression representing the result of the program. The rules for the conditional and for the while loop are as expected:

$$R_7 = \frac{(X, e) \to (X', e)}{\begin{array}{c}(\mathsf{if}\ X\ \mathsf{then}\ Y_1\ \mathsf{else}\ Y_2; Z; Y, e) \to \\ (\mathsf{if}\ X'\ \mathsf{then}\ Y_1\ \mathsf{else}\ Y_2; Z; Y, e)\end{array}} \qquad R_8 = \frac{}{\begin{array}{c}(\mathsf{if}\ X\ \mathsf{then}\ Y_1\ \mathsf{else}\ Y_2; Z; Y, e) \to \\ (Y_1; Z; Y, e)\end{array}}\ Zero(X)$$

$$R_9 = \frac{}{\begin{array}{c}(\mathsf{if}\ X\ \mathsf{then}\ Y_1\ \mathsf{else}\ Y_2; Z; Y, e) \to \\ (Y_2; Z; Y, e)\end{array}}\ NonZero(X)$$

$$R_{10} = \frac{}{\begin{array}{c}(\mathsf{while}\ X\ \mathsf{do}\ X_0; Z; Y, e) \to \\ (\mathsf{if}\ X\ \mathsf{then}\ (X_0; \mathsf{while}\ X\ \mathsf{do}\ X_0; \mathsf{emp}) \\ \mathsf{else}\ Z; \mathsf{emp}; Y, e)\end{array}}$$

Note that in the above 10 rules, $X, X', Y, Y', Z, x, e, X_0, Y_1, Y_2 \in \mathcal{X}$ are variables. Furthermore, e is sorted to only match environments. The big-step semantics $B(S) = \{R, R'_1, \ldots, R'_{10}\}$ obtained through our transformation is:

$$R = \frac{}{V \Downarrow V}\ V\downarrow \qquad R'_1 = \frac{(X, e) \Downarrow (X', e),\ (X' + Y, e) \Downarrow V}{(X + Y, e) \Downarrow V}$$

$$R'_2 = \frac{(Y, e) \Downarrow (Y', e),\ (X + Y', e) \Downarrow V}{(X + Y, e) \Downarrow V}\ Nat(X) \qquad R'_3 = \frac{(Z, e) \Downarrow V}{(X + Y, e) \Downarrow V}\ +(X, Y, Z)$$

$$R'_4 = \frac{(Y, e) \Downarrow V}{(x, e) \Downarrow V}\ Lookup(e, x, Y) \qquad R'_5 = \frac{(X, e) \Downarrow (X', e),\ ((x := X'); Z; Y, e) \Downarrow V}{((x := X); Z; Y, e) \Downarrow V}$$

$$R'_6 = \frac{(Z; Y, e')\Downarrow V}{((x := X); Z; Y, e)\Downarrow V}\ \mathit{Nat}(X),\ \mathit{Update}(e, x, X, e')$$

$$R'_7 = \frac{(X, e)\Downarrow(X', e), (\mathsf{if}\ X'\ \mathsf{then}\ Y_1\ \mathsf{else}\ Y_2; Z; Y, e)\Downarrow V}{(\mathsf{if}\ X\ \mathsf{then}\ Y_1\ \mathsf{else}\ Y_2; Z; Y, e)\Downarrow V}$$

$$R'_8 = \frac{(Y_1; Z; Y, e)\Downarrow V}{(\mathsf{if}\ X\ \mathsf{then}\ Y_1\ \mathsf{else}\ Y_2; Z; Y, e)\Downarrow V}\ \mathit{Zero}(X)$$

$$R'_9 = \frac{(Y_2; Z; Y, e)\Downarrow V}{(\mathsf{if}\ X\ \mathsf{then}\ Y_1\ \mathsf{else}\ Y_2; Z; Y, e)\Downarrow V}\ \mathit{NonZero}(X)$$

$$R'_{10} = \frac{(\mathsf{if}\ X\ \mathsf{then}\ (X_0; \mathsf{while}\ X\ \mathsf{do}\ X_0; \mathsf{emp})\ \mathsf{else}\ Z; \mathsf{emp}; Y, e)\Downarrow V}{(\mathsf{while}\ X\ \mathsf{do}\ X_0; Z; Y, e)\Downarrow V}$$

It can be shown that the small-step semantics of IMP, as defined above, satisfies Assumptions 1, 2, 3 and 4. Therefore, by our result, the big-step semantics described above is equivalent to the small-step semantics. Note that Assumption 1 (confluence) is satisfied due to the deterministic nature of the language.

5 Related Work

Each semantic style has its own advantages and disadvantages. Work on addressing the disadvantages of big-step semantics includes [14], which presents a method to reduce verbosity called *pretty big step semantics* and [10,15], proposing methods of distinguishing between non-terminating and stuck configurations. However, it is surprising that little work has gone into (automatic) transformation of one style of semantics into another, given that proving that two semantics are equivalent can be nontrivial.

Huizing *et al.* [16] show how to automatically transform big-step semantics into small-step semantics. In some sense, they propose the exact inverse of our transformation with the goal of obtaining a semantics suitable for reasoning about concurrency. However, their transformation is not natural in the sense that the small-step semantics does not look like what would be written "by hand": instead, a stack is artificially added to program configurations in order to obtain the small-step semantics.

A line of work by Danvy et al. ([17,18,19,20]) shows that *functional implementations* of small-step semantics and big-step semantics can be transformed into each other via sound program transformations (like the well-known CPS transform). Their transformation looks similar to ours, but operates on interpreters of the language and is different from our work in several significant ways. Firstly, we address the transformation of the *semantics* itself (defined in a simple metalanguage) and not of the implementation of the semantics as an interpreter written in a functional language. This is in principle somewhat more powerful since tools other than interpreters (like program verifiers) can also take advantage of the newly obtained semantics. Secondly, because our transformation is completely formal, we are able to *prove* that it is correct (under a number of assumptions on the initial small-step semantics). This is in contrast to the above line of work, where the program transformations are performed manually and are shown to produce equivalent interpreters for a set of example languages. In particular, Danvy et al.

do not prove a meta-theorem stating that their transformation is sound for any language – they conclude that small-step machines can be mechanically transformed into big-step machines by generalizing from a set of example languages. On the other hand and in contrast to the present work, their transformation also works in reverse (obtaining small-step machines from big-step machines) and is more flexible than ours since the set of transformations is not fixed a-priori.

In order to formalize our transformation we define in Section 2 a *meta-language* for describing small-step (and big-step) structural operational semantics. Such meta-languages (also called *rule formats*) abound [21,22] in the literature. However these restricted rule formats are used to prove meta-theorems about well-definedness, compositionality, etc. and not for changing the style of the semantics.

6 Discussion and Further Work

We have presented a very simple syntactic trick for transforming small-step semantics into big-step semantics. We have analysed the transformations on several examples including both lambda-calculus based languages and an imperative language and we have identified four assumptions under which the transformation is sound and complete in the sense of yielding a big-step semantics which is equivalent to the initial small-step semantics. The confluence assumption (1) seems to be unavoidable since we have already shown that big-step semantics cannot deal with non-determinism; furthermore, it is already known for many variants and extensions of lambda-calculi that they satisfy confluence. The second assumption (2) regarding the definition of the final configuration ($\downarrow$) predicate does not seem to be too strong since it is what we expect of any sound small-step semantics. Finally, the last two assumptions (3 – star-soundness and 4 – star-completeness) are the most problematic in the sense that they are semantic assumptions which must be proven to hold. Note however that they hold for a variety of programming languages that we have analysed (imperative, functional) in different settings (call-by-name, call-by-value) and with both explicit (for the IMP language) and implicit substitutions (for the lambda-calculi). However, obtaining a sound syntactic approximation for the last two assumptions is an important step for further work.

We have also shown (Theorem 3) that a sound big-step semantics can be obtained from the initial small-step semantics under Assumption 3 (start-soundness) only. Having a sound (but not necessarily complete) big-step semantics can be acceptable in case the big-step semantics is used for generating a compiler, since a compiler is free to choose among the behaviors of the program. This could lead to obtaining an (efficient) compiler directly from the small-step semantics. As future work, we would like to investigate syntactic approximations of the four assumptions, the degree to which checking the assumptions can be automated and the possibility of generating variations of big-step semantics such as the ones in [10,14,23].

References

1. Strachey, C.: Towards a formal semantics. In: Formal Language Description Languages for Computer Programming, pp. 198–220. North-Holland (1966)

2. Floyd, R.W.: Assigning meanings to programs. In: Schwartz, J.T. (ed.) Mathematical Aspects of Computer Science, pp. 19–32. AMS, Providence (1967)
3. Hoare, C.A.R.: An axiomatic basis for computer programming. Communications of the ACM 12(10), 576–580 (1969)
4. Plotkin, G.D.: A structural approach to operational semantics. Technical Report DAIMI FN-19, University of Aarhus (1981)
5. Plotkin, G.D.: A structural approach to operational semantics. J. Log. Algebr. Program. 60-61, 17–139 (2004)
6. Kahn, G.: Natural semantics. In: Brandenburg, F.J., Wirsing, M., Vidal-Naquet, G. (eds.) STACS 1987. LNCS, vol. 247, pp. 22–39. Springer, Heidelberg (1987)
7. Mosses, P.D.: Modular structural operational semantics. J. Log. Algebr. Program. 60-61, 195–228 (2004)
8. Pettersson, M.: A compiler for natural semantics. In: Gyimóthy, T. (ed.) CC 1996. LNCS, vol. 1060, pp. 177–191. Springer, Heidelberg (1996)
9. Klein, G., Nipkow, T.: A machine-checked model for a Java-like language, virtual machine, and compiler. ACM T. Prog. Lang. Syst. 28, 619–695 (2006)
10. Leroy, X., Grall, H.: Coinductive big-step operational semantics. Information and Computation 207(2), 284–304 (2009)
11. Roşu, G., Şerbănuţă, T.F.: An overview of the K semantic framework. Journal of Logic and Algebraic Programming 79(6), 397–434 (2010)
12. Meseguer, J., Roşu, G.: The rewriting logic semantics project: A progress report. In: Owe, O., Steffen, M., Telle, J.A. (eds.) FCT 2011. LNCS, vol. 6914, pp. 1–37. Springer, Heidelberg (2011)
13. Plotkin, G.D.: Call-by-name, call-by-value and the lambda-calculus. Theor. Comput. Sci. 1(2), 125–159 (1975)
14. Charguéraud, A.: Pretty-big-step semantics. In: Felleisen, M., Gardner, P. (eds.) Programming Languages and Systems. LNCS, vol. 7792, pp. 41–60. Springer, Heidelberg (2013)
15. Cousot, P., Cousot, R.: Bi-inductive structural semantics. Information and Computation 207(2), 258–283 (2009)
16. Huizing, C., Koymans, R., Kuiper, R.: A small step for mankind. In: Dams, D., Hannemann, U., Steffen, M. (eds.) de Roever Festschrift. LNCS, vol. 5930, pp. 66–73. Springer, Heidelberg (2010)
17. Danvy, O.: Defunctionalized interpreters for programming languages. In: ICFP 2008, pp. 131–142. ACM, New York (2008)
18. Danvy, O., Millikin, K.: On the equivalence between small-step and big-step abstract machines: a simple application of lightweight fusion. Inf. Process. Lett. 106(3), 100–109 (2008)
19. Danvy, O., Millikin, K., Munk, J., Zerny, I.: Defunctionalized interpreters for call-by-need evaluation. In: Blume, M., Kobayashi, N., Vidal, G. (eds.) FLOPS 2010. LNCS, vol. 6009, pp. 240–256. Springer, Heidelberg (2010)
20. Danvy, O., Johannsen, J., Zerny, I.: A walk in the semantic park. In: Khoo, S.C., Siek, J.G. (eds.) PEPM, pp. 1–12. ACM (2011)
21. Groote, J.F., Mousavi, M., Reniers, M.A.: A hierarchy of SOS rule formats. In: Proceedings of SOS 2005, Lisboa, Portugal. ENTCS, Elsevier (2005)
22. Aceto, L., Fokkink, W., Verhoef, C.: Structural operational semantics. In: Handbook of Process Algebra, pp. 197–292. Elsevier (1999)
23. Uustalu, T.: Coinductive big-step semantics for concurrency. In: Vanderbauwhede, W., Yoshida, N. (eds.) Proceedings of PLACES 2013. EPTCS (2013) (to appear)