Trace-Based Coinductive Operational Semantics for While

Big-step and Small-step, Relational and Functional Styles

Keiko Nakata and Tarmo Uustalu

Institute of Cybernetics at Tallinn University of Technology,
Akadeemia tee 21, EE-12618 Tallinn, Estonia, {keiko,tarmo}@cs.ioc.ee

Abstract. We present four coinductive operational semantics for the While language accounting for both terminating and non-terminating program runs: big-step and small-step relational semantics and big-step and small-step functional semantics. The semantics employ traces (possibly infinite sequences of states) to record the states that program runs go through. The relational semantics relate statement-state pairs to traces, whereas the functional semantics return traces for statement-state pairs. All four semantics are equivalent. We formalize the semantics and their equivalence proofs in the constructive setting of Coq.

1 Introduction

Now and then we must program a partially recursive function whose domain of definedness we cannot decide or is undecidable, e.g., an interpreter. Reactive programs such as operating systems and data base systems are not supposed to terminate. To reason about such programs properly, we need semantics that account for both terminating and non-terminating program runs. Compilers, for example, should preserve both terminating and non-terminating behaviors of source programs [13, 10]. Standard operational semantics ignore (or say too little about) non-terminating runs, so finer semantic accounts are necessary.

In this paper, we present four coinductive semantics for the While language that we claim to be both adequate for reasoning about non-terminating runs as well as well-designed. They represent four different styles of operational semantics: big-step and small-step relational and big-step and small-step functional semantics. Our semantics are based on traces, defined coinductively as possibly infinite non-empty sequences of states. What is more, the evaluation and normalization relations and functions are also coinductive/corecursive. The functional semantics are constructively possible thanks to the fact that in the trace-based setting, While becomes a total rather than partial language (every run defines a trace, even if it may be infinite). All four semantics are constructively equivalent. We have formalized our development in the Coq proof assistant, using the Ssreflect syntax extension, see `http://cs.ioc.ee/~keiko/majas.tar.gz`.

It might be objected against this paper that the results are unsurprising, since the semantics appear simple and enjoy all expected properties. They are

simple indeed, but in the case of the two big-step semantics, this is a consequence of very careful design decisions. As a matter of fact, getting coinductive big-step semantics right is tricky, and in this situation it is really fortunate that simple solutions are available. In the paper, we discuss some of the design considerations and also show some design options that we rejected deliberately. Previous work in the literature [9, 6, 14] also contains some designs that are more complicated than ours or fail to have some clearly desirable properties or both. A skeptical reader may also worry that While is a toy language. We argue that While is sufficient for highlighting all important issues. In fact, our designs scale without pain to procedures and language constructs for effects such as exceptions, nondeterminism and interactive input-output.

Programming and reasoning with coinductive types in type theory require taking special care about productivity. Here the type checker of Coq helps us avoid mistakes by ruling out improductivity. But some limitations are imposed by the implementation. For instance, 15 years ago a type-based approach for ensuring productivity of corecursive definitions was developed [8]. This approach is more flexible than the syntactic guardedness approach [7] of Coq, but it has not been implemented. Several coding techniques have been proposed to circumvent the limitations [2, 14]. In our development, we rely on syntactic productivity.

The remainder of the paper is organized as follows. We introduce traces in Section 2. We present the big-step relational semantics in Section 3, the small-step relational semantics in Section 4, and the big-step and small-step functional semantics in Sections 5 and 6, proving them equivalent along the way. We discuss related work in Section 7 to conclude in Section 8.

The language we consider is the While language, defined inductively by the following productions:

$$\mathit{stmt} \quad s \ ::= \ \mathsf{skip} \mid s_0; s_1 \mid x := e \mid \mathsf{if}\ e\ \mathsf{then}\ s_t\ \mathsf{else}\ s_f \mid \mathsf{while}\ e\ \mathsf{do}\ s_t$$

We assume given a supply of variables and a set of (pure) expressions, whose elements are ranged over by metavariables x and e respectively. We assume the set of values to be the integers, non-zero integers counting as truth and zero as falsity. The metavariable v ranges over values. A state, ranged over by σ, maps variables to values. The notation $\sigma[x \mapsto v]$ denotes the update of σ with v at x. We assume given an evaluation function $[\![e]\!]\sigma$, which evaluates e in the state σ. We write $\sigma \models e$ and $\sigma \not\models e$ to denote that e is true, resp. false in σ.

2 Traces

We describe the semantics of statements in terms of traces. A trace is a possibly infinite non-empty sequence of states, the sequence of all states that the run of the statement passes through, including the given initial state. We enforce non-emptiness by having the nil constructor to also take a state as an argument. Formally traces are defined coinductively by the following productions:

$$\mathit{trace} \quad \tau ::= \langle \sigma \rangle \mid \sigma :: \tau$$

$$\frac{}{(x := e, \sigma) \Rightarrow \sigma :: \langle \sigma[x \mapsto [\![e]\!]\sigma] \rangle} \quad \frac{}{(\mathsf{skip}, \sigma) \Rightarrow \langle \sigma \rangle} \quad \frac{(s_0, \sigma) \Rightarrow \tau \quad (s_1, \tau) \overset{*}{\Rightarrow} \tau'}{(s_0; s_1, \sigma) \Rightarrow \tau'}$$

$$\frac{\sigma \models e \quad (s_t, \sigma :: \langle \sigma \rangle) \overset{*}{\Rightarrow} \tau}{(\mathsf{if}\ e\ \mathsf{then}\ s_t\ \mathsf{else}\ s_f, \sigma) \Rightarrow \tau} \quad \frac{\sigma \not\models e \quad (s_f, \sigma :: \langle \sigma \rangle) \overset{*}{\Rightarrow} \tau}{(\mathsf{if}\ e\ \mathsf{then}\ s_t\ \mathsf{else}\ s_f, \sigma) \Rightarrow \tau}$$

$$\frac{\sigma \models e \quad (s_t, \sigma :: \langle \sigma \rangle) \overset{*}{\Rightarrow} \tau \quad (\mathsf{while}\ e\ \mathsf{do}\ s_t, \tau) \overset{*}{\Rightarrow} \tau'}{(\mathsf{while}\ e\ \mathsf{do}\ s_t, \sigma) \Rightarrow \tau'} \quad \frac{\sigma \not\models e}{(\mathsf{while}\ e\ \mathsf{do}\ s_t, \sigma) \Rightarrow \sigma :: \langle \sigma \rangle}$$

$$\frac{(s, \sigma) \Rightarrow \tau}{(s, \langle \sigma \rangle) \overset{*}{\Rightarrow} \tau} \quad \frac{(s, \tau) \overset{*}{\Rightarrow} \tau'}{(s, \sigma :: \tau) \overset{*}{\Rightarrow} \sigma :: \tau'}$$

Fig. 1. Big-step relational semantics

We define bisimilarity of two traces τ, τ', written $\tau \approx \tau'$, by the coinductive interpretation of the following inference rules[1]:

$$\frac{}{\langle \sigma \rangle \approx \langle \sigma \rangle} \quad \frac{\tau \approx \tau'}{\sigma :: \tau \approx \sigma :: \tau'}$$

Bisimilarity is reflexive, symmetric and transitive, i.e., an equivalence. The proofs are straightforward. The reader can find a gentle introduction to coinduction in Coq in [1, Ch. 13].

We want to think of bisimilar traces as equal, corresponding to quotienting the set of traces as defined above by bisimilarity. In our type-theoretic implementation, we do not "compute" the quotient. Instead, we view *trace* with bisimilarity as a setoid, i.e., a set with an equivalence relation. Accordingly, we have to make sure that all functions and predicates we define on *trace* are in fact setoid functions and predicates, i.e., insensitive to bisimilarity.

3 Big-step relational semantics

The main contribution of the paper is the big-step relational semantics, presented in Fig. 1. The semantics is given by two relations $\Rightarrow$ and $\overset{*}{\Rightarrow}$, defined mutually coinductively. The $\Rightarrow$ relation relates a statement-state pair to a trace and is the evaluation relation of our interest: the proposition $(s, \sigma) \Rightarrow \tau$ expresses that running s from an initial state σ results in trace τ. It is defined by case distinction on the statement.

The auxiliary $\overset{*}{\Rightarrow}$ relation relates a statement-trace pair to a trace. Roughly the proposition $(s, \tau) \overset{*}{\Rightarrow} \tau'$ states that running s from the last state of an already accumulated trace τ results in trace τ'. The rules literally define $\overset{*}{\Rightarrow}$ as the coinductive prefix closure of $\Rightarrow$. A more precise description of $(s, \tau) \overset{*}{\Rightarrow} \tau'$ is therefore as follows. If τ is finite, then s is run from the last state of τ and τ'

[1] Following X. Leroy [14], we use double horizontal lines in sets of inference rules that are to be interpreted coinductively and single horizontal lines in inductive definitions.

is obtained from τ by appending the trace produced by s. If τ is infinite (so it does not have a last state), then $(s, \tau) \overset{*}{\Rightarrow} \tau'$ is derivable for any τ' bisimilar to τ, in particular for τ. This design has the desirable consequence that, if a run of the first statement of a sequence diverges, the second statement is not run at all. Indeed, if $(s_0, \sigma) \Rightarrow \tau$ and τ is infinite, then we can derive $(s_1, \tau) \overset{*}{\Rightarrow} \tau$ and further $(s_0; s_1, \sigma) \Rightarrow \tau$. Similarly, if a run of the body of a while-loop diverges, we do not get around to retesting the guard and continuing.

A remarkable feature of the definition of $\overset{*}{\Rightarrow}$ is that it does not hinge on deciding whether the trace is finite or not, which is constructively impossible. A proof of $(s, \tau) \overset{*}{\Rightarrow} \tau'$ simply traverses the already accumulated trace τ: if the last element is hit, the statement is run, otherwise the traversal goes on forever.

Evaluation is a setoid predicate and it is deterministic (up to bisimulation, which is appropriate since we think of bisimilarity as equality):

Lemma 1. *For any σ, s, τ, τ', if $(s, \sigma) \Rightarrow \tau$ and $\tau \approx \tau'$ then $(s, \sigma) \Rightarrow \tau'$.*

Lemma 2. *For any σ, s, τ and τ', if $(s, \sigma) \Rightarrow \tau$ and $(s, \sigma) \Rightarrow \tau'$ then $\tau \approx \tau'$.*

Some design decisions we have made are that skip does not grow a trace, so we have $(\mathsf{skip}, \sigma) \Rightarrow \langle\sigma\rangle$. But an assignment and testing the guard of an if- or while-statement contribute a state, i.e., constitute a small step, e.g., we have $(x := 17, \sigma) \Rightarrow \sigma :: \langle\sigma[x \mapsto 17]\rangle$, $(\mathsf{while\ false\ do\ skip}, \sigma) \Rightarrow \sigma :: \langle\sigma\rangle$ and $(\mathsf{while\ true\ do\ skip}, \sigma) \Rightarrow \sigma :: \sigma :: \sigma :: \ldots$. This is good for several reasons. First, we have that skip is the identity of sequential composition, i.e., the semantics does not distinguish s, $\mathsf{skip}; s$ and $s; \mathsf{skip}$. Second, we get a notion of small steps that fully agrees with the textbook-style small-step semantics given in the next section. The third and most important outcome is that any while-loop always progresses, because testing of the guard is a small step. Another option would be to regard testing of the guard to be instantaneous, but take leaving the loop body, or a backward jump in terms of low-level compiled code, to constitute a small step. But then we would not agree to the textbook small-step semantics.

It is not mandatory to record full states in a trace as we are doing in this paper. It would make perfect sense to record just some observable part of the intermediate states, or to only record that some states were passed through (to track ticks of the clock). Neither is (strong) bisimilarity the only interesting notion of equality of traces. Viable alternatives are various weak versions of bisimilarity (allowing collapsing finite sequences of ticks).

Discussions on alternative designs In the rest of this section we reveal some subtleties in designing coinductive big-step semantics, by looking at several seemingly not so different but problematic alternatives that we reject[2].

Since progress of loops is not required for wellformedness of the definitions of $\Rightarrow$ and $\overset{*}{\Rightarrow}$, one might be tempted to regards guard testing to be instantaneous and modify the rules for the while-loop to take the form

$$\frac{\sigma \models e \quad (s_t, \sigma) \Rightarrow \tau \quad (\mathsf{while}\ e\ \mathsf{do}\ s_t, \tau) \overset{*}{\Rightarrow} \tau'}{(\mathsf{while}\ e\ \mathsf{do}\ s_t, \sigma) \Rightarrow \tau'} \qquad \frac{\sigma \not\models e}{(\mathsf{while}\ e\ \mathsf{do}\ s_t, \sigma) \Rightarrow \langle\sigma\rangle}$$

[2] Our Coq development includes complete definitions of these alternative semantics.

This leads to undesirable outcomes. We can derive $(\mathsf{while\ true\ do\ skip}, \sigma) \Rightarrow \langle\sigma\rangle$, which means that the non-terminating $\mathsf{while\ true\ do\ skip}$ is considered semantically equivalent to the terminal (immediately terminating) skip. Worse, we can also derive $(\mathsf{while\ true\ do\ skip}; x := 17, \sigma) \Rightarrow \sigma :: \langle\sigma[x \mapsto 17]\rangle$, which is even more inadequate: a sequence can continue to run after the non-termination of the first statement. Yet worse, inspecting the rules closer we discover we are also able to derive $(\mathsf{while\ true\ do\ skip}, \sigma) \Rightarrow \tau$ for any τ! Mathematically, giving up insisting on progress in terms of growing the trace has also the consequence that the relational semantics cannot be turned into a functional one, although While should intuitively be total and deterministic. In a functional semantics, evaluation must be a trace-valued function and in a constructive setting such a function must be productive.

Another option, where assignments and test of guards are properly taken to constitute steps, could be to define $\overset{*}{\Rightarrow}$ by case distinction on the statement by rules such as

$$\frac{\tau \models^* e \quad (s_t, \mathit{duplast}\ \tau) \overset{*}{\Rightarrow} \tau' \quad (\mathsf{while}\ e\ \mathsf{do}\ s_t, \tau') \overset{*}{\Rightarrow} \tau''}{(\mathsf{while}\ e\ \mathsf{do}\ s_t, \tau) \overset{*}{\Rightarrow} \tau''} \qquad \frac{\tau \not\models^* e}{(\mathsf{while}\ e\ \mathsf{do}\ s_t, \tau) \overset{*}{\Rightarrow} \mathit{duplast}\ \tau}$$

Here, $\mathit{duplast}\ \tau$, defined corecursively, traverses τ and duplicates its last state, if it is finite. Similarly, $\tau \models^* e$ and $\tau \not\models^* e$ traverse τ and evaluate e in the last state, if it is finite:

$$\frac{\tau \models^* e}{\sigma :: \tau \models^* e} \quad \frac{\sigma \models e}{\langle\sigma\rangle \models^* e} \qquad \frac{\tau \not\models^* e}{\sigma :: \tau \not\models^* e} \quad \frac{\sigma \not\models e}{\langle\sigma\rangle \not\models^* e}$$

(The rules for skip and sequence are very simple and appealing in this design.) The relation $\Rightarrow$ would then be defined uniformly by the rule

$$\frac{(s, \langle\sigma\rangle) \overset{*}{\Rightarrow} \tau}{(s, \sigma) \Rightarrow \tau}$$

It turns out that we can still derive $(\mathsf{while\ true\ do\ skip}, \sigma) \Rightarrow \tau$ for any τ. We can even derive $(\mathsf{while\ true\ do}\ x := x + 1, \sigma) \Rightarrow \tau$ for any τ!

The third alternative (Leroy and Grall use this technique in [14]) is most close to ours. It introduces, instead of our $\overset{*}{\Rightarrow}$ relation, an auxiliary relation *split*, defined coinductively by

$$\frac{}{\mathit{split}\ \langle\sigma\rangle\ \langle\sigma\rangle\ \sigma\ \langle\sigma\rangle} \qquad \frac{}{\mathit{split}\ (\sigma :: \tau)\ \langle\sigma\rangle\ \sigma\ (\sigma :: \tau)} \qquad \frac{\mathit{split}\ \tau\ \tau_0\ \sigma'\ \tau_1}{\mathit{split}\ (\sigma :: \tau)\ (\sigma :: \tau_0)\ \sigma'\ \tau_1}$$

so that $\mathit{split}\ \tau'\ \tau_0\ \sigma'\ \tau_1$ expresses that the trace τ' can be split into a concatenation of traces τ_0 and τ_1 glued together at a mid-state σ'. Then the evaluation relation is defined by replacing the uses of $\overset{*}{\Rightarrow}$ with *split*, e.g., the rule for the sequence statement would be:

$$\frac{\mathit{split}\ \tau'\ \tau_0\ \sigma'\ \tau_1 \quad (s_0, \sigma) \Rightarrow \tau_0 \quad (s_1, \sigma') \Rightarrow \tau_1}{(s_0; s_1, \sigma) \Rightarrow \tau'}$$

$$\frac{}{\mathsf{skip}\not\downarrow} \qquad \frac{s_0\not\downarrow \quad s_1\not\downarrow}{s_0; s_1\not\downarrow}$$

$$\frac{}{(x := e, \sigma) \to (\mathsf{skip}, \sigma[x \mapsto [\![e]\!]\sigma])} \qquad \frac{s_0\not\downarrow \quad (s_1,\sigma) \to (s_1',\sigma')}{(s_0; s_1, \sigma) \to (s_1', \sigma')} \qquad \frac{(s_0,\sigma) \to (s_0',\sigma')}{(s_0; s_1, \sigma) \to (s_0'; s_1, \sigma')}$$

$$\frac{\sigma \models e}{(\mathsf{if}\ e\ \mathsf{then}\ s_t\ \mathsf{else}\ s_f, \sigma) \to (s_t, \sigma)} \qquad \frac{\sigma \not\models e}{(\mathsf{if}\ e\ \mathsf{then}\ s_t\ \mathsf{else}\ s_f, \sigma) \to (s_f, \sigma)}$$

$$\frac{\sigma \models e}{(\mathsf{while}\ e\ \mathsf{do}\ s_t, \sigma) \to (s_t; \mathsf{while}\ e\ \mathsf{do}\ s_t, \sigma)} \qquad \frac{\sigma \not\models e}{(\mathsf{while}\ e\ \mathsf{do}\ s_t, \sigma) \to (\mathsf{skip}, \sigma)}$$

$$\frac{s\not\downarrow}{(s,\sigma) \rightsquigarrow \langle\sigma\rangle} \qquad \frac{(s,\sigma) \to (s',\sigma') \quad (s',\sigma') \rightsquigarrow \tau}{(s,\sigma) \rightsquigarrow \sigma :: \tau}$$

Fig. 2. Small-step relational semantics

This third alternative does not cause any outright anomalies for While. But alarmingly s_1 has to be run from some (underdetermined) state within a run of $s_0; s_1$ even if the run of s_0 does not terminate. In a richer language with abnormal terminations, we get a serious problem: no evaluation is derived for $(\mathsf{while\ true\ do\ skip}); \mathsf{abort}$ although the abort statement should not be reached.

4 Small-step relational semantics

Devising an adequate small-step relational semantics is an easy problem compared to the one of the previous section. We can adapt the textbook inductive small-step semantics, which only accounts for terminating runs. Our semantics, given in Fig. 2, is based on a terminality predicate and one-step reduction relation. The proposition $s\not\downarrow$ states that s is terminal (terminates in no steps), which is possible for a sequence of skips. The proposition $(s,\sigma) \to (s',\sigma')$ states that in state σ the statement s one-step reduces to s' with the next state being σ'. These are exactly the same as one would use for an inductive semantics. The normalization relation is the terminal many-step reduction relation, defined coinductively to allow for the possibility of infinitely many steps. The proposition $(s,\sigma) \rightsquigarrow \tau$ expresses that running s from σ results in the trace τ.

Normalization is a setoid predicate and it is deterministic:

Lemma 3. *For any s, σ, τ, τ', if $(s,\sigma) \rightsquigarrow \tau$ and $\tau \approx \tau'$ then $(s,\sigma) \rightsquigarrow \tau'$.*

Lemma 4. *For any s, σ, τ, τ', if $(s,\sigma) \rightsquigarrow \tau$ and $(s,\sigma) \rightsquigarrow \tau'$ then $\tau \approx \tau'$.*

Equivalence to big-step relational semantics Of course we expect the big-step and small-step semantics to be equivalent. We will show this in two ways: the first approach, presented in this section, directly proves the equivalence; the second approach, given in Section 6, proves the equivalence by going through

functional semantics. The first approach is stronger in that it does not rely on the determinism of the semantics, thus prepares a better avenue for generalization to a language with non-determinism. (Our functional semantics deals with single-valued functions and thus the second approach relies on the determinism.)

The following lemma connects the big-step semantics with the terminality predicate and one-step reduction relation and is proved by induction.

Lemma 5. *For any s, σ and τ, if $(s, \sigma) \Rightarrow \tau$ then either $s \downarrow$ and $\tau = \langle\sigma\rangle$, or else there are s', σ', τ' such that $(s, \sigma) \to (s', \sigma')$ and $\tau \approx \sigma :: \tau'$ and $(s', \sigma') \Rightarrow \tau'$.*

Then correctness of the big-step semantics relative to the small-step semantics follows by coinduction:

Proposition 1. *For any s, σ and τ, if $(s, \sigma) \Rightarrow \tau$ then $(s, \sigma) \rightsquigarrow \tau$.*

The opposite direction, that the small-step semantics is correct relative to the big-step semantics, is more interesting. The proof proceeds by coinduction. At the crux is the case of the sequence statement: we are given a normalization $(s_0; s_1, \sigma) \rightsquigarrow \tau$ and the coinduction hypotheses for s_0 (resp. s_1) that enable us to deduce $(s_0, \sigma') \Rightarrow \tau'$ (resp. $(s_1, \sigma') \Rightarrow \tau'$) from $(s_0, \sigma') \rightsquigarrow \tau'$ (resp. $(s_1, \sigma') \rightsquigarrow \tau'$) for any σ', τ'. Naively, we have what we need to close the case. The assumption $(s_0; s_1, \sigma) \rightsquigarrow \tau$ ensures that τ can be split into two parts τ_0 and τ_1 such that τ_0 corresponds to running s_0 and τ_1 to running s_1. If τ_0 is finite, we can traverse τ_0 until we hit its last state, to then invoke the coinduction hypothesis on s_1. If τ_0 is infinite, we can deduce $\tau \approx \tau_0$ and $(s_1, \tau_0) \overset{*}{\Rightarrow} \tau_0$ by coinduction.

The actual proof is more involved. First we have to explicitly construct τ_0 and τ_1. This is possible by examining the proof of $(s_0; s_1, \sigma) \rightsquigarrow \tau$. Our proof defines an auxiliary function *midp* $(s_0\ s_1 : \mathit{stmt})\ (\sigma : \mathit{state})\ (\tau : \mathit{trace})\ (h : (s_0; s_1, \sigma) \rightsquigarrow \tau) : \mathit{trace}$ by corecursion as follows. We look at the last inference in the proof h of $(s_0; s_1, \sigma) \rightsquigarrow \tau$. If $s_0; s_1$ is terminal, we return $\langle\sigma\rangle$. Otherwise we have a proof h_0 of $(s_0; s_1, \sigma) \to (s', \sigma')$ and a proof h' of $(s', \sigma') \rightsquigarrow \tau'$ for some σ', τ' such that $\tau = \sigma :: \tau'$. We look at the last inference in h_0. If s_0 is terminal, we also return $\langle\sigma\rangle$. Else it must be the case that $(s_0, \sigma) \to (s_0', \sigma')$ for some s_0' such that $s' = s_0'; s_1$ and we return $\sigma :: \mathit{midp}\ s_0'\ s_1\ \sigma'\ \tau'\ h'$. The corecursive call is guarded by consing σ. The following lemma is proved by coinduction.

Lemma 6. *For any s_0, s_1, σ, τ, $h : (s_0; s_1, \sigma) \rightsquigarrow \tau$, $(s_0, \sigma) \rightsquigarrow \mathit{midp}\ s_0\ s_1\ \sigma\ \tau\ h$.*

Second, we cannot decide whether τ_0 is finite as this would amount to deciding whether running s_0 from σ terminates. Our big-step semantics was carefully crafted to avoid stumbling upon this problem, by introduction of the coinductive prefix closure $\overset{*}{\Rightarrow}$ of $\Rightarrow$ to uniformly handle the cases of both the finite and infinite already accumulated trace. We need a small-step counterpart to it:

$$\frac{(s, \sigma) \rightsquigarrow \tau}{(s, \langle\sigma\rangle) \overset{*}{\rightsquigarrow} \tau} \qquad \frac{(s, \tau) \overset{*}{\rightsquigarrow} \tau'}{(s, \sigma :: \tau) \overset{*}{\rightsquigarrow} \sigma :: \tau'}$$

The proposition $(s, \tau) \overset{*}{\rightsquigarrow} \tau'$ states that running s from the last state of an already accumulated trace τ (if it has one) results in the total trace τ'. The following lemma is proved by coinduction.

$$\frac{s \not\downarrow}{(s,\sigma) \rightsquigarrow^{\mathrm{ind}} \sigma} \qquad \frac{(s,\sigma) \to (s',\sigma') \quad (s',\sigma') \rightsquigarrow^{\mathrm{ind}} \sigma''}{(s,\sigma) \rightsquigarrow^{\mathrm{ind}} \sigma''}$$

Fig. 3. Inductive small-step relational semantics

Lemma 7. *For any s_0, s_1, σ, τ, $h : (s_0; s_1, \sigma) \rightsquigarrow \tau$, $(s_1, \mathit{midp}\ s_0\ s_1\ \sigma\ \tau\ h) \overset{*}{\rightsquigarrow} \tau$.*

Only now we can finally prove that the small-step relational semantics is correct relative to the big-step relational semantics.

Proposition 2. *For any s, σ, τ, τ', the following two conditions hold:*

- *if $(s,\sigma) \rightsquigarrow \tau$ then $(s,\sigma) \Rightarrow \tau$,*
- *if $(s,\tau) \overset{*}{\rightsquigarrow} \tau'$ then $(s,\tau) \overset{*}{\Rightarrow} \tau'$.*

Proof. Both conditions are proved at once by mutual coinduction. We only show the first condition in the case of the sequence statement, to demonstrate how the relation $\overset{*}{\rightsquigarrow}$ helps us avoid having to decide finiteness. Suppose we have $h : (s_0; s_1, \sigma) \rightsquigarrow \tau$. By Lemmata 6 and 7, we have $(s_0, \sigma) \rightsquigarrow \mathit{midp}\ s_0\ s_1\ \sigma\ \tau\ h$ and $(s_1, \mathit{midp}\ s_0\ s_1\ \sigma\ \tau\ h) \overset{*}{\rightsquigarrow} \tau$. By invoking the coinduction hypothesis on them, we obtain $(s_0, \sigma) \Rightarrow \mathit{midp}\ s_0\ s_1\ \sigma\ \tau\ h$ and $(s_1, \mathit{midp}\ s_0\ s_1\ \sigma\ \tau\ h) \overset{*}{\Rightarrow} \tau$, from which we deduce $(s_0; s_1, \sigma) \Rightarrow \tau$.

Differences between Coq's *Prop* and *Set* force normalization to be *Set*-valued rather than *Prop*-valued, since our definition of the *trace*-valued *midp* function relies on case distinction on the proof of the given normalization proposition. Case distinction on a proof of a *Prop*-proposition is not available for constructing an element of a *Set*-set. This in turn requires the evaluation relation to also be *Set*-valued, to be comparable to normalization. A further complication is that, for technical reasons, the proofs of Lemmata 6 and 7 must rely on John Major equality [15] and the principle that two JM-equal elements of the same type are equal. Given that Coq's support for programming with (co)inductive families (in ML-style, as opposed to proving in the tactic language) is also weak (so *midp* was easily manufactured in the tactic language, but we failed to construct it in ML-style), one might wish to prove the equivalence of the big-step and small-step semantics in some altogether different way. In the subsequent sections we study functional semantics. These offer us a less direct route that is less painful in the aspects we have just described.

Adequacy relative to inductive small-step relational semantics The textbook inductive small-step relational semantics defines normalization as the inductive terminal many-step reduction. The definition is given in Fig. 3. This normalization relation associates a state-statement pair with a state (the terminal state) rather than a trace, although a trace-based version (for an inductive concept of traces, i.e., finite sequences of states) would be obtained by a straightforward modification. For completeness of our development, we prove that the inductive and coinductive semantics agree on terminating runs.

We introduce a last-state predicate on traces inductively by the rules

$$\frac{}{\langle\sigma\rangle \downarrow \sigma} \qquad \frac{\tau \downarrow \sigma'}{\sigma :: \tau \downarrow \sigma'}$$

Proposition 3 states that the inductive semantics is correct relative to the coinductive semantics. Proposition 4 states that the coinductive semantics is correct relative to the inductive semantics for terminating runs. Both propositions are proved by induction.

Proposition 3. *For any s and σ, if $(s,\sigma) \rightsquigarrow^{\text{ind}} \sigma'$ then there is τ such that $(s,\sigma) \rightsquigarrow \tau$ and $\tau \downarrow \sigma'$.*

Proposition 4. *For any s, τ, σ, σ', if $(s,\sigma) \rightsquigarrow \tau$ and $\tau \downarrow \sigma'$ then $(s,\sigma) \rightsquigarrow^{\text{ind}} \sigma'$.*

The connection between our coinductive big-step semantics and the inductive big-step semantics can now be concluded from the well-known equivalence between the inductive big-step and small-step semantics. Y. Bertot has formalized the proof of this equivalence in Coq [3].

We conclude this section by citing an observation by V. Capretta [4]. The infiniteness predicate on traces is defined coinductively by the rule

$$\frac{\tau^\uparrow}{(\sigma :: \tau)^\uparrow}$$

We can prove in Coq the proposition $\forall\tau, (\neg\exists\sigma, \tau \downarrow \sigma) \to \tau^\uparrow$. However the proposition $\forall\tau, (\exists\sigma, \tau \downarrow \sigma) \vee \tau^\uparrow$can only be proved from $\forall\tau, (\exists\sigma, \tau \downarrow \sigma) \vee \neg(\exists\sigma, \tau \downarrow \sigma)$. Constructively, this instance of the classical law of excluded middle states decidability of finiteness. For this reason, we reject what could be called sum-type semantics. For instance, a relational semantics could relate a statement-state pair to either a state for a terminating run or a special token ∞ for a non-terminating run, i.e., an element from the sum type $state+1$, where 1 is the one-element type. Or, it could be given as the disjunction of an inductive trace-based semantics, describing terminating runs, and a coinductive trace-based semantics, describing non-terminating runs, an approach studied in [14].

5 Big-step functional semantics

We now proceed to functional versions of our semantics. The standard state-based approach to While does not allow for a (constructive) functional semantics, as this would require deciding the halting problem. Working with traces has the benefit that we do not have to decide: any statement and initial state uniquely determine some trace and we do not have to know whether this trace is finite or infinite. The semantics is given in Fig. 4. The evaluation function $eval : stmt \to state \to trace$ is defined by recursion on the statement. In the cases for sequence and while it calls auxiliary functions *sequence* and *loop*.

We first look at *loop*. It is defined together with a further auxiliary function *loopseq* by mutual corecursion. *loop* takes three arguments: k for evaluating the

```
Fixpoint eval (s : stmt) (σ : state) {struct s} : trace :=
  match s with
  | skip ⇒ ⟨σ⟩
  | x := e ⇒ σ :: ⟨σ[x ↦ [[e]]σ]⟩
  | s0; s1 ⇒ sequence (eval s1) (eval s0 σ)
  | if e then st else sf ⇒ σ :: if σ ⊨ e then eval st σ else eval sf σ
  | while e do st ⇒ σ :: loop (eval st) [[e]] σ
  end

CoFixpoint sequence (k : state → trace) (τ : trace) : trace :=
  match τ with
  | ⟨σ⟩ ⇒ k σ
  | σ :: τ' ⇒ σ :: sequence k τ'
  end

CoFixpoint loop (k : state → trace) (p : state → bool) (σ : state) : trace :=
  if p σ then
    match k σ with
    | ⟨σ'⟩ ⇒ σ' :: loop k p σ'
    | σ' :: τ ⇒ σ' :: loopseq k p τ
    end
  else ⟨σ⟩
with loopseq (k : state →  trace) (p : state → bool) (τ : trace) : trace :=
  match τ with
  | ⟨σ⟩ ⇒ σ :: loop k p σ
  | σ :: τ' ⇒ σ :: loopseq k p τ'
  end
```

Fig. 4. Big-step functional semantics

loop body from a state; p for testing the boolean guard on a state; and a state σ, which is the initial state. *loopseq* takes a trace τ, the initial trace, instead of a state, as the third argument. The two functions work as follows. *loop* takes care of repeating of the loop body, once the guard of a while loop has been evaluated. It analyzes the result and, if the guard is false, then the run of the loop terminates. If it is true, then the loop body is evaluated by calling k. *loop* then constructs the trace of the loop body by examining the result of k. If the loop body does not augment the trace, which can only happen, if the loop body is a sequence of skips, a new round of repeating the loop body is started by a corecursive call to *loop*. The corecursive call is guarded by first augmenting the trace, which corresponds to the new evaluation of the boolean guard. If the loop body augments the trace, the new round is reached by reconstruction of the trace of the current repetition with *loopseq*. On the exhaustion of this trace, *loopseq* corecursively calls *loop*, again appropriately guarded. Our choice of augmenting traces at boolean guards facilitates implementing *loop* in Coq: we exploit it to satisfy Coq's syntactic guardedness condition.

sequence, defined by simple corecursion, is similar to *loopseq*, but does not involve repetition. It takes two arguments: k for running a statement (the second

statement of a sequence) from a state and τ the already accumulated trace (resulting from running the first statement of the sequence). After reconstructing τ, *sequence* calls k on the last state of τ.

Proposition 5 proves the big-step functional semantics correct relative to the big-step relational semantics. The proof proceeds by induction on the statement and performs coinductive reasoning in the cases for sequence and while. Moreover, the way induction and coinduction hypotheses are invoked mimics the way *eval* makes recursive calls and *sequence* and *loop* make corecursive calls.

Proposition 5. *For any s, σ, $(s, \sigma) \Rightarrow eval\ s\ \sigma$.*

Proof. By induction on s. We show the interesting cases of sequence and while.

- $s = s_0; s_1$: We are given as the induction hypotheses that, for any σ, $(s_0, \sigma) \Rightarrow eval\ s_0\ \sigma$ and $(s_1, \sigma) \Rightarrow eval\ s_1\ \sigma$. We must prove $(s_0; s_1, \sigma) \Rightarrow eval\ (s_0; s_1)\ \sigma$. We do so by proving the following condition by coinduction: for any τ, $(s_1, \tau) \overset{*}{\Rightarrow} sequence\ (eval\ s_1)\ \tau$. The proof of the condition proceeds by case distinction on τ and invokes the induction hypothesis on s_1 for the case where τ is a singleton. Then we close the case by combining $(s_0, \sigma) \Rightarrow (eval\ s_0\ \sigma)$, which is obtained from the induction hypothesis on s_0, and $(s_1, (eval\ s_0\ \sigma)) \overset{*}{\Rightarrow} sequence\ (eval\ s_1)\ (eval\ s_0\ \sigma)$, which is obtained from the condition proved.
- $s = \mathsf{while}\ e\ \mathsf{do}\ s_t$: We are given as the induction hypothesis that, for any σ, $(s_t, \sigma) \Rightarrow eval\ s_t\ \sigma$. We must prove $(\mathsf{while}\ e\ \mathsf{do}\ s_t, \sigma) \Rightarrow eval\ (\mathsf{while}\ e\ \mathsf{do}\ s_t)\ \sigma$. We do so by proving the following two conditions simultaneously by mutual coinduction:
  - for any σ, $(\mathsf{while}\ e\ \mathsf{do}\ s_t, \sigma) \Rightarrow \sigma :: loop\ (eval\ s_t)\ [\![e]\!]\ \sigma$,
  - for any τ, $(\mathsf{while}\ e\ \mathsf{do}\ s_t, \tau) \overset{*}{\Rightarrow} loopseq\ (eval\ s_t)\ [\![e]\!]\ \tau$.

  The proof of the first condition invokes the induction hypothesis on s_t. The case follows immediately from the first condition.

As an obvious corollary of Proposition 5, the relational semantics is total:

Corollary 1. *For any s, σ, there exists τ such that $(s, \sigma) \Rightarrow \tau$.*

This corollary is valuable on its own, and in fact it is one motivation for defining the functional big-step semantics. Since the conclusion is not a coinductive predicate, we cannot prove the corollary directly by coinduction.

Correctness of the big-step functional semantics relative to the big-step relational semantics is easy. In the light of Lemma 2 (determinism of the big-step relational semantics), it is an immediate consequence of Proposition 5.

Proposition 6. *For any s, σ, τ, if $(s, \sigma) \Rightarrow \tau$, then $\tau \approx eval\ s\ \sigma$.*

The fact that the corecursive functions *sequence* and *loop* must produce traces in a guarded way in the functional semantics lends further support to our definition of the big-step relational semantics. Mere wellformedness of a coinductive definition of the evaluation relation does not suffice, the rules must be tight enough to properly define the trace of a run.

```
Fixpoint red (s : stmt) (σ : state) {struct s} : option (stmt * state) :=
  match s with
  | skip ⇒ None
  | x := e ⇒ Some (skip, σ[x ↦ [[e]]σ])
  | s0; s1 ⇒
    match (red s0 σ) with
    | Some (s0', σ') ⇒ Some (s0'; s1, σ')
    | None ⇒ red s1 σ
    end
  | if e then st else sf ⇒ if [[e]]σ then Some (st, σ) else Some (sf, σ)
  | while e do st ⇒ if [[e]]σ then Some (st; while e do st, σ) else Some (skip, σ)
  end

CoFixpoint norm (s : stmt) (σ : state) : trace :=
  match red s σ with
    | None ⇒ ⟨σ⟩
    | Some (s', σ') ⇒ σ :: norm s' σ'
  end
```

Fig. 5. Small-step functional semantics

6 Small-step functional semantics

Our small-step functional semantics, defined in Fig. 5, is quite similar to the small-step relational semantics, except that it uses a function to perform one-step reductions. The option-returning one-step reduction function *red* is a functional equivalent to the jointly total and deterministic terminality predicate and one-step reduction relation of Fig. 2. It returns *None*, if the given statement is terminal; otherwise, it one-step reduces the given statement from the given state and returns the resulting statement-state pair. The normalization function *norm* calls *red* repeatedly; it is defined by corecursion (guardedness is achieved by consing the current state to the corecursive call on the next state).

The small-step functional semantics is correct relative to the small-step relational semantics:

Proposition 7. *For all s, σ, $(s, \sigma) \rightsquigarrow^* \mathit{norm}\ s\ \sigma$.*

That the small-step relational semantics is correct relative to the small-step functional semantics is an immediate consequence of Proposition 7 and Lemma 4 (determinism of small-step relational semantics).

Proposition 8. *For all s, σ, τ, if $(s, \sigma) \rightsquigarrow^* \tau$, then $\tau \approx \mathit{norm}\ s\ \sigma$.*

To verify the equivalence of the small-step functional semantics to the big-step functional semantics, we first prove two auxiliary lemmas. Lemma 8 relates *norm* and *sequence*. Lemma 9 relates *norm* and *loop* together with *loopseq*. The former is proved by coinduction, the latter is by mutual coinduction.

Lemma 8. *For any s_0, s_1, σ, $\mathit{norm}\ (s_0; s_1)\ \sigma \approx \mathit{sequence}\ (\mathit{norm}\ s_1)\ (\mathit{norm}\ s_0\ \sigma)$.*

Lemma 9. *For any e, s_t and σ, the following two conditions hold:*

- *norm* (while e do s_t) $\sigma \approx \sigma ::$ *loop* (*norm* s_t) $[\![e]\!]$ σ,
- *for any* s, *norm* (s; while e do s_t) $\sigma \approx$ *loopseq* (*norm* s_t) $[\![e]\!]$ (*norm* s σ).

Equipped with these lemmata we are in the position to show the big-step and small-step functional semantics to agree up to bisimilarity.

Proposition 9. *For any s and σ, eval s $\sigma \approx$ norm s σ.*

Proof. By induction on s. We outline the proof for the main cases.

- $s = s_0; s_1$: We prove by coinduction the following condition: for any τ and τ' such that $\tau \approx \tau'$, *sequence* (*eval* s_1) $\tau \approx$ *sequence* (*norm* s_1) τ'. The proof of the condition uses the induction hypothesis on s_1. Then the condition, Lemma 8 and the induction hypothesis on s_0 together close the case.

- $s =$ while e do s_t: We prove the following two conditions simultaneously by mutual coinduction using the induction hypothesis:
  - for any σ, *loop* (*eval* s_t) $[\![e]\!]$ $\sigma \approx$ *loop* (*norm* s_t) $[\![e]\!]$ σ,
  - for any τ, τ' such that $\tau \approx \tau'$,
    loopseq (*eval* s_t) $[\![e]\!]$ $\tau \approx$ *loopseq* (*norm* s_t) $[\![e]\!]$ τ'.

  Then the first condition and Lemma 9 together close the case.

We can now prove that the small-step relational semantics correct relative to the big-step relational semantics without having to rely on dependent pattern-matching or JM equality by going through the functional semantics:

Proposition 10. *For any s, σ and τ, if $(s, \sigma) \rightsquigarrow \tau$ then $(s, \sigma) \Rightarrow \tau$.*

Proof. By Prop. 7 and Lemma 4, $\tau \approx$ *norm* s σ. By Prop. 9 and the transitivity of the bisimilarity relation, $\tau \approx$ *eval* s σ. By Prop. 5 and Lemma 1, $(s, \sigma) \Rightarrow \tau$.

7 Related work

X. Leroy and H. Grall [14] study two approaches to big-step relational semantics for lambda-calculus, accounting for both terminating and non-terminating evaluations, fully formalized in Coq. In both approaches, evaluation relates lambda-terms to normal forms or to reduction sequences.

The first approach, inspired by Cousot and Cousot [5], uses two evaluation relations, an inductive one for terminating evaluations and a coinductive one for non-terminating evaluations. The proof of equivalence to the small-step semantics requires the use of an instance of the excluded middle, constructively amounting to deciding halting. In essence, this means adopting a sum-type solution. This has deep implications even for the big-step semantics alone: the determinism of the evaluation relation, for example, can only be shown by going through the small-step semantics.

Leroy used this approach in his work on a certified C compiler [13]. To the best of our knowledge, this was the first practical application of mechanized coinductive semantics. C is one of the most used languages for developing programs that are not supposed to terminate, such as operating systems. Hence it is important that a certified compiler for C preserves the semantics of non-terminating programs. The work on the Compcert compiler is a strong witness of the importance and practicality of mechanized coinductive semantics.

In our approach to While, we have a single evaluation relation for both terminating and non-terminating runs. The big-step semantics is equivalent to the small-step semantics constructively. Furthermore, the big-step semantics is constructively deterministic and the proof of this is without an indirection through the small-step semantics.

Leroy and Grall [14] also study a different big-step semantics where both terminating and non-terminating runs are described by a single coinductively defined evaluation relation ("coevaluation") relating lambda-terms to normal forms or reduction sequences. This semantics does not agree with the small-step semantics, since it assigns a result even to an infinite reduction sequence and continues reducing a function even after the argument diverges.

Coinductive big-step relational semantics for While similar in some aspects to Leroy and Grall's work on lambda-calculus appear in the works of Glesner [9] and Nestra [16, 17]. Regardless of whether evaluation relates statement-state pairs to possibly infinite traces, possibly non-wellfounded trees of states ("fractions") or transfinite traces, these approaches have it in common that the result of a non-terminating run can be non-deterministic even for While-programs, which should be deterministic. For one technical reason or another, it becomes possible in all these semantics that after an infinite number of small steps a run reaches an underdetermined limit state and continues from there. In the case of Nestra [16, 17], this seems intended: he devised his non-standard "fractional" and transfinite semantics to justify a program slicing transformation that is unsound under the standard semantics. Elsewhere, the outcome appears accidental and undesired.

In our approach, we take the result of a program run to be given precisely by what can be finitely observed: we record the state of the program at every finite time instant. We never run ahead of the clock by jumping over some intermediate states (in particular, we never run ahead of the clock infinitely much) and we reject transfinite time. As a result of this design decision, the big-step semantics agrees precisely with the small-step semantics and does so even constructively.

Coinductive functional semantics similar to ours have appeared in the works of J. Rutten and V. Capretta. A difference is that instead of trace-based semantics they looked at delayed state based semantics, i.e., semantics that, for a given statement-state pair, return a possibly infinitely delayed state. Delayed states, or Burroni conaturals over states, are like conatural numbers (possibly infinite natural numbers), except that instead of the number zero their deconstruction terminates (if it does) with a state.

J. Rutten [18] gave a delayed state based coinductive small-step functional semantics for While in coalgebraic terms. The one-step reduction function is a

coalgebra on statement-state pairs. The final coalgebra is given by the analysis of a delayed state into a readily available state or a unit delay of a further delayed state (the predecessor function on Burroni conaturals over states). The small-step semantics, which sends a statement-state pair to a delayed state, is given by the unique map from a coalgebra to the final coalgebra. He also discussed weak bisimilarity of delayed states. This identifies all finite delays. As he worked in classical set theory, the quotient of the set of delayed states by weak bisimilarity is isomorphic to the sum of the set of states and a one-element set. However the coalgebraic approach is not confined to the category of sets and in constructive settings the theory of weak bisimilarity is richer.

V. Capretta [4] carried out a similar project in a constructive, type-theoretic setting, focusing on combinators essential for big-step functional semantics (our *sequence* and *loop*). Central to him was the realization that the delay type constructor is a monad, more specifically a completely iterative monad (with $\langle\rangle$ the unit, *sequence* the Kleisli extension operation and *loop* the iteration operation). Similarly to Leroy and Grall, he formalized his development in Coq.

Our work is very much inspired by the designs of Rutten and Capretta. Here, however, we have replaced the delay monad by the trace monad, which is also completely iterative. Moreover, we have also considered relational semantics.

A general categorical account of small-step trace semantics has been given by I. Hasuo et al. [12].

8 Conclusion

We have devised four trace-based coinductive semantics for While in different styles of operational semantics. We were pleased to find that simple semantics covering both terminating and non-terminating program runs are possible even in the big-step relational and functional styles. The metatheory of our coinductive semantics is remarkably analogous to that of the textbook inductive semantics and on finite runs they agree. Remarkably, everything can be arranged so that in a constructive setting we never have to decide whether a trace is finite or infinite.

Acknowledgments K. Nakata thanks X. Leroy for interesting discussions on the coinductive semantics for the Compcert compiler during her post-doctoral stay in the Gallium team at INRIA Rocquencourt. The Ssreflect library was instrumental in producing the formal development. She is also thankful for helpful advice she received via the Coq and Ssreflect mailing-lists. T. Uustalu acknowledges the many inspiring discussions on the delay monad with T. Altenkirch and V. Capretta. Both authors were supported by the Estonian Science Foundation grant no. 6940 and the EU FP6 IST integrated project no. 15905 MOBIUS.

References

1. Bertot, Y., Castéran, P.: Coq'Art: Interactive Theorem Proving and Program Development. Springer (2004)

2. Bertot, Y.: Filters on coinductive streams, an application to Eratosthenes' sieve. In Urzyczyn, P. (Ed.): Proc. of 7th Int. Conf on Typed Lambda Calculi and Applications, TLCA 2005 (Nara, Apr. 2005), Lect. Notes in Comput. Sci., Vol. 3461. Springer (2005) 102–115
3. Bertot, Y.: A survey of programming language semantics styles. Coq development. (2007) `http://www-sop.inria.fr/marelle/Yves.Bertot/proofs.html`
4. Capretta, V.: General recursion via coinductive types. Logical Methods in Computer Science **1**(2) (2005) 1–18
5. Cousot, P., Cousot, R.: Inductive definitions, semantics and abstract interpretation. In Conf. Record of 19th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, POPL '92 (Albuquerque, NM, Jan. 1992). ACM Press (1992) 83–94
6. Cousot, P., Cousot, R.: Bi-inductive structural semantics. Inform. and Comput. **207**(2) (2009) 258–283
7. Giménez, E.: Codifying guarded definitions with recursive schemes. In Dybjer, P., Nordström, Smith, J. M. (Eds.): Selected Papers from Int. Wksh. on Types for Proofs and Programs, TYPES '94 (Båstad, June 1994), Lect. Notes in Comput. Sci., Vol. 996. Springer (1995) 39–59
8. Giménez, E.: Structural recursive definitions in type theory. In Larsen, K. G., Skyum, S., Wynskel, G. (Eds.): Proc. of 25th Int. Coll. on Automata, Languages and Programming, ICALP '98 (Aalborg, July 1998), Lect. Notes in Comput. Sci., Vol. 1443. Springer (1998) 397–408
9. Glesner, S.: A proof calculus for natural semantics based on greatest fixed point semantics. In Knoop, J., Necula, G. C., Zimmermann, W. (Eds.): Proc. of 3rd Int. Wksh. on Compiler Optimization Meets Compiler Verification, COCV 2004 (Barcelona, Apr. 2004), Electron. Notes in Theor. Comput. Sci., Vol. 132(1). Elsevier (2005) 73–93
10. Glesner, S., Leitner, J., Blech, J.O.: Coinductive verification of program optimizations using similarity relations. In Knoop, J., Necula, G. C., Zimmermann, W. (Eds.): Proc. of 5th Int. Wksh. on Compiler Optimization Meets Compiler Verification, COCV '06 (Vienna, Apr. 2006). Electron. Notes in Theor. Comput. Sci., Vol. 176(3). Elsevier (2007) 61–77
11. Gonthier, G., Mahboubi, A.: A small scale reflection extension for the Coq system. Technical Report RR-6455, INRIA (2008)
12. Hasuo, I., Jacobs, B., Sokolova, A. (2007): Generic trace semantics via coinduction. Logical Methods in Comput. Sci. **3**(4), article 11.
13. Leroy, X.: The Compcert verified compiler. Commented Coq development (2008) `http://compcert.inria.fr/doc/`
14. Leroy, X., Grall, H.: Coinductive big-step operational semantics. Inform. and Comput. **207**(2) (2009) 285–305
15. McBride, C.: Elimination with a motive. In Callaghan P. et al. (Eds.): Selected Papers from Int. Wksh. on Types for Proofs and Programs, TYPES 2000 (Durham, Dec. 2000), Lect. Notes in Comput. Sci., Vol. 2277. Springer (2002) 197–216
16. Nestra, H.: Fractional semantics. In Johnson, M., Vene, V. (Eds.): Proc. of 11th Int. Conf. on Algebraic Methodology and Software Technology, AMAST 2006 (Kuressaare, July 2006), Lect. Notes in Comput. Sci., Vol. 4019. Springer (2006) 278–292
17. Nestra, H.: Transfinite semantics in the form of greatest fixpoint. J. of Logic and Algebr. Program (to appear)
18. Rutten, J.: A note on coinduction and weak bisimilarity for While programs. Theor. Inform. and Appl. **33**(4–5) (1999) 393–400