From Small-step Semantics to Big-step Semantics, Automatically

Ştefan Ciobâcă
Faculty of Computer Science, University "Alexandru Ioan Cuza"
Iaşi, Romania
stefan.ciobaca@gmail.com

***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, relation which 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 the small-step and the big-step semantics are needed for a language, the proof of a theorem stating that the two semantics are equivalent should also be provided in order to trust that they 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. We exemplify our transformation on a number of case studies.**

I. 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 programs 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] (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 given for a programming language. In such a case, the equivalence of the different semantics must be proven in order to be sure that they describe the same language.

In this article, we focus on two (sub)styles of operational semantics: small-step structural operational semantics [4] and big-step structural operational semantics [5] (also called natural semantics). Small-step semantics are given as a relation ($\rightarrow$) between terms which model program configurations consis-modellingcode 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 which 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 describes 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. In contrast to small-step semantics, which can model non-determinism and which differentiates programs that get stuck from programs that loop endlessly, big-step semantics cannot model non-determinism and cannot differentiate programs that get stuck from programs that loop forever. However, big-step semantics makes it a lot easier to reason about programs [6] and to generate efficient compilers [7].

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 (variation) of 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 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 be sound, a number of assumptions must hold of the small-step semantics.

In Section II, we describe the meta-language we use for the small-step and big-step semantics. In Section III, we formalize our transformation and present all of the assumptions under which it is sound. In Section IV we work out several case studies and in Section V we present related work. Section VI contains a discussion of the transformation and directions for further work. In the Appendix we give the full soundness proof of the proposed transformation.

II. 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 concrete program configurations by an arbitrary algebra $\mathcal{A}$ over a signature Σ. Abstract configurations are built from the signature Σ and over the 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 II.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 implicitly 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 II.2. *Continuing the previous example, for call-by-value lambda calculi, the predicate* $\downarrow$ *(denoting final configurations) is defined to be true only for configurations which are either variables or lambda-abstractions:*

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

In the context of lambda calculi, we also consider a predicate $Subst(M, x, N, P)$ *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

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

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 II.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:*

$$(R_1)\ \frac{X{\rightarrow}X'}{XY{\rightarrow}X'Y} \qquad (R_2)\ \frac{Y{\rightarrow}Y'}{XY{\rightarrow}XY'}\ \mathrm{X}{\downarrow}$$

$$(R_3)\ \frac{}{(\lambda x.X)Y{\rightarrow}Z}\ Y{\downarrow}, Subst(X, x, Y, Z)$$

Note that in the above rules, $x, X, X', Y, Y', Z \in \mathcal{X}$ *are variables; furthermore* $x \in \mathcal{X}$ *is (implicitly) 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 $M \not\rightarrow N$ for any N (i.e. configurations which are considered final cannot take any step). The reverse implication is not expected, since a configuration such as x_0x_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

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

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 $Q_1, \ldots, Q_m \in \mathcal{P} \setminus \{\rightarrow, \Downarrow\}$ 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, $\rightarrow$ 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 II.4. *Continuing the previous examples, we consider the following big-step semantics* $B = \{T_1, T_2\}$ *for the call-by-value lambda calculus:*

$$(T_1)\ \frac{}{X{\Downarrow}X}\ \mathrm{X}{\downarrow}$$

$$(T_2)\ \frac{X{\Downarrow}\lambda x.X', Y{\Downarrow}Y', Z{\Downarrow}V}{XY{\Downarrow}V}\ Subst(X', x, Y', Z)$$

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 (implicitly) 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 $\rightarrow^*$ for the reflexive-transitive closure of $\rightarrow$. The following definition

captures the fact that a small-step semantics and a big-step semantics define the same programming language.

Definition II.1. *We say that a small-step semantics $\rightarrow$ and a big-step semantics $\Downarrow$ are* equivalent *when the conjunction of $M{\rightarrow}^*N$ and of $N{\downarrow}$ is equivalent to $M{\Downarrow}N$.*

As a folklore result, we have that the two semantics that we have defined above for call-by-value lambda calculus are equivalent:

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

III. 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 in the sense of obtaining big-step semantics equivalent to the original small-step semantics.

A. The Transformation

Let $S = \{R_1, \ldots, R_k\}$ be a set of small-step inference rules $R_1, \ldots, R_k$ defininig 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 III.1. *Continuing Example II.3, where $S = \{R_1, R_2, R_3\}$ was defined to be:*

$$(R_1)\ \frac{X{\rightarrow}X'}{XY{\rightarrow}X'Y} \qquad (R_2)\ \frac{Y{\rightarrow}Y'}{XY{\rightarrow}XY'}\ \mathrm{X}{\downarrow}$$

$$(R_3)\ \frac{}{(\lambda x.X)Y{\rightarrow}Z}\ Y{\downarrow}, Subst(X, x, Y, Z)$$

we have that $B(S) = \{R, R'_1, R'_2, R'_3\}$ is the set:

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

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

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

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 II.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. We speculate that simplification rules could reduce this gap between the generated rules and the manually designed rules but we leave this as an open problem for further study.

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

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

The Appendix contains the proof of this theorem. By the above theorem, we obtain immediately:

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

B. The Assumptions

In order for the automatic derivation of the big-step semantics from the small-step semantics to be sound, we require that the inference system S satisfies the following assumptions:

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

Assumption III.2. *For any concrete configuration M, we have that $M{\downarrow}$ implies $M{\not\rightarrow}N$ for any concrete configuration N.*

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

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

We define *star-soundness* and *star-completeness* below.

Definition III.1 (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 is still sound when $\rightarrow$ is replaced by ${\rightarrow}^*$, i.e.: for any substitution 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 that* $M\sigma{\rightarrow}^*N\sigma$.

Definition III.2 (Star-complete Inference Rule)**.** *We say that 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-complete *w.r.t to a small-step semantics* $\rightarrow$ *if for every substitution* σ *such that* $M_1\sigma{\rightarrow}N_1\sigma,\ldots,M_n\sigma{\rightarrow}N_n\sigma$, $Q_1(\tilde{P}_1\sigma),\ldots,Q_m(\tilde{P}_m\sigma)$ *and* $N\sigma{\rightarrow}^*V$ *for some concrete configuration* V *with* $V{\downarrow}$*, we have that there exists a substitution* σ' *which agrees with* σ *on* $\mathcal{V}ar(M,M_1,\ldots,M_n)$ *such that* $N_1\sigma{\rightarrow}^*N_1\sigma',\ldots,N_n\sigma{\rightarrow}^*N_n\sigma', Q_1(\tilde{P}_1\sigma'),\ldots,Q_m(\tilde{P}_m\sigma')$ *and* $N_1\sigma'{\downarrow},\ldots,N_n\sigma'{\downarrow}$ *where the number of steps in each of the derivations* $N_i\sigma{\rightarrow}^*N_i\sigma'$ $(1\leq i\leq n)$ *is strictly smaller than the number of rewrite steps in the derivation* $N\sigma{\rightarrow}^*V$.

The call-by-value lambda calculus defined in Example II.3 satisfies the above assumptions:

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

Sketch.:

It is well known (e.g. starting with the seminal result of Plotkin [8]) that assumption III.1 (confluence or "the Church-Rosser" property) hold of various (extensions of) lambda-calculi. Assumptions III.2, III.3, III.4 follow easily by case analysis and induction.

■

IV. Case Studies

A. Call-by-Name Lambda Calculus

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

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

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

$$(R'_1)\ \frac{X{\Downarrow}X', X'Y{\Downarrow}V}{XY{\Downarrow}V}\ V{\downarrow}$$

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

It can be shown that the call-by-name lambda-calculus defined above satisfies Assumptions III.1, III.2, III.3 and III.4 as well and therefore the above transformation is sound.

B. 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:

Var	::=		variables
	|	$x_0 \mid \ldots \mid x_n \mid \ldots$	
Exp	::=		expressions
	|	Var	variable
	|	$0 \mid 1 \mid \ldots \mid n \mid \ldots$	natural number
	|	Exp + Exp	arithmetic sum
	|	λVar.Exp	function definition
	|	μVar.Exp	recursive definition
	|	Exp Exp	function application
	|	let Var = Exp in Exp	let binding

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 λVar.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:

$$R_1 = \frac{X{\rightarrow}X'}{X+Y{\rightarrow}X'+Y}$$

$$R_2 = \frac{Y{\rightarrow}Y'}{X+Y{\rightarrow}X+Y'}\ X{\downarrow}$$

$$R_3 = \frac{}{X+Y{\rightarrow}Z}\ +(X,Y,Z)$$

$$R_4 = \frac{}{\mu x.X{\rightarrow}Z}\ Subst(X,x,\mu x.X,Z)$$

$$R_5 = \frac{X{\rightarrow}X'}{XY{\rightarrow}X'Y} \qquad R_6 = \frac{Y{\rightarrow}Y'}{XY{\rightarrow}XY'}\ X{\downarrow}$$

$$R_7 = \frac{}{(\lambda x.X)Y{\rightarrow}Z}\ Y{\downarrow}, Subst(X,x,Y,Z)$$

$$R_8 = \frac{X{\rightarrow}X'}{\text{let } x=X \text{ in } Y{\rightarrow}\text{let } x=X' \text{ in } Y}$$

$$R_9 = \frac{Y{\rightarrow}Y'}{\text{let } x=X \text{ in } Y{\rightarrow}\text{let } x=X \text{ in } Y'}\ X{\downarrow}$$

$$R_{10} = \frac{}{\text{let } x=X \text{ in } Y{\rightarrow}Z}\ X{\downarrow}, Y{\downarrow}, Subst(Y,x,X,Z)$$

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. By our transformation, we obtain the following (equivalent) big-step semantics $B(S) = \{R, R'_1, \ldots, R'_{10}\}$:

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

$$R'_2 = \frac{Y \Downarrow Y', X + Y' \Downarrow V}{X + Y \Downarrow V}\, X\!\downarrow$$

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

$$R'_4 = \frac{Z \Downarrow V}{\mu x.X \Downarrow V}\, \mathit{Subst}(X, x, \mu x.X, Z)$$

$$R'_5 = \frac{X \Downarrow X', X'Y \Downarrow V}{XY \Downarrow V} \qquad R'_6 = \frac{Y \Downarrow Y', XY' \Downarrow V}{XY \Downarrow V}\, X\!\downarrow$$

$$R'_7 = \frac{Z \Downarrow V}{(\lambda x.X)Y \Downarrow V}\, Y\!\downarrow, \mathit{Subst}(X, x, Y, Z)$$

$$R'_8 = \frac{X \Downarrow X', \mathsf{let}\ x = X'\ \mathsf{in}\ Y \Downarrow V}{\mathsf{let}\ x = X\ \mathsf{in}\ Y \Downarrow V}$$

$$R'_9 = \frac{Y \Downarrow Y', \mathsf{let}\ x = X\ \mathsf{in}\ Y' \Downarrow V}{\mathsf{let}\ x = X\ \mathsf{in}\ Y \Downarrow V}\, X\!\downarrow$$

$$R'_{10} = \frac{Z \Downarrow V}{\mathsf{let}\ x = X\ \mathsf{in}\ Y \Downarrow V}\, X\!\downarrow, Y\!\downarrow, \mathit{Subst}(Y, x, X, Z)$$

Not surprisingly, the small-step semantics of Mini-ML satisfies Assumptions III.1, III.2, III.3 and III.4 as well. Therefore, by our result, the big-step semantics described above is equivalent to the small-step semantics.

C. 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	
	|	$x_0 \mid \ldots \mid x_n \mid \ldots$	
Exp	::=	expressions	
	|	Var	variable
	|	$0 \mid 1 \mid \ldots \mid n \mid \ldots$	natural number
	|	Exp + Exp	arithmetic sum
	|	Exp ≤ Exp	comparison
Seq	::=	sequence of statements	
	|	emp	empty sequence
	|	Var := Exp; Seq	assignment
	|	if Exp then Seq else Seq; Seq	conditional
	|	while Exp do Seq; Seq	loop
Pgm	::=	programs	
	|	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 as defined below:

Env	::=	environment (list of bindings)	
	|	∅	empty
	|	Var ↦ Nat, Env	non-empty
Cfg	::=	configuration	
	|	(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 $\mathit{Lookup}(e, x, n)$ to hold exactly when the variable x is mapped to the integer n by the environment e (breaking ties as described above). The predicate $\mathit{Update}(e, x, n, e')$ 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 $\mathit{Nat}(N)$ 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 $(\mathsf{emp}; M, e)$):

$$R_1 = \frac{(X, e) \rightarrow (X', e)}{(X + Y, e) \rightarrow (X' + Y, e)}$$

$$R_2 = \frac{(Y, e) \rightarrow (Y', e)}{(X + Y, e) \rightarrow (X + Y', e)}\ \mathit{Nat}(X)$$

$$R_3 = \frac{}{(X + Y, e) \rightarrow (Z, e)}\ +(X, Y, Z)$$

$$R_4 = \frac{}{(x, e) \rightarrow (Y, e)}\ \mathit{Lookup}(e, x, Y)$$

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

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

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

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:

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

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

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

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

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. Next we describe the big-step semantics $B(S) = \{R, R'_1, \ldots, R'_{10}\}$ obtained through our transformation:

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

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

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

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

$$R'_4 = \frac{(Y, e) \Downarrow V}{(x, e) \Downarrow V}\ \mathit{Lookup}(e, x, Y)$$

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

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

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

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

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

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

It can be shown that the small-step semantics of IMP, as defined above, satisfies Assumptions III.1, III.2, III.3 and III.4. Therefore, by our result, the big-step semantics described above is equivalent to the small-step semantics. Note that Assumption III.1 (confluence) is satisfied due to the deterministic nature of the language. If the language had side-effects and a non strict $+$ operator, confluence would no longer hold and therefore our result would not hold. Note however that this is not a weakness of our transformation, but of the big-step semantics style itself because it cannot faithfully capture non-determinism as we discuss in Section VI.

V. Related Work

There are three large classes of semantic style for programming languages: operational, axiomatic [3], [2] and denotational [1]. Initially, operational semantics were considered

inferior to denotation and axiomatic semantics, but Plotkin [4] revived the interest in operational semantics by introducing (small-step) *structural* operational semantics with an emphasis on the structure of the abstract syntax of the language. Later, Kahn [5] introduced big-step semantics (also called natural semantics).

Each semantic style has its own advantages and disadvantages. Small-step semantics can easily capture non-determinism in concurrent execution, while big-step semantics allow to easily obtain efficient compilers [7] and perform (mechanical) reasoning [9]. On the other hand, big-step semantics does not make the distinction between programs that block because they *go wrong* (e.g. by trying to add an integer to a boolean) and programs that do not terminate (due to the presence of e.g. an infinite loop), although there is work addressing this issue [10].

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. The only other closely related work that we are aware of is that of Huizing *et al.* [11] who 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 and others ([12], [13], [14], [15]) show that *functional implementations* of small-step semantics and big-step semantics can be transformed into each other via program manipulation techniques. However, in this paper, we address the transformation of the *semantics* itself and not of its implementations in a functional language. Transforming the semantics is more powerful, as the semantics can not only be implemented to obtain an interpreter but it can be used to reason about programs.

In order to formalize our transformation we define in Section II a *meta-language* for describing small-step (and big-step) structural operational semantics. Such meta-languages (also called *rule formats*) abound [16], [17] 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.

VI. 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 case studies 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. 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, big-step semantics has the disadvantage that it cannot distinguish between a program that does not terminate (such as $\mu x.x$ in Mini-ML) and a program which performs an illegal operation (such as 12 in Mini-ML – 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). The reason that the big-step semantics cannot distinguish between these is that in both cases there does exist a final configurations M such that $\mu x.x \Downarrow M$ or $12 \Downarrow M$. Furthermore, big-step semantics cannot be used to model non-determinism: Consider a language extending IMP with a C-like *add-and-assign* operator $+=$ and the following statement: $x := (x \mathrel{+}= x{+}1){+}((x \mathrel{+}= x{+}2){+}(x \mathrel{+}= x{+}3))$. If the order of evaluation of the arguments of the $+$ operator is not fixed by the language, the following big-step rules:

$$R_1 = \frac{}{V_1 + V_2 \Downarrow V} +(V_1, V_2, V)$$

$$R_2 = \frac{M \Downarrow V_1, V_1 + N \Downarrow V}{M + N \Downarrow V}$$

$$R_3 = \frac{N \Downarrow V_2, M + V_2 \Downarrow V}{M + N \Downarrow V}$$

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. However, big-step semantics does have a few notable advantages. First of all, it can be used to produce efficient interpreters and compilers [7] 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 in particular about the correctness of program transformations with a big-step semantics is easier [6], [10].

The two advantages above are actually our motivation for transforming small-step semantics into big-step semantics. Our research concerns the K Semantic Framework [18], which is a framework for defining programming languages based on rewriting logic [19]. 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. Currently the language definitions written in K are compiled to Maude [20] for execution and state-space exploration, but we also 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.

For our transformation to be sound, we have identified a number of semantic assumptions which the language being defined must satisfy. The confluence assumption (III.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 (III.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 (III.3 – star-soundness and III.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 of 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.

Another direction for further work is to drop the requirement of confluence (Assumption III.1). Of course, as we have already shown, big-step semantics cannot handle non-determinism. Therefore, the resulting natural semantics would necessarily approximate the initial small-step semantics. This can be acceptable in case the big-step semantics are used for generating a compiler, since a compiler is free to choose among the behaviors of the program. Note that the proof of the reverse implication in Theorem III.1 only requires Assumption III.3. Therefore, this direction of research seems particularly fruitful, since combined with a relaxed direct implication, it would lead to obtaining an (efficient) compiler directly from the small-step semantics with fewer assumptions.

REFERENCES

[1] C. Strachey, "Towards a formal semantics," in *Formal Language Description Languages for Computer Programming*. North Holland, 1966, pp. 198–220.

[2] R. W. Floyd, "Assigning meanings to programs," in *Mathematical Aspects of Computer Science*, ser. Proceedings of Symposia in Applied Mathematics, J. T. Schwartz, Ed., vol. 19. Providence, Rhode Island: American Mathematical Society, 1967, pp. 19–32.

[3] C. A. R. Hoare, "An axiomatic basis for computer programming," *Communications of the ACM*, vol. 12, no. 10, pp. 576–580 and 583, October 1969.

[4] G. D. Plotkin, "A structural approach to operational semantics," *J. Log. Algebr. Program.*, vol. 60-61, pp. 17–139, 2004.

[5] G. Kahn, "Natural semantics," in *STACS 87*, ser. LNCS, F. Brandenburg, G. Vidal-Naquet, and M. Wirsing, Eds. Springer Berlin / Heidelberg, 1987, vol. 247, pp. 22–39. [Online]. Available: http://dx.doi.org/10.1007/BFb0039592

[6] G. Klein and T. Nipkow, "A machine-checked model for a java-like language, virtual machine, and compiler," *ACM Trans. Program. Lang. Syst.*, vol. 28, pp. 619–695, July 2006. [Online]. Available: http://doi.acm.org/10.1145/1146809.1146811

[7] M. Pettersson, "A compiler for natural semantics," in *Compiler Construction, 6th International Conference, CC'96, Linköping, Sweden, April 24-26, 1996, Proceedings*, ser. LNCS, T. Gyimóthy, Ed., vol. 1060. Springer, 1996, pp. 177–191.

[8] G. D. Plotkin, "Call-by-name, call-by-value and the lambda-calculus," *Theor. Comput. Sci.*, vol. 1, no. 2, pp. 125–159, 1975.

[9] X. Leroy, "A formally verified compiler back-end," *Journal of Automated Reasoning*, vol. 43, no. 4, pp. 363–446, 2009.

[10] X. Leroy and H. Grall, "Coinductive big-step operational semantics," *Information and Computation*, vol. 207, no. 2, pp. 284–304, 2009.

[11] C. Huizing, R. Koymans, and R. Kuiper, "A small step for mankind," in *Concurrency, Compositionality, and Correctness*, ser. LNCS, D. Dams, U. Hannemann, and M. Steffen, Eds. Springer Berlin / Heidelberg, 2010, vol. 5930, pp. 66–73. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-11512-7_5

[12] O. Danvy, "Defunctionalized interpreters for programming languages," in *Proceedings of the 13th ACM SIGPLAN international conference on Functional programming*, ser. ICFP '08. New York, NY, USA: ACM, 2008, pp. 131–142. [Online]. Available: http://doi.acm.org/10.1145/1411204.1411206

[13] O. Danvy and K. Millikin, "On the equivalence between small-step and big-step abstract machines: a simple application of lightweight fusion," *Inf. Process. Lett.*, vol. 106, no. 3, pp. 100–109, 2008.

[14] O. Danvy, K. Millikin, J. Munk, and I. Zerny, "Defunctionalized interpreters for call-by-need evaluation," in *FLOPS*, ser. Lecture Notes in Computer Science, M. Blume, N. Kobayashi, and G. Vidal, Eds., vol. 6009. Springer, 2010, pp. 240–256.

[15] O. Danvy, J. Johannsen, and I. Zerny, "A walk in the semantic park," in *PEPM*, S.-C. Khoo and J. G. Siek, Eds. ACM, 2011, pp. 1–12.

[16] J. F. Groote, M. Mousavi, and M. A. Reniers, "A hierarchy of SOS rule formats," in *Proceedings of SOS'05*, ser. ENTCS. Lisboa, Portugal: Elsevier Science B.V., 2005.

[17] L. Aceto, W. Fokkink, and C. Verhoef, "Structural operational semantics," in *Handbook of Process Algebra*. Elsevier, 1999, pp. 197–292.

[18] G. Roşu and T. F. Şerbănuţă, "An overview of the K semantic framework," *Journal of Logic and Algebraic Programming*, vol. 79, no. 6, pp. 397–434, 2010.

[19] J. Meseguer and G. Roşu, "The rewriting logic semantics project: A progress report," in *Proceedings of the 17th International Symposium on Fundamentals of Computation Theory (FCT'11)*, ser. LNCS, vol. 6914. Springer, 2011, pp. 1–37, invited talk.

[20] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. L. Talcott, "Introduction," in *All About Maude*, ser. Lecture Notes in Computer Science, M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. L. Talcott, Eds., vol. 4350. Springer, 2007, pp. 1–28.

APPENDIX

This appendix contains the full proof of Theorem III.1:

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

Proof:

We first recall the definition of $B(S)$. Let $S = \{R_1, \ldots, R_k\}$ be the set of small-step inference rules $R_1, \ldots, R_k$ defininng the small-step semantics $\rightarrow$. The set $B(S)$ is defined to $B(S) = \{R, R'_1, \ldots, R'_k\}$, where

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

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 ($1 \leq i \leq k$). Having recalled the definition of $B(S)$, we proceed to prove the theorem. We assume that S satisfies Assumptions III.1, III.2, III.3 and III.4 and we show that $A \rightarrow^* C$ and $C\downarrow$ is equivalent to $A \Downarrow C$ for any (ground) configurations A and C. We prove the two directions of the equivalence separately:

$\Rightarrow$ We assume that $A \rightarrow^* C$ and that $C\downarrow$ and we show that $A \Downarrow C$ by induction on the length of the derivation $A \rightarrow^* C$.

Base case. If $A = C$, then $A\downarrow$ is true by the hypothesis. We obtain that $A \Downarrow C$ is true by rule $R \in B(S)$.

Inductive case. Otherwise, there exists B such that $A \rightarrow B \rightarrow^* C$ and such that the length of the derivation $B \rightarrow^* C$ is smaller than the length of the derivation $A \rightarrow^* C$. We can therefore apply the induction hypothesis on B and C to obtain that $B \Downarrow C$.

As $A \rightarrow B$, we have that there exists a rule $R_i \in S$ (for some $1 \leq i \leq k$) with

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

and a substitution σ grounding for R_i such that $A = M\sigma$, $B = N\sigma$, $M_j\sigma \rightarrow N_j\sigma$ for all $1 \leq j \leq n$ and $Q_j(\tilde{P}_j\sigma)$ for all $1 \leq j \leq m$.

By Assumption III.4, we have that R_i is star-complete. Therefore, by Definition III.2 there exists a substitution σ' compatible with σ on $\mathcal{V}ar(M, M_1, \ldots, M_n)$ such that $M_1\sigma = M_1\sigma' \rightarrow^* N_1\sigma', \ldots, M_n\sigma = M_n\sigma' \rightarrow^* N_n\sigma'$, $N_1\sigma'\downarrow, \ldots, N_n\sigma'\downarrow$ and

$$Q_1(\tilde{P}_1\sigma'), \ldots, Q_m(\tilde{P}_m\sigma'). \tag{1}$$

Furthermore, the length of the derivation $M_j\sigma' \rightarrow^* N_j\sigma'$ is strictly smaller than the length of the derivation $M\sigma = M\sigma' \rightarrow^* N\sigma'$. By applying the induction hypothesis, we obtain that

$$M_j\sigma' \Downarrow N_j\sigma' \text{ for all } 1 \leq j \leq n. \tag{2}$$

By Assumption III.3, we have that R_i is star-sound and therefore $M\sigma' \rightarrow^* N\sigma'$. By hypothesis, we have that $M\sigma' = M\sigma$ and that $M\sigma \rightarrow N\sigma$, which trivially implies $M\sigma' \rightarrow^* N\sigma'$. By Assumption III.1, we have that $\rightarrow$ is ground-confluent and, as $N\sigma \leftarrow^* M\sigma' \rightarrow^* N\sigma'$, we have that there exists a concrete configuration P such that $N\sigma \rightarrow^* P$ and $N\sigma' \rightarrow^* P$.

Furthermore, by hypothesis, we have that $N\sigma \rightarrow^* C$ and that $C\downarrow$. By applying ground-confluence of $\rightarrow$ on $N\sigma \rightarrow^* P$ and $N\sigma \rightarrow^* C$, we obtain that there exists a concrete configuration C' such that $C \rightarrow^* C'$ and $P \rightarrow^* C'$. But by Assumption III.2, we have that $C\downarrow$ implies $C \not\rightarrow$ and therefore the concrete configuration C' must be the same as C: $C' = C$. We obtain that $P \rightarrow^* C$. We already have $N\sigma' \rightarrow^* P$ and therefore

$$N\sigma' \rightarrow^* C \tag{3}$$

by transitivity.

We consider the substitution $\sigma'' = \sigma' \uplus \{V \mapsto C\}$, i.e. the same substitution as σ' but which sends the distinguished variable V in rule R'_i to C ($\uplus$ denotes disjoint union). We have that all of the precedents of rule R'_i are met by Equation (2), Equation (3) and Equation (1) and therefore the conclusion follows: $M\sigma'' \Downarrow C$. But $M\sigma'' = M\sigma' = M\sigma = A$ and we have therefore what we wanted to prove:

$$A \Downarrow C.$$

$\Leftarrow$ We assume that $A \Downarrow C$ and we show that $A \rightarrow^* C$ and that $C\downarrow$. We make the proof by induction on the proof tree of $A \Downarrow C$.

If the last step of the proof that $A \Downarrow C$ involves the rule $R \in B(S)$, we have that $A = C$ and that $C\downarrow$, which trivially gives the conclusion.

If the last step of the proof that $A \Downarrow C$ involves another rule $R'_i \in B(S)$(with $1 \leq i \leq k$) where

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

we have that there exists a substitution σ grounding for R'_i such that $A = M\sigma$, $C = V\sigma$, $M_1\sigma \Downarrow N_1\sigma, \ldots, M_n\sigma \Downarrow N_n\sigma$, $N\sigma \Downarrow V\sigma$ and $Q_1(\tilde{P}_1\sigma), \ldots, Q_m(\tilde{P}_m\sigma)$. By the induction hypothesis, we obtain that $M_1\sigma \rightarrow^* N_1\sigma, \ldots, M_n\sigma \rightarrow^* N_n\sigma$, $N\sigma \rightarrow^* V\sigma$ and $V\sigma\downarrow$.

As $R'_i \in B(S)$, we have by the construction of $B(S)$ that the rule

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

must be in S: $R_i \in S$. By Assumption III.3, $\to$ is star-sound and, by Definition III.1, we immediately conclude that $M\sigma \to^* N\sigma$. But $N\sigma \to^* V\sigma$ and therefore $M\sigma \to^* V\sigma$. Furthermore we already have that $A = M\sigma$, $C = V\sigma$ and that $V\sigma\!\downarrow$. Therefore, we immediately obtain our conclusion: $A \to^* C$ and $C\!\downarrow$.

■