Coinductive big-step operational semantics

Xavier Leroy, Hervé Grall

Coinductive big-step operational semantics

Xavier Leroy [a,*] Hervé Grall [b]

[a] *INRIA Paris-Rocquencourt*
Domaine de Voluceau, B.P. 105, 78153 Le Chesnay, France

[b] *École des Mines de Nantes*
La Chantrerie, 4, rue Alfred Kastler, B.P. 20722, 44307 Nantes, France

Abstract

Using a call-by-value functional language as an example, this article illustrates the use of coinductive definitions and proofs in big-step operational semantics, enabling it to describe diverging evaluations in addition to terminating evaluations. We formalize the connections between the coinductive big-step semantics and the standard small-step semantics, proving that both semantics are equivalent. We then study the use of coinductive big-step semantics in proofs of type soundness and proofs of semantic preservation for compilers. A methodological originality of this paper is that all results have been proved using the Coq proof assistant. We explain the proof-theoretic presentation of coinductive definitions and proofs offered by Coq, and show that it facilitates the discovery and the presentation of the results.

Key words: Coinduction, Operational semantics, Big-step semantics, Natural semantics, Small-step semantics, Reduction semantics, Type soundness, Compiler correctness, Mechanized proofs, The Coq proof assistant

1 Introduction

There exist two widely-used styles of operational semantics: *big-step semantics*, popularized by Kahn [1] under the name *natural semantics*, relates programs to the final results of their evaluations; *small-step semantics*, popularized by Plotkin [2,3] under the name *structural operational semantics*, repeatedly applies a one-step reduction relation to form reduction sequences. Small-step semantics is more expressive since it can describe the evaluation

* Corresponding author
Email address: Xavier.Leroy@inria.fr (Xavier Leroy).

of both terminating and non-terminating programs, as finite or infinite reduction sequences, respectively. In contrast, big-step semantics describes only the evaluation of terminating programs, and fails to distinguish between non-terminating programs and programs that "go wrong". For this reason, small-step semantics is generally preferred, in particular for proving the soundness of type systems.

However, big-step semantics is more convenient than small-step semantics for some applications. One that is dear to our heart is proving the correctness (preservation of program behaviours) of program transformations, especially compilation of a high-level programming language down to a lower-level language. The first author's experience and that of others [4,5,6] is that fairly complex, optimizing compilation passes can be proved correct (for terminating source programs) relatively easily using big-step semantics and inductions on the structure of big-step evaluation derivations. In contrast, compiler correctness proofs using small-step semantics can address both terminating and diverging source programs, but are more difficult even for simple, non-optimizing compilation schemes [7].

In this article, we illustrate how coinductive definitions and proofs enable big-step semantics to describe both terminating and diverging evaluations. The target of our study is a simple call-by-value functional language. We study two approaches: the first, initially proposed by Cousot and Cousot [8], complements the normal inductive big-step evaluation rules for terminating evaluations with coinductive big-step rules describing diverging evaluations; the second simply interprets coinductively the normal big-step evaluation rules, thus enabling them to describe both terminating and non-terminating evaluations. These semantics are defined in sections 3 and 7, respectively.

The main technical results of this article are of two kinds. First, we prove that the coinductive big-step definition of divergence is equivalent to the more familiar definitions using either small-step semantics (section 4) or a simple form of denotational semantics (section 5). We also extend these equivalence results to trace semantics (section 6). Then, we study two applications of the big-step definition of divergence: a novel approach to stating and proving the soundness of type systems (section 8), and proofs of semantic preservation for compilation down to an abstract machine (section 9).

An originality of this article is that all results were not only proved using a proof assistant (the Coq system), but even developed in interaction with this tool, and only then transcribed to standard mathematical notations. The Coq proof assistant [9,10] provides built-in support for coinductive definitions and proofs by coinduction. This support follows a proof-theoretic approach to induction and coinduction that we present in section 2 and relate with the standard approach using fixed points. The proof-theoretic approach leads to

proofs by coinduction that are simpler than the standard arguments based on F-consistent relations [11,12]. Our use of Coq has therefore been doubly beneficial: it facilitated the discovery and presentation of the results in this article, while at the same time generating strong confidence in them.

2 Induction and coinduction: A proof-theoretic approach

Following the classical presentation of Aczel [13], an inference system over a set $\mathcal{U}$ of judgments is a set of inference rules. An *inference rule* is an ordered pair $(\mathcal{A}, c)$, where $c \in \mathcal{U}$ is the *conclusion* of the rule and $\mathcal{A} \subseteq \mathcal{U}$ is the set of its *premises* or *antecedents*. A rule is usually written as follows:

$$\frac{\mathcal{A}}{c}$$

The intuitive interpretation of this rule is that the judgment c can be inferred from the set of judgments $\mathcal{A}$.

2.1 Fixed-point approach

One way to give meaning to an inference system is to consider the fixed points of the associated inference operator. If Φ is an inference system over $\mathcal{U}$, we define the operator $F_\Phi : \wp(\mathcal{U}) \to \wp(\mathcal{U})$ as

$$F_\Phi(S) = \{c \in \mathcal{U} \mid \exists \mathcal{A} \subseteq S,\ (\mathcal{A}, c) \in \Phi\}.$$

In other terms, $F_\Phi(S)$ is the set of judgments that can be inferred in one step from the judgments in S by using the inference rules.

A set S is said to be *closed* if $F_\Phi(S) \subseteq S$, and *consistent* if $S \subseteq F_\Phi(S)$. A closed set S is such that no new judgments can be inferred from S. A consistent set S is such that all judgments that cannot be inferred from S are not in S.

The inference operator is monotone: $F_\Phi(S) \subseteq F_\Phi(S')$ if $S \subseteq S'$. By Tarski's fixed point theorem for complete lattices [14, p. 286], it follows that the inference operator possesses both a least fixed point and a greatest fixed point, which are the smallest F_Φ-closed set and the largest F_Φ-consistent set, respectively.

$$\begin{aligned} \mathrm{lfp}(F_\Phi) &= \bigcap \{S \mid F_\Phi(S) \subseteq S\} \\ \mathrm{gfp}(F_\Phi) &= \bigcup \{S \mid S \subseteq F_\Phi(S)\} \end{aligned}$$

The least fixed point $\mathrm{lfp}(F_\Phi)$ is the *inductive interpretation* of the inference system Φ, and the greatest fixed point $\mathrm{gfp}(F_\Phi)$ is its *coinductive interpretation*. These interpretations lead to the following two proof principles:

- Induction principle: to prove that all judgments in the inductive interpretation belong to a set S, show that S is F_Φ-closed.
- Coinduction principle: to prove that all judgments in a set S belong to the coinductive interpretation, show that S is F_Φ-consistent.

2.2 Proof-theoretic approach

In contrast with the fixed point approach, the proof-theoretic approach starts from the *proofs* admissible in an inference system. These proofs naturally correspond to *derivations*, also called *proof trees*. These are trees whose nodes are labeled with judgments $c \in \mathcal{U}$ and such that for all nodes n, the label c of n and the labels $\mathcal{A}$ of the children of n correspond to an inference rule: $(\mathcal{A}, c) \in \Phi$. The conclusion of a derivation is the label of its root node.

A derivation d is *well-founded* if it has no infinite branch; d is *ill-founded* otherwise. If every rule in Φ has a finite set of premises, well-founded derivations are finite while ill-founded derivations are infinite.

In the proof-theoretic approach, the *inductive interpretation* of the inference system Φ is the set $\Delta(\Phi)$ of conclusions of well-founded derivations, while the *coinductive interpretation* is the set $\nabla(\Phi)$ of conclusions of arbitrary derivations (ill-founded or well-founded). These interpretations come with the following proof principles:

- Induction principle: to prove that all judgments in the inductive interpretation belong to a set S, proceed by structural induction over well-founded derivations. That is, show that $c \in S$ if c is the conclusion of a derivation d, assuming that $j \in S$ for all conclusions j of the strict subderivations of d.
- Coinduction principle: to prove that all judgments in a set S are in the coinductive interpretation, build a system of recursive equations between derivations, with unknowns $(x_j)_{j \in S}$. Each equation is of the form

$$x_j = \frac{x_{j_1} \qquad x_{j_2} \qquad \dots}{j}$$

  and must be justified by an inference rule: $(\{j_1, j_2, \dots\}, j) \in \Phi$. These equations are *guarded*, meaning that there are no trivial equations $x_j = x_{j'}$. It follows that the system has a unique solution [15], and this solution σ is such that for all $j \in S$, $\sigma(x_j)$ is a valid derivation that proves j. Therefore, all $j \in S$ are also in $\nabla(\Phi)$.

2.3 Equivalence between the two approaches

The following theorem shows that the interpretations defined using fixed points and using derivations coincide.

Theorem 1 *For all inference systems* Φ, $\mathrm{lfp}(F_\Phi) = \Delta(\Phi)$ *and* $\mathrm{gfp}(F_\Phi) = \nabla(\Phi)$.

Proof. It is easy to show that $\Delta(\Phi)$ is F_Φ-closed and that $\nabla(\Phi)$ is F_Φ-consistent. Therefore, $\mathrm{lfp}(F_\Phi) \subseteq \Delta(\Phi)$ and $\nabla(\Phi) \subseteq \mathrm{gfp}(F_\Phi)$.

Consider a F_Φ-closed set S. A structural induction over well-founded derivations d shows that the conclusion of d is in S. Therefore, $\Delta(\Phi) \subseteq S$. Since $\mathrm{lfp}(F_\Phi)$ is F_Φ-closed, the inclusion $\Delta(\Phi) \subseteq \mathrm{lfp}(F_\Phi)$ follows.

Finally, consider a F_Φ-consistent set S. For any judgment j in S, there exists a rule (K_j, j) in Φ, where $K_j \subseteq S$. We define a system of guarded recursive equations, with variables $(x_j)_{j \in S}$.

$$x_j = \frac{(x_k)_{k \in K_j}}{j}$$

The solution σ of this system is such that for all $j \in S$, the derivation $\sigma(x_j)$ is valid in Φ and proves j. Therefore, $S \subseteq \nabla(\Phi)$. Since $\mathrm{gfp}(F_\Phi)$ is F_Φ-consistent, the inclusion $\mathrm{gfp}(F_\Phi) \subseteq \nabla(\Phi)$ follows. □

The equality $\mathrm{lfp}(F_\Phi) = \Delta(\Phi)$ is proved by Aczel [13]. The equality $\mathrm{gfp}(F_\Phi) = \nabla(\Phi)$ is proved in the second author's PhD dissertation [16, p. 77], but to our knowledge there is no other published proof. This is, however, a well-known result. For instance, it has recently been used to extend logic programming with coinductive terms and derivations [17].

2.4 Induction and coinduction in the Coq proof assistant

The Coq proof assistant that we use to develop the present work follows the proof-theoretic formulation of induction and coinduction. In accordance with the propositions-as-types, proofs-as-programs paradigm, inference systems are presented as inductively or coinductively-defined predicates, resembling data type definitions in ML or Haskell. Such a predicate is defined by a set of constructors, corresponding to inference rules. Applied to terms representing proofs for its premises, a constructor returns a proof term for its conclusion.

Proofs by induction and by coinduction are both represented as recursive

functions. For a proof by induction, the Coq type system demands that the recursive function be *structural*: the arguments to recursive calls are strict subterms of the recursive parameter. For a proof by coinduction, the Coq type system demands that the recursive function be *productive*: its result is a constructor application, and the results of recursive calls are only used as arguments to this constructor. Such productive recursive functions correspond closely to the systems of guarded equations used above.

While proof terms can be provided explicitly by the user, most of the time they are built incrementally by the Coq proof assistant in response to tactics entered by the user. When using tactics, proofs by coinduction are as easy to conduct as proofs by induction: in response to the `cofix` tactic, the system provides the expected result as an additional hypothesis, then makes sure that this hypothesis is only used in positions permitted by productive recursive functions. (See [18] and [10, chap. 13] for more details, and the proof of lemma 5 below for a concrete example.) The proof sketches we give in the remainder of this article are written in the same proof style, and play fast and loose with coinduction. In particular, except for the very first proofs, we do not exhibit F_Φ-consistent sets nor systems of guarded equations between derivations. The skeptical reader is referred to the corresponding Coq development [19] for full details.

Coq is based on a constructive logic (the Calculus of Constructions), but proofs in classical logic can be expressed in Coq by adding axioms that are known to be consistent with Coq's logic. The majority of our proofs are constructive, but some use the axiom of excluded middle. The proofs that use this axiom are marked "**(classical)**".

3 The language and its big-step semantics

The language we consider in this article is the λ-calculus extended with constants: the simplest functional language that exhibits run-time errors (terms that "go wrong"). Its syntax is as follows:

$$\begin{array}{lrcl} \text{Variables:} & x, y, z, \ldots & & \\ \text{Constants:} & c & ::= & 0 \mid 1 \mid \ldots \\ \text{Terms:} & a, b, v & ::= & x \mid c \mid \lambda x.a \mid a\ b \end{array}$$

We write $a[x \leftarrow b]$ for the capture-avoiding substitution[1] of b for all free occurrences of x in a. We say that a term v is a value, and write $v \in \mathsf{Values}$,

[1] The Coq development does not treat terms modulo α-conversion, therefore the substitution $a[x \leftarrow b]$ can capture variables. However, it is capture-avoiding if b is closed, and this suffices to define evaluation and reduction of closed source terms.

if v is either a constant c or an abstraction $\lambda x.b$.

The standard call-by-value semantics in big-step style for this language is defined by the inductive interpretation of the following inference rules. They define the relation $a \Rightarrow v$ (read: "a evaluates to v").

$$c \Rightarrow c \quad (\Rightarrow\text{-const}) \qquad\qquad \lambda x.a \Rightarrow \lambda x.a \quad (\Rightarrow\text{-fun})$$

$$\frac{a_1 \Rightarrow \lambda x.b \qquad a_2 \Rightarrow v_2 \qquad b[x \leftarrow v_2] \Rightarrow v}{a_1\ a_2 \Rightarrow v}\ (\Rightarrow\text{-app})$$

Lemma 2 *If $a \Rightarrow v$, then $v \in \mathsf{Values}$.*

Proof. Induction on a derivation of $a \Rightarrow v$. □

Lemma 3 *The $\Rightarrow$ relation is deterministic: if $a \Rightarrow v$ and $a \Rightarrow v'$, then $v = v'$.*

Proof. By induction on the derivation of $a \Rightarrow v$ and case analysis over that of $a \Rightarrow v'$. □

The rules above capture only terminating evaluations. Writing $\delta = \lambda x.\ x\ x$ and $\omega = \delta\ \delta$, we have for instance:

Lemma 4 *$\omega \Rightarrow v$ is false for all terms v.*

Proof. We show that $a \Rightarrow v$ implies $a \neq \omega$ by induction on the derivation of $a \Rightarrow v$. □

Following Cousot and Cousot [8] and the second author's PhD work [16], we define divergence (infinite evaluations) by the coinductive interpretation[2] of the following inference rules. They define the relation $a \overset{\infty}{\Rightarrow}$ (read: "a diverges").

$$\frac{a_1 \overset{\infty}{\Rightarrow}}{a_1\ a_2 \overset{\infty}{\Rightarrow}}\ (\overset{\infty}{\Rightarrow}\text{-app-l}) \qquad\qquad \frac{a_1 \Rightarrow v \qquad a_2 \overset{\infty}{\Rightarrow}}{a_1\ a_2 \overset{\infty}{\Rightarrow}}\ (\overset{\infty}{\Rightarrow}\text{-app-r})$$

$$\frac{a_1 \Rightarrow \lambda x.b \qquad a_2 \Rightarrow v \qquad b[x \leftarrow v] \overset{\infty}{\Rightarrow}}{a_1\ a_2 \overset{\infty}{\Rightarrow}}\ (\overset{\infty}{\Rightarrow}\text{-app-f})$$

Note that we have imposed (arbitrarily) a left-to-right evaluation order for applications.

[2] Throughout this article, double horizontal lines in inference rules denote inference rules that are to be interpreted coinductively; single horizontal lines denote the inductive interpretation.

Lemma 5 $\omega \overset{\infty}{\Rightarrow}$ *holds.*

Proof. The proof is by coinduction. Assume $\omega \overset{\infty}{\Rightarrow}$ as coinduction hypothesis. We can derive $\omega \overset{\infty}{\Rightarrow}$ with rule ($\overset{\infty}{\Rightarrow}$-app-f), using the coinduction hypothesis as third premise.

Since this is the first proof by coinduction in this article, we now detail the proof sketch given above using the various approaches outlined in section 2.

Greatest fixed point. Consider the inference operator F associated with the rules defining $\overset{\infty}{\Rightarrow}$, namely

$$\begin{aligned} F(S) = \quad & \{a_1\ a_2 \mid a_1 \in S\} \\ & \cup \{a_1\ a_2 \mid \exists v,\ a_1 \Rightarrow v \wedge a_2 \in S\} \\ & \cup \{a_1\ a_2 \mid \exists x, b, v,\ a_1 \Rightarrow \lambda x.b \wedge a_2 \Rightarrow v \wedge b[x \leftarrow v] \in S\} \end{aligned}$$

The set $S = \{\omega\}$ is F-consistent. Indeed, $\omega \in F(\{\omega\})$ by the third line of the definition of F. Therefore, $S \subseteq \mathrm{gfp}(F)$, implying that $\omega \overset{\infty}{\Rightarrow}$ holds.

Systems of guarded recursive equations. Consider the following equation with unknown d (a derivation):

$$d = \frac{\delta \Rightarrow \lambda x.\ x\ x \qquad \delta \Rightarrow \delta \qquad d}{\delta\ \delta \overset{\infty}{\Rightarrow}}$$

Since $(x\ x)[x \leftarrow \delta] = \delta\ \delta$, this equation is justified by rule ($\overset{\infty}{\Rightarrow}$-app-f). Moreover, it is guarded. Therefore, its solution is a valid derivation that proves $\delta\ \delta \overset{\infty}{\Rightarrow}$. It follows that this judgment holds.

Coq proof term. Consider the Coq proof term `evalinf_omega` defined by the following corecursion:

```
CoFixpoint evalinf_omega : evalinf omega :=
  let eval_delta : eval delta delta :=
    eval_fun x (App (Var x) (Var x)) in
  evalinf_app_f delta delta x (App (Var x) (Var x)) delta
    eval_delta
    eval_delta
    evalinf_omega.
```

The two constructor functions `eval_fun` and `evalinf_app_f` correspond to the inference rules ($\Rightarrow$-fun) and ($\overset{\infty}{\Rightarrow}$-app-f), respectively. They receive as arguments instantiations for the free variables of the rules (x and a for ($\Rightarrow$-fun); a_1, a_2, x, b, v for ($\overset{\infty}{\Rightarrow}$-app-f)), followed by proof terms for their premises (proofs of $\delta \Rightarrow \delta$, $\delta \Rightarrow \delta$ and $\omega \overset{\infty}{\Rightarrow}$ for ($\overset{\infty}{\Rightarrow}$-app-f)). The term `evalinf_omega` has type `evalinf omega`, which proves that this proposition representing $\omega \overset{\infty}{\Rightarrow}$ is true.

Coq proof script. The following commented sequence of tactics builds the proof term above in an interactive manner.

```
Lemma evalinf_omega: evalinf omega.
Proof.
  cofix COINDHYP.
```
Prepare a proof by coinduction. The current goal $\omega \stackrel{\infty}{\Rightarrow}$ *becomes an hypothesis named* `COINDHYP`
```
  unfold omega. eapply evalinf_app_f.
```
Apply the constructor for rule $\stackrel{\infty}{\Rightarrow}$*-app-f*
```
  unfold delta. apply eval_fun.
```
Prove the first premise (evaluation of δ*)*
```
  unfold delta. apply eval_fun.
```
Prove the second premise (evaluation of δ*)*
```
  simpl. fold delta. fold omega.
```
Replace $(x\ x)[x \leftarrow \delta]$ *by* ω*.*
```
  apply COINDHYP.
```
Prove the third premise by invoking the coinduction hypothesis.
```
Qed.
```

□

Lemma 6 *$a \Rightarrow v$ and $a \stackrel{\infty}{\Rightarrow}$ are mutually exclusive.*

Proof. By induction on the derivation of $a \Rightarrow v$, case analysis on that of $a \stackrel{\infty}{\Rightarrow}$, and lemma 3. □

Programs that neither evaluate nor diverge according to the rules above are said to "go wrong". For instance, the program 0 0 goes wrong since neither $0\ 0 \Rightarrow v$ nor $0\ 0 \stackrel{\infty}{\Rightarrow}$ hold for any v.

4 Relation with small-step semantics

The one-step reduction relation $\rightarrow$ is defined by the call-by-value β-reduction axiom plus two context rules for reducing under applications, assuming left-to-right evaluation order.

$$\frac{v \in \mathsf{Values}}{(\lambda x.a)\ v \rightarrow a[x \leftarrow v]}\ (\rightarrow\text{-}\beta)$$

$$\frac{a_1 \rightarrow a_2}{a_1\ b \rightarrow a_2\ b}\ (\rightarrow\text{-app-l}) \qquad \frac{a \in \mathsf{Values} \quad b_1 \rightarrow b_2}{a\ b_1 \rightarrow a\ b_2}\ (\rightarrow\text{-app-r})$$

Lemma 7 *The $\to$ relation is deterministic: if $a \to a'$ and $a \to a''$, then $a' = a''$.*

Proof. By induction on the derivation of $a \to a'$ and case analysis over that of $a \to a''$. □

There are three kinds of reduction sequences of interest. The first, written $a \xrightarrow{*} b$ ("a reduces to b in zero, one or several steps"), is the standard reflexive transitive closure of $\to$; it captures finite reductions. The second, written $a \xrightarrow{\infty}$ ("a reduces infinitely"), captures infinite reductions. The third, written $a \xrightarrow{\mathsf{co}*} b$ ("a reduces to b in zero, one, several or infinitely many steps"), is the coinductive interpretation of the rules for reflexive transitive closure; it captures both finite and infinite reductions. These relations are defined by the following rules, interpreted inductively for $\xrightarrow{*}$ and coinductively for $\xrightarrow{\infty}$ and $\xrightarrow{\mathsf{co}*}$.

$$\frac{}{a \xrightarrow{*} a} \qquad \frac{a \to a' \quad a' \xrightarrow{*} b}{a \xrightarrow{*} b} \qquad \frac{a \to a' \quad a' \xrightarrow{\infty}}{a \xrightarrow{\infty}} \qquad \frac{}{a \xrightarrow{\mathsf{co}*} a} \qquad \frac{a \to a' \quad a' \xrightarrow{\mathsf{co}*} b}{a \xrightarrow{\mathsf{co}*} b}$$

It is true that $\xrightarrow{\mathsf{co}*}$ is the union of $\xrightarrow{*}$ and $\xrightarrow{\infty}$, in the following sense.

Lemma 8 *$a \xrightarrow{\mathsf{co}*} b$ if and only if $a \xrightarrow{*} b$ or $a \xrightarrow{\infty}$.*

Proof (classical). For the "if" part, we show that $a \xrightarrow{*} b \implies a \xrightarrow{\mathsf{co}*} b$ by induction on $a \xrightarrow{*} b$, and that $a \xrightarrow{\infty} \implies a \xrightarrow{\mathsf{co}*} b$ by coinduction. For the "only if" part, we show that $a \xrightarrow{\mathsf{co}*} b \wedge \neg(a \xrightarrow{*} b) \implies a \xrightarrow{\infty}$ by coinduction. The result follows by excluded middle over $a \xrightarrow{*} b$. □

We now turn to relating the reduction relations (small-step) and the evaluation relations (big-step). It is well known that normal evaluation is equivalent to finite reduction to a value.

Theorem 9 *$a \Rightarrow v$ if and only if $a \xrightarrow{*} v$ and $v \in \mathsf{Values}$.*

Proof. The "only if" part is an easy induction on $a \Rightarrow v$. For the "if" part, we first show the following two lemmas: (1) $v \Rightarrow v$ if $v \in \mathsf{Values}$, and (2) $a \Rightarrow v$ if $a \to b$ and $b \Rightarrow v$. The result follows by induction on the proof of $a \xrightarrow{*} v$. □

Similarly, divergence ($\overset{\infty}{\Rightarrow}$) is equivalent to infinite reduction ($\xrightarrow{\infty}$). The proof uses the following lemma.

Lemma 10 *For all terms a, either $a \xrightarrow{\infty}$, or there exists b such that $a \xrightarrow{*} b$ and $b \not\to$, that is, $\forall b',\ \neg(b \to b')$.*

Proof (classical). We first show that $\forall b,\ a \xrightarrow{*} b \implies \exists b',\ b \to b'$ implies

$a \stackrel{\infty}{\rightarrow}$ by coinduction. We then argue by excluded middle on $a \stackrel{\infty}{\rightarrow}$. □

Theorem 11 *$a \stackrel{\infty}{\Rightarrow}$ if and only if $a \stackrel{\infty}{\rightarrow}$.*

Proof (classical). For the "only if" part, we first show that $a \stackrel{\infty}{\Rightarrow}$ implies $\exists b,\ a \rightarrow b \wedge b \stackrel{\infty}{\Rightarrow}$ by structural induction on a, then conclude by coinduction. For the "if" part, we proceed by coinduction and case analysis over a. The only non-trivial case is $a = a_1\ a_2$. Using lemma 10, we distinguish three cases: (1) a_1 reduces infinitely; (2) a_1 reduces to a value but a_2 reduces infinitely; (3) a_1 and a_2 reduce to values $\lambda x.b$ and v respectively, and $b[x \leftarrow v]$ reduces infinitely. We conclude $a \stackrel{\infty}{\Rightarrow}$ by applying the appropriate inference rule for each case, the coinduction hypothesis for the $\stackrel{\infty}{\Rightarrow}$ premise, and theorem 9 for the $\Rightarrow$ premises. □

5 Relation with denotational semantics

Denotational semantics is an alternate way to characterize divergent and convergent terms. In this section, we develop a simple denotational semantics for call-by-value λ-calculus and prove that it captures the same notions of convergence and divergence as our big-step operational semantics. To facilitate the mechanization of these results in the Coq theorem prover, we adopt an elementary presentation of the denotational semantics that does not require the full generality of Scott domains.

We define the computation $\mathcal{C}_n(a)$ of a term a at maximal recursion depth $n \in I\!N$ by recursion over n, as follows.

$$\begin{aligned}
\mathcal{C}_0(a) &= \bot \\
\mathcal{C}_{n+1}(x) &= \texttt{err} \\
\mathcal{C}_{n+1}(c) &= c \\
\mathcal{C}_{n+1}(\lambda x.a) &= \lambda x.a \\
\mathcal{C}_{n+1}(a_1\ a_2) &= \mathcal{C}_n(a_1) \rhd (v_1 \mapsto \\
&\qquad \mathcal{C}_n(a_2) \rhd (v_2 \mapsto \\
&\qquad\quad \texttt{if } v_1 = \lambda x.b \texttt{ then } \mathcal{C}_n(b[x \leftarrow v_2]) \texttt{ else err}))
\end{aligned}$$

The monadic composition operator $\rhd$ used in the application case is defined by

$$\bot \rhd f = \bot \qquad \texttt{err} \rhd f = \texttt{err} \qquad v \rhd f = f(v).$$

The result of $\mathcal{C}_n(a)$, or in other terms the outcome of executing a at depth n, is one of the following three possibilities: (1) a value v, denoting normal termination with v as final value; (2) the symbol `err`, denoting abrupt termination on a run-time error (such as encountering a free variable or an application of a

constant); (3) the symbol $\bot$, indicating that the computation cannot complete within n recursive steps.

The flat ordering $\leq$ over results is defined by $\bot \leq r$ and $r \leq r$ for all r. The $\mathcal{C}$ function is monotone with respect to this ordering:

Lemma 12 *If $n \leq m$, then $\mathcal{C}_n(a) \leq \mathcal{C}_m(a)$.*

Proof. By induction over n and case analysis over a. □

We say that a term a executes with result r, or in other terms that r is the denotation of a, and we write $\mathcal{D}(a, r)$, if $\mathcal{C}_n(a) = r$ for almost all n:

$$\mathcal{D}(a, r) \stackrel{\text{def}}{=} \exists p,\ \forall n,\ n \geq p \implies \mathcal{C}_n(a) = r.$$

Since $\mathcal{C}$ is monotone, the following properties hold trivially:

Lemma 13 *If $\mathcal{D}(a, r)$, then for all n, either $\mathcal{C}_n(a) = \bot$ or $\mathcal{C}_n(a) = r$.*

Lemma 14 *If $r \neq \bot$ and $\mathcal{C}_n(a) = r$ for some n, then $\mathcal{D}(a, r)$.*

Lemma 15 *$\mathcal{D}(a, \bot)$ if and only if $\mathcal{C}_n(a) = \bot$ for all n.*

It follows that every term has one and exactly one denotation.

Lemma 16 *For all terms a, there exists a result r such that $\mathcal{D}(a, r)$.*

Proof (classical). By excluded middle, either $\forall n, \mathcal{C}_n(a) = \bot$ or $\exists n, \mathcal{C}_n(a) \neq \bot$. In the former case, we obviously have $\mathcal{D}(a, \bot)$. In the latter case, pick n such that $\mathcal{C}_n(a) \neq \bot$ and take $r = \mathcal{C}_n(a)$. By lemma 14, we have $\mathcal{D}(a, r)$. □

Lemma 17 *If $\mathcal{D}(a, r_1)$ and $\mathcal{D}(a, r_2)$, then $r_1 = r_2$.*

Proof. Notice that $r_1 = \mathcal{C}_n(a) = r_2$ for sufficiently large n. □

We now relate this denotational semantics with the big-step operational semantics of section 3, starting with the terminating case.

Theorem 18 *$a \Rightarrow v$ if and only if $\mathcal{D}(a, v)$.*

Proof. For the "if" part, we show that $\mathcal{C}_n(a) = v$ implies $a \Rightarrow v$ by induction over n and case analysis over a and over the results of the recursive computations. The case $a = x$ contradicts the hypothesis $\mathcal{C}_n(a) = v$. For the cases $a = c$ or $a = \lambda x.b$, we have $v = a$ by definition of $\mathcal{C}$ and the result follows by rules ($\Rightarrow$-const) or ($\Rightarrow$-fun). Finally, if $a = a_1\ a_2$, the exploitation of the hypothesis $\mathcal{C}_n(a) = v$ leads to $\mathcal{C}_{n-1}(a_1) = \lambda x.b$ and $\mathcal{C}_{n-1}(a_2) = v_2$ and $\mathcal{C}_{n-1}(b[x \leftarrow v_2]) = v$. The result follows from the induction hypothesis and rule ($\Rightarrow$-app).

For the "only if" part, we proceed by induction over the derivation of $a \Rightarrow v$ and exhibit an n such that $\mathcal{C}_n(a) = v$. From this, $\mathcal{D}(a, v)$ follows by lemma 14. The cases where a is a constant or a function are trivial, since $\mathcal{C}_1(a) = v$ in these cases. For the application case $a = a_1\ a_2$, the induction hypothesis leads to $\mathcal{C}_{n_1}(a_1) = \lambda x.b$ and $\mathcal{C}_{n_2}(a_2) = v_2$ and $\mathcal{C}_{n_3}(b[x \leftarrow v_2]) = v$ for some n_1, n_2, n_3. Taking $n = 1 + \max(n_1, n_2, n_3)$, we have $\mathcal{C}_n(a) = v$ by definition and monotonicity of $\mathcal{C}$, and the result follows. □

Theorem 19 *$a \stackrel{\infty}{\Rightarrow}$ if and only if $\mathcal{D}(a, \bot)$.*

Proof. For the "only if" part, we show that $a \stackrel{\infty}{\Rightarrow}$ implies $\mathcal{C}_n(a) = \bot$ by induction over n and case analysis on the last rule used in the derivation of $a \stackrel{\infty}{\Rightarrow}$. In all three cases, $a = a_1\ a_2$. If $a_1 \stackrel{\infty}{\Rightarrow}$, $\mathcal{C}_n(a) = \mathcal{C}_{n-1}(a_1) = \bot$ by induction hypothesis. If $a_1 \Rightarrow v_1$ and $a_2 \stackrel{\infty}{\Rightarrow}$, we have $\mathcal{D}(a, v_1)$ by theorem 18. By induction hypothesis, $\mathcal{C}_{n-1}(a_2) = \bot$. By lemma 13, either $\mathcal{C}_{n-1}(a_1) = \bot$ or $\mathcal{C}_{n-1}(a_1) = v_1$. In both cases, $\mathcal{C}_n(a) = \bot$. The third and last case ($a_1 \Rightarrow \lambda x.b$ and $a_2 \Rightarrow v_2$ and $b[x \leftarrow v_2] \stackrel{\infty}{\Rightarrow}$) is similar.

The "if" part is proved by coinduction and case analysis over a. The cases $a = x$, $a = c$ and $a = \lambda x.b$ trivially contradict the hypothesis $\mathcal{D}(a, \bot)$. Therefore, it must be the case that $a = a_1\ a_2$. Let r_1 and r_2 be the denotations of a_1 and a_2. (They exist by lemma 16.) We argue by case over r_1 and r_2, exploiting the definition of $\mathcal{C}$ for sufficiently large values of n. There are only three cases that do not contradict the hypothesis $\mathcal{D}(a, \bot)$: (1) $r_1 = \bot$; (2) r_1 is a value v_1 and $r_2 = \bot$; (3) r_1 is a value $\lambda x.b$ and r_2 is a value v_2 and $\mathcal{D}(b[x \leftarrow v_2], \bot)$. We conclude $a \stackrel{\infty}{\Rightarrow}$ by applying the appropriate inference rule for each case, the coinduction hypothesis for the $\stackrel{\infty}{\Rightarrow}$ premise, and theorem 18 for the $\Rightarrow$ premises. □

6 Extension to trace semantics

Besides expressing both terminating and diverging executions, small-step semantics have another advantage over big-step semantics: reduction sequences contain all intermediate reducts of the source term in addition to its final value, therefore providing a complete trace of the execution. Such execution traces are useful both for static analysis (by abstract interpretation of collecting semantics) and to state and prove stronger semantic preservation properties for program transformations. In particular, when the input language is imperative and features observable actions such as input/output, traces of observable events are crucial to state and prove observational equivalence results.

In this section, following the second author's work [16], we show how to extend the big-step semantics of section 3 so that they produce not only the outcome

of an evaluation (final value or divergence), but also a (possibly infinite) execution trace.

6.1 Traces

The traces we consider are finite or infinite sequences of terms representing the intermediate reducts of the source program.

Finite traces: $t ::= \epsilon \mid a.t$ (inductive interpretation)
Infinite traces: $T ::= a.T$ (coinductive interpretation)

By abuse of notation, we write $t.t'$ and $t.T$ for the concatenation of a finite trace t and a finite or infinite trace. Concatenation is associative and ϵ is a neutral element for concatenation.

If $t = a_1.a_2 \dots a_n$ is a finite trace, we define the left application $t\ b$ of this trace to a term b and the right application $v\ t$ of a value v to this trace as follows:

$$t\ b = (a_1\ b).(a_2\ b) \dots (a_n\ b)$$
$$v\ t = (v\ a_1).(v\ a_2) \dots (v\ a_n)$$

We similarly define the applications $T\ b$ and $v\ T$ where T is an infinite trace.

We define bisimilarity between infinite traces, written $T_1 \cong T_2$, by the following coinductive rule:

$$\frac{T_1 \cong T_2}{a.T_1 \cong a.T_2}$$

Concatenation and application of traces are compatible with bisimilarity.

In set theory, bisimilarity is equivalent to equality. In Coq's constructive logic, bisimilarity is coarser than equality: there exists infinite traces that are bisimilar but cannot be proved equal [10, chap. 13]. Some of the following results require the use of bisimilarity instead of equality in definitions and statements, in order to be provable in Coq.

6.2 Small-step semantics with traces

While our objective is to instrument big-step semantics to produce execution traces, we start by doing this for the small-step semantics, which is easier and helps us define precisely the traces we expect for an execution. For a finite reduction sequence $a_1 \to a_2 \to \dots \to a_{n-1} \to a_n$, the expected (finite) trace

is $t = a_1.a_2 \ldots a_{n-1}$, that is, the initial term and its intermediate reducts but not the final term. Equivalently, the trace comprises the source terms for all reduction steps performed in the sequence. This is formalized by the following rules for the predicate $a \xrightarrow{*} a' \,/\, t$ (read: "a reduces in zero, one or several steps to a' with trace t").

$$a \xrightarrow{*} a \,/\, \epsilon \qquad\qquad \frac{a \to a' \quad a' \xrightarrow{*} b \,/\, t}{a \xrightarrow{*} b \,/\, a.t}$$

For an infinite reduction sequence $a_1 \to \ldots \to a_n \to \ldots$, the expected (infinite) trace is $T = a_1 \ldots a_n \ldots$ This is captured by the following coinductive rule defining the predicate $a \xrightarrow{\infty} \,/\, T$ (read: "a reduces infinitely with trace T").

$$\frac{a \to b \quad b \xrightarrow{\infty} \,/\, T}{\quad a \xrightarrow{\infty} \,/\, a.T \quad}$$

It is intuitively clear that the small-step semantics with traces is a refinement of that without traces. We now formalize this intuition, which is not obvious to prove constructively in the case of infinite reductions.

Lemma 20 *$a \xrightarrow{*} b$ if and only if $\exists t,\ a \xrightarrow{*} b \,/\, t$.*

Proof. Straightforward by induction over the reduction sequences $a \xrightarrow{*} b$ and $a \xrightarrow{*} b \,/\, t$. □

Lemma 21 *$a \xrightarrow{\infty}$ if and only if $\exists T,\ a \xrightarrow{\infty} \,/\, T$.*

Proof. The "if" part is an easy proof by coinduction. The "only if" part is more involved: since the conclusion $\exists T,\ a \xrightarrow{\infty} \,/\, T$ is not a coinductively-defined predicate, we cannot reason directly by coinduction. Instead, we must construct explicitly a suitable infinite trace T. To this end, we first define a reduction function $\mathcal{R}$ from terms to optional terms that is equivalent to the one-step reduction predicate, that is

$$\mathcal{R}(a) = \begin{cases} \mathtt{Some}(b), & \text{if } a \to b; \\ \mathtt{None}, & \text{if } a \not\to. \end{cases}$$

This function is total (by induction over a), therefore proving that one-step reduction is decidable. Next, to every term a we associate an infinite trace $\mathcal{T}(a)$ of all the successive reducts of a. This trace is defined, by guarded corecursion, as

$$\mathcal{T}(a) = \begin{cases} a.\mathcal{T}(b), & \text{if } \mathcal{R}(a) = \mathtt{Some}(b); \\ a.\mathcal{T}(a), & \text{if } \mathcal{R}(a) = \mathtt{None}. \end{cases}$$

We then show that $a \xrightarrow{\infty}$ implies $a \xrightarrow{\infty} \,/\, \mathcal{T}(a)$. This follows by coinduction from the fact that $\mathcal{T}(a) = a.\mathcal{T}(b)$ whenever $a \to b$. □

As a corollary, we obtain the following analogue of lemma 10.

Lemma 22 *For all terms a, either there exist a term b and a trace t such that $a \xrightarrow{*} b \,/\, t$ and $b \not\to$, or there exists an infinite trace T such that $a \xrightarrow{\infty} \,/\, T$.*

Proof (classical). Follows from lemmas 10, 20 and 21. □

Additionally, the trace-based reduction relations are deterministic up to bisimilarity between infinite traces. This is an immediate consequence of the determinism of one-step reductions (lemma 7).

Lemma 23 *If $a \xrightarrow{*} v_1 \,/\, t_1$ and $a \xrightarrow{*} v_2 \,/\, t_2$, then $t_1 = t_2$ and $v_1 = v_2$.*

Lemma 24 *If $a \xrightarrow{\infty} \,/\, T_1$ and $a \xrightarrow{\infty} \,/\, T_2$, then $T_1 \cong T_2$.*

Note that the stronger conclusion $T_1 = T_2$ is not provable in Coq. Another consequence of the determinism of one-step reductions is the following obvious decomposition property for infinite reductions.

Lemma 25 *If $a \xrightarrow{\infty} \,/\, T$ and $a \xrightarrow{*} b \,/\, t$, there exists T' such that $b \xrightarrow{\infty} \,/\, T'$ and $T = t.T'$.*

6.3 Big-step semantics with traces

We now add traces to the big-step definitions of evaluation and divergence. The corresponding predicates are $a \Rightarrow v \,/\, t$ ("a evaluates to v with finite trace t") and $a \overset{\infty}{\Rightarrow} \,/\, T$ ("a diverges with infinite trace T").

$$\frac{}{c \Rightarrow c \,/\, \epsilon}\ (\Rightarrow\text{-const}) \qquad \frac{}{\lambda x.a \Rightarrow \lambda x.a \,/\, \epsilon}\ (\Rightarrow\text{-fun})$$

$$\frac{\begin{array}{c} a_1 \Rightarrow \lambda x.b \,/\, t_1 \quad a_2 \Rightarrow v_2 \,/\, t_2 \quad b[x \leftarrow v_2] \Rightarrow v \,/\, t_3 \\ t = (t_1\ a_2).((\lambda x.b)\ t_2).((\lambda x.b)\ v_2).t_3 \end{array}}{a_1\ a_2 \Rightarrow v \,/\, t}\ (\Rightarrow\text{-app})$$

$$\frac{a_1 \overset{\infty}{\Rightarrow} \,/\, T_1 \quad T \cong T_1\ a_2}{a_1\ a_2 \overset{\infty}{\Rightarrow} \,/\, T}\ (\overset{\infty}{\Rightarrow}\text{-app-l})$$

$$\frac{a_1 \Rightarrow v \,/\, t_1 \quad a_2 \overset{\infty}{\Rightarrow} \,/\, T_2 \quad T \cong (t_1\ a_2).(v\ T_2)}{a_1\ a_2 \overset{\infty}{\Rightarrow} \,/\, T}\ (\overset{\infty}{\Rightarrow}\text{-app-r})$$

$$\frac{\begin{array}{c} a_1 \Rightarrow \lambda x.b \,/\, t_1 \quad a_2 \Rightarrow v_2 \,/\, t_2 \quad b[x \leftarrow v_2] \overset{\infty}{\Rightarrow} \,/\, T_3 \\ T \cong (t_1\ a_2).((\lambda x.b)\ t_2).((\lambda x.b)\ v_2).T_3 \end{array}}{a_1\ a_2 \overset{\infty}{\Rightarrow} \,/\, T}\ (\overset{\infty}{\Rightarrow}\text{-app-f})$$

The construction of the trace in the rules for applications is justified as follows. Assume, for instance, $a_1 \Rightarrow \lambda x.b \,/\, t_1$ and $a_2 \Rightarrow v_2 \,/\, t_2$. The application $a_1\ a_2$ performs one β-reduction $(\lambda x.b)\ v_2 \to b[x \leftarrow v_2]$ in addition to those coming from the evaluations of the premises of the rule. The source term for this reduction, $(\lambda x.b)\ v_2$, is therefore added to the trace. It is preceded by $t_1\ a_2$ (the trace for a_1 put into a left application context $[\,]\ a_2$) and by $(\lambda x.b)\ t_2$ (the trace for a_2 put into a right application context $(\lambda x.b)\ [\,]$). The source of the β-reduction is then followed by the trace corresponding to the evaluation of the function body $b[x \leftarrow v_2]$.

Another point to note is the use of bisimilarity $T \cong \ldots$ instead of equality $T = \ldots$ in the coinductive rules defining $\stackrel{\infty}{\Rightarrow}$. This allows traces to be replaced by bisimilar traces at every inference step, therefore enabling us to prove more statements about $\stackrel{\infty}{\Rightarrow}$ within the limits of Coq's coinductive proofs. (For instance, the proof of theorem 31 no longer goes through if $\stackrel{\infty}{\Rightarrow}$ is defined with equalities between traces instead of bisimilarities.) This subtle point is moot in set theory, where bisimilarity is equivalent to equality.

Lemma 26 *$\omega \stackrel{\infty}{\Rightarrow} \,/\, T$ holds where T is the infinite trace $\omega.\omega.\omega\ldots$*

Proof. By coinduction, using rule ($\stackrel{\infty}{\Rightarrow}$-app-f). □

6.4 Equivalence between the trace semantics

We now show the equivalence between the big-step and small-step semantics with traces, extending the results of section 4.

Theorem 27 *$a \Rightarrow v \,/\, t$ if and only if $a \stackrel{*}{\to} v \,/\, t$ and $v \in \mathsf{Values}$.*

Proof. The "only if" part is an easy induction on the derivation of $a \Rightarrow v \,/\, t$. For the "if" part, we first show the following two lemmas: (1) $v \Rightarrow v \,/\, \epsilon$ if $v \in \mathsf{Values}$, and (2) $a \Rightarrow v \,/\, a.t$ if $a \to b$ and $b \Rightarrow v \,/\, t$. The result follows by induction on the derivation of $a \stackrel{*}{\to} v \,/\, t$. □

Theorem 28 *$a \stackrel{\infty}{\Rightarrow} \,/\, T$ implies $a \stackrel{\infty}{\to} \,/\, T$.*

Proof. We first show by induction on a that $a \stackrel{\infty}{\Rightarrow} \,/\, T$ implies the existence of b and T' such that $a \to b$ and $b \stackrel{\infty}{\Rightarrow} \,/\, T'$ and $T \cong a.T'$. We then define the following variant $\stackrel{\infty,\cong}{\longrightarrow}$ of the infinite reduction predicate, by the coinductive inference rule

$$\frac{a \to b \qquad b \stackrel{\infty,\cong}{\longrightarrow} \,/\, T' \qquad T \cong a.T'}{a \stackrel{\infty,\cong}{\longrightarrow} \,/\, T}$$

This variant enables us to replace the infinite trace T by a bisimilar one at

every proof step, while remaining within the subset of proofs that Coq accepts as productively coinductive. We can therefore show that $a \stackrel{\infty}{\Rightarrow} / T$ implies $a \stackrel{\infty,\cong}{\longrightarrow} / T$ by coinduction, using the decomposition property stated earlier. We conclude by proving that $a \stackrel{\infty,\cong}{\longrightarrow} / T$ implies $a \stackrel{\infty}{\rightarrow} / T$, again by coinduction. □

As a corollary of theorem 28, the big-step divergence relation $\stackrel{\infty}{\Rightarrow}$ is deterministic up to bisimilarity of the traces. It is interesting to note that we could not find a more direct Coq proof of this fact.

Lemma 29 *If $a \stackrel{\infty}{\Rightarrow} / T_1$ and $a \stackrel{\infty}{\Rightarrow} / T_2$, then $T_1 \cong T_2$.*

Proof. Follows from lemma 24 and theorem 28. □

The converse of theorem 28 relies on the following inversion lemma for infinite reduction sequences starting with an application.

Lemma 30 *Assume $a\ b \stackrel{\infty}{\rightarrow} / T$.*

(1) If $a \stackrel{\infty}{\rightarrow} / T'$, then $T \cong T'\ b$.
(2) If $a \in$ Values and $b \stackrel{\infty}{\rightarrow} / T'$, then $T \cong a\ T'$.
(3) If $a \stackrel{*}{\rightarrow} a' / t$, then there exists T' such that $a'\ b \stackrel{\infty}{\rightarrow} / T'$ and $T = (t\ b).T'$.
(4) If $a \in$ Values and $b \stackrel{*}{\rightarrow} b' / t$, then there exists T' such that $a\ b' \stackrel{\infty}{\rightarrow} / T'$ and $T = (a\ t).T'$.

Proof. For (1) and (2), we show by coinduction that $a\ b \stackrel{\infty}{\rightarrow} / T'\ b$ and $a\ b \stackrel{\infty}{\rightarrow} / a\ T'$, respectively, then conclude by lemma 24.

Property (3) follows from the decomposition lemma 25 and the fact that $a\ b \stackrel{*}{\rightarrow} a'\ b / t\ b$ whenever $a \stackrel{*}{\rightarrow} a' / t$. Similarly, property (4) follows from the decomposition lemma 25 and the fact that $a\ b \stackrel{*}{\rightarrow} a\ b' / a\ t$ if $a \in$ Values and $b \stackrel{*}{\rightarrow} b' / t$. □

Theorem 31 *$a \stackrel{\infty}{\rightarrow} / T$ implies $a \stackrel{\infty}{\Rightarrow} / T$.*

Proof (classical). The proof proceeds by coinduction and case analysis over a. It must be the case that $a = a_1\ a_2$, otherwise a cannot reduce infinitely. Using lemma 22, we distinguish three cases:

(1) $a_1 \stackrel{\infty}{\rightarrow} / T_1$. This implies $a_1 \stackrel{\infty}{\Rightarrow} / T_1$ by coinduction hypothesis. Moreover, we have $T \cong T_1\ a_2$ by case (1) of lemma 30, which implies the expected result by rule ($\stackrel{\infty}{\Rightarrow}$-app-l).
(2) $a_1 \stackrel{*}{\rightarrow} v / t_1$ and $v \not\rightarrow$ and $a_2 \stackrel{\infty}{\rightarrow} / T_2$. By case (3) of lemma 30, we have $v\ a_2 \stackrel{\infty}{\rightarrow} / T'$ for some T' such that $T = (t_1\ a_2).T'$. This implies that $v \in$ Values. Moreover, $T' \cong v\ T_2$ by case (2) of lemma 30. Theorem 27 gives $a_1 \Rightarrow v / t$ and the coinduction hypothesis gives $a_2 \stackrel{\infty}{\Rightarrow} T_2$. The result

follows from rule ($\stackrel{\infty}{\Rightarrow}$-app-r).

(3) $a_1 \stackrel{*}{\rightarrow} v_1 \,/\, t_1$ and $v_1 \not\rightarrow$ and $a_2 \stackrel{*}{\rightarrow} v_2 \,/\, t_2$ and $v_2 \not\rightarrow$. Using cases (3) and (4) of lemma 30, it follows that $v_1 = \lambda x.b$ for some x, b, that $v_2 \in \mathsf{Values}$, and that $(\lambda x.b)\ v_2 \stackrel{\infty}{\rightarrow}\ / \ T'$ for some T' such that $T = (t_1\ a_2).((\lambda x.b)\ t_2).T'$. By inversion, we deduce $b[x \leftarrow v_2] \stackrel{\infty}{\rightarrow}\ / \ T_3$ for some T_3 such that $T' \cong ((\lambda x.b)\ v_2).T_3$. The result follows by rule ($\stackrel{\infty}{\Rightarrow}$-app-f), the coinduction hypothesis, and theorem 27.

□

7 Coevaluation

7.1 Definition and properties

So far, we have described terminating and non-terminating evaluations using two separate sets of inference rules, one interpreted inductively and the other coinductively. An attempt to describe both kinds of evaluations at the same time, in a more concise way, is to interpret coinductively the standard evaluation rules for terminating evaluations. This defines the relation $a \stackrel{\mathsf{co}}{\Rightarrow} b$ (read: "a coevaluates to b").

$$c \stackrel{\mathsf{co}}{\Rightarrow} c \quad (\stackrel{\mathsf{co}}{\Rightarrow}\text{-const}) \qquad\qquad \lambda x.a \stackrel{\mathsf{co}}{\Rightarrow} \lambda x.a \quad (\stackrel{\mathsf{co}}{\Rightarrow}\text{-fun})$$

$$\frac{a_1 \stackrel{\mathsf{co}}{\Rightarrow} \lambda x.b \qquad a_2 \stackrel{\mathsf{co}}{\Rightarrow} v_2 \qquad b[x \leftarrow v_2] \stackrel{\mathsf{co}}{\Rightarrow} v}{a_1\ a_2 \stackrel{\mathsf{co}}{\Rightarrow} v} \ (\stackrel{\mathsf{co}}{\Rightarrow}\text{-app})$$

It is clear from the definition of $\stackrel{\mathsf{co}}{\Rightarrow}$ that coevaluation includes all terminating evaluations, plus some diverging ones.

Lemma 32 *If $a \Rightarrow v$, then $a \stackrel{\mathsf{co}}{\Rightarrow} v$.*

Proof. By induction on the derivation of $a \Rightarrow v$. □

Lemma 33 *$\omega \stackrel{\mathsf{co}}{\Rightarrow} v$ for all terms v.*

Proof. By coinduction, using rule ($\stackrel{\mathsf{co}}{\Rightarrow}$-app) with the coinduction hypothesis as third premise. □

Naively, we could expect that $\stackrel{\mathsf{co}}{\Rightarrow}$ is equivalent to the union of the $\Rightarrow$ and $\stackrel{\infty}{\Rightarrow}$ relations. This equivalence holds in one direction only, from coevaluation to evaluation.

Lemma 34 *If $a \stackrel{\mathsf{co}}{\Rightarrow} v$, then either $a \Rightarrow v$ or $a \stackrel{\infty}{\Rightarrow}$.*

Proof (classical). We show that $a \overset{\mathsf{co}}{\Rightarrow} v$ and $\neg(a \Rightarrow v)$ implies $a \overset{\infty}{\Rightarrow}$. The result then follows by excluded middle on $a \Rightarrow v$. The auxiliary property is proved by coinduction and case analysis on a. The cases for variables, constants and abstractions trivially contradict one of the hypotheses. If $a = a_1\ a_2$, an inversion on the hypothesis $a \overset{\mathsf{co}}{\Rightarrow} v$ shows that $a_1 \overset{\mathsf{co}}{\Rightarrow} \lambda x.b$ and $a_2 \overset{\mathsf{co}}{\Rightarrow} v_2$ and $b[x \leftarrow v_2] \overset{\mathsf{co}}{\Rightarrow} v$. Using excluded middle, it must be that at least one of these three terms does not evaluate, otherwise, $a \Rightarrow v$ would hold. The result follows by applying the rule for $\overset{\infty}{\Rightarrow}$ that matches the term that does not evaluate, and using the coinduction hypothesis. □

However, the reverse implication from evaluation to coevaluation does not hold: there exists terms that diverge but do not coevaluate. Consider for instance $a = \omega\ (0\ 0)$. It is true that $a \overset{\infty}{\Rightarrow}$, but there is no term v such that $a \overset{\mathsf{co}}{\Rightarrow} v$, because the coevaluation of the argument $0\ 0$ goes wrong (there is no v such that $0\ 0 \overset{\mathsf{co}}{\Rightarrow} v$). Section 8.2 shows another example of a diverging term that does not coevaluate, this time involving no subterm that goes wrong.

Another unusual feature of coevaluation is that it is not deterministic. For instance, $\omega \overset{\mathsf{co}}{\Rightarrow} v$ for any term v. However, $\overset{\mathsf{co}}{\Rightarrow}$ is deterministic for terminating terms, in the following sense:

Lemma 35 *If $a \Rightarrow v$ and $a \overset{\mathsf{co}}{\Rightarrow} v'$, then $v' = v$.*

Proof. By induction on the derivation of $a \Rightarrow v$ and inversion on $a \overset{\mathsf{co}}{\Rightarrow} v'$. □

Moreover, there exists diverging terms that coevaluate to only one value. An example is $(\lambda x.0)\ \omega$, which coevaluates to 0 but not to any other term.

7.2 Connection with small-step semantics

Concerning the connections between coevaluation (big-step) and coreduction (small-step) in the style of section 4, the expected equivalence between $\overset{\mathsf{co}}{\Rightarrow}$ and $\overset{\mathsf{co}*}{\rightarrow}$ holds in one direction only.

Lemma 36 *$a \overset{\mathsf{co}}{\Rightarrow} v$ implies $a \overset{\mathsf{co}*}{\rightarrow} v$.*

Proof. Using classical logic, this follows from lemmas 34 and equivalence theorems 9, 11 and 8. However, the result can be proved directly in constructive logic. We first show that $a \overset{\mathsf{co}}{\Rightarrow} v \Longrightarrow a \in \mathsf{Values} \vee \exists b,\ a \rightarrow b \wedge b \overset{\mathsf{co}}{\Rightarrow} v$ by induction on a. The result follows by coinduction. □

The reverse implication obviously does not hold for terms a that diverge but do not coevaluate, such as the term $a = \omega\ (0\ 0)$ mentioned previously: if $a \overset{\infty}{\Rightarrow}$, we have $a \overset{\infty}{\rightarrow}$ and therefore $a \overset{\mathsf{co}*}{\rightarrow} v$ for any v, but $a \overset{\mathsf{co}}{\Rightarrow} v$ does not hold. Another

counterexample to the reverse implication is $a = (\lambda x.\ 0)\ \omega$ and $v = 1$. Since $a \xrightarrow{\infty}$, we have $a \overset{\mathsf{co}*}{\to} v$. However, $a \overset{\mathsf{co}}{\Rightarrow} v$ does not hold since the only term to which a coevaluates is 0.

7.3 Coevaluation for CPS terms

Notwithstanding the negative results of sections 7.1 and 7.2, there exists a class of terms for which coevaluation correctly captures both terminating and diverging evaluations: terms that are in continuation-passing style (CPS). A distinguishing feature of these terms is that function arguments are always values. CPS terms are defined by the following grammar:

$$\begin{array}{lll} a \in \mathsf{Atoms} & ::= & x \mid c \mid \lambda x.b \\ b \in \mathsf{CPS\text{-}terms} & ::= & a \mid b\ a \end{array}$$

Less formally, CPS terms are built from atoms (variables, constants and function abstractions) using multiple applications in tail-call position.

It is well known that CPS terms are stable by substitution of atoms for variables.

Lemma 37 *If $a \in \mathsf{Atoms}$ and $b \in \mathsf{CPS\text{-}terms}$, then $b[x \leftarrow a] \in \mathsf{CPS\text{-}terms}$.*

Consequently, the value of a CPS term is an atom.

Lemma 38 *If $b \in \mathsf{CPS\text{-}terms}$ and $b \Rightarrow v$, then $v \in \mathsf{Atoms}$. As a corollary, if $b \in \mathsf{CPS\text{-}terms}$ and $b \Rightarrow \lambda x.b'$, then $b' \in \mathsf{CPS\text{-}terms}$.*

Proof. By induction on the derivation of $b \Rightarrow v$, using lemma 37 for the application case. □

The main result of this section is that a closed CPS term coevaluates to a value if and only if it evaluates or it diverges. The restriction to closed terms is important since, for instance, the CPS term $\omega\ x$ diverges but its coevaluation goes wrong on the free variable x.

The following lemma lists useful properties of CPS atoms.

Lemma 39 *Let $a \in \mathsf{Atoms}$.*

(1) $a \Rightarrow a$ if a is closed.
(2) It is not the case that $a \overset{\infty}{\Rightarrow}$.
(3) If $a \Rightarrow v$, then $v = a$.

The key technical lemma below shows that diverging, closed CPS terms co-evaluate to a well-chosen value.

Lemma 40 *Define $\Omega = \lambda x.\omega$. If $b \in$ CPS-terms, b is closed and $b \stackrel{\infty}{\Rightarrow}$, then $b \stackrel{\mathsf{co}}{\Rightarrow} \Omega$.*

Proof. By coinduction. The CPS term b cannot be an atom (this would contradict the divergence hypothesis), therefore $b = b'\ a$ with b' a closed CPS term and a a closed CPS atom. Analysis on the last rule used in the derivation of $b \stackrel{\infty}{\Rightarrow}$ reveals three cases. In the first case, $b' \stackrel{\infty}{\Rightarrow}$. By coinduction hypothesis, $b' \stackrel{\mathsf{co}}{\Rightarrow} \Omega = \lambda x.\omega$. By lemmas 39 and 32, $a \stackrel{\mathsf{co}}{\Rightarrow} a$. Finally, $\omega[x \leftarrow a] = \omega$ coevaluates to Ω by lemma 33. Applying rule ($\stackrel{\mathsf{co}}{\Rightarrow}$-app), it follows that $b \stackrel{\mathsf{co}}{\Rightarrow} \Omega$.

The second case, $a \stackrel{\infty}{\Rightarrow}$, is impossible by lemma 39. This leaves the third case: $b' \Rightarrow \lambda x.b''$ and $a \Rightarrow v$ and $b''[x \leftarrow v] \stackrel{\infty}{\Rightarrow}$. By lemma 38, b'' is a CPS term. By lemma 39, $v = a$ and therefore v is a CPS atom. It follows that $b''[x \leftarrow v]$ is a CPS term (lemma 37). Moreover, this term is closed because of the usual properties of free variables w.r.t. evaluation and substitution. Using lemma 32 and the coinduction hypothesis, we obtain $b' \stackrel{\mathsf{co}}{\Rightarrow} \lambda x.b''$ and $a \stackrel{\mathsf{co}}{\Rightarrow} v$ and $b''[x \leftarrow v] \stackrel{\mathsf{co}}{\Rightarrow} \Omega$, from which $b \stackrel{\mathsf{co}}{\Rightarrow} \Omega$ follows by rule ($\stackrel{\mathsf{co}}{\Rightarrow}$-app). □

The claimed equivalence result follows as a corollary.

Theorem 41 *Let b be a closed CPS term. We have $\exists v, b \stackrel{\mathsf{co}}{\Rightarrow} v$ if and only if $b \stackrel{\infty}{\Rightarrow}$ or $\exists v, b \Rightarrow v$.*

Proof. Follows from lemmas 32, 34 and 40. □

8 Type soundness proofs

We now turn to using our coinductive evaluation and reduction relations for proving the soundness of type systems. To be more specific, we will use the simply-typed λ-calculus with recursive types as our type system. We obtain recursive types by interpreting the type algebra $\tau ::= \mathtt{int} \mid \tau_1 \to \tau_2$ coinductively, as in [12]. The typing rules are recalled below. Type environments, written E, are finite maps from variables to types.

$$\frac{E(x) = \tau}{E \vdash x : \tau} \qquad E \vdash c : \mathtt{int}$$

$$\frac{E + \{x : \tau'\} \vdash a : \tau}{E \vdash \lambda x.a : \tau' \to \tau} \qquad \frac{E \vdash a_1 : \tau' \to \tau \quad E \vdash a_2 : \tau'}{E \vdash a_1\ a_2 : \tau}$$

Enabling recursive types makes the type system non-normalizing and makes it possible to write interesting programs. In particular, the call-by-value fixpoint operator $Y = \lambda f.\ (\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (\lambda y.\ (x\ x)\ y))$ is well-typed, with types $((\tau \to \tau') \to \tau \to \tau') \to \tau \to \tau'$ for all types τ and τ'. (The self-applications $x\ x$ are well-typed under the assumption $x : \sigma$, where the recursive type σ is defined by the equation $\sigma = \sigma \to \tau \to \tau'$.)

8.1 Type soundness proofs using small-step semantics

Wright and Felleisen [20] introduced a proof technique for showing type soundness that relies on small-step semantics and is standard nowadays. The proof relies on the twin properties of *type preservation* (also called *subject reduction*) and *progress*:

Lemma 42 (Preservation) *If $a \to b$ and $\emptyset \vdash a : \tau$, then $\emptyset \vdash b : \tau$*

Lemma 43 (Progress) *If $\emptyset \vdash a : \tau$, then either $a \in \mathsf{Values}$ or there exists b such that $a \to b$.*

The formal statement of type soundness in Felleisen and Wright's approach is the following:

Theorem 44 (Type soundness, 1) *If $\emptyset \vdash a : \tau$ and $a \xrightarrow{*} b$, then either $b \in \mathsf{Values}$ or b reduces.*

Proof. We first show that $\emptyset \vdash b : \tau$ by induction over $a \xrightarrow{*} b$, using the preservation lemma. We then conclude with the progress lemma. □

The authors that follow this approach then conclude that well-typed closed terms either reduce to a value or reduce infinitely. However, this conclusion is generally neither expressed nor proved formally. In our approach, it is easy to do so:

Theorem 45 (Type soundness, 2) *If $\emptyset \vdash a : \tau$, then either $a \xrightarrow{\infty}$, or there exists v such that $a \xrightarrow{*} v$ and $v \in \mathsf{Values}$.*

Proof (classical). By lemma 10, either $a \xrightarrow{\infty}$ or $\exists b,\ a \xrightarrow{*} b \wedge b \not\to$. The result is obvious in the first case. In the second case, we note that $\emptyset \vdash b : \tau$ as a consequence of the preservation lemma, then use the progress lemma to conclude that $b \in \mathsf{Values}$. □

An alternate, equivalent formulation of this theorem uses the coreduction relation $\xrightarrow{\mathsf{co}*}$.

Theorem 46 (Type soundness, 3) *If $\emptyset \vdash a : \tau$, then there exists v such*

that $a \overset{\mathsf{co}*}{\to} v$ *and* $v \in \mathsf{Values}$.

Proof. Follows from theorem 45 and lemma 8. □

An arguably nicer characterisation of "programs that do not go wrong" is given by the relation $a \overset{\mathsf{safe}}{\to}$ (read: "a reduces safely"), defined coinductively by the following rules:

$$\frac{v \in \mathsf{Values}}{v \overset{\mathsf{safe}}{\to}} \qquad\qquad \frac{a \to b \quad b \overset{\mathsf{safe}}{\to}}{a \overset{\mathsf{safe}}{\to}}$$

These rules are interpreted coinductively so that $a \overset{\mathsf{safe}}{\to}$ holds if a reduces infinitely. We can then state and show type soundness without recourse to classical logic:

Theorem 47 (Type soundness, 4) *If* $\emptyset \vdash a : \tau$*, then* $a \overset{\mathsf{safe}}{\to}$.

Proof. By coinduction. Applying the progress lemma, either $a \in \mathsf{Values}$ and we are done, or $a \to b$ for some b. In the latter case, $\emptyset \vdash b : \tau$ by the preservation property, and the result follows from the coinduction hypothesis. □

8.2 Type soundness proofs using big-step semantics

The standard big-step semantics (defined by the $\Rightarrow$ relation) is awkward for proving type soundness because it does not distinguish between terms that diverge and terms that go wrong: in both cases, there is no value v such that $a \Rightarrow v$. Consequently, the obvious type soundness statement "if $\emptyset \vdash a : \tau$, there exists v such that $a \Rightarrow v$" is false for all type systems that do not guarantee normalization. The best result we can prove, then, is the following big-step equivalent to the preservation lemma:

Lemma 48 (Preservation, big-step style) *If* $a \Rightarrow v$ *and* $\emptyset \vdash a : \tau$*, then* $\emptyset \vdash v : \tau$.

Proof. Easy induction on the derivation of $a \Rightarrow v$, using the fact that typing is stable by substitution: if $\{x : \tau'\} \vdash a : \tau$ and $\emptyset \vdash b : \tau'$, then $\emptyset \vdash a[x \leftarrow b] : \tau$. □

The standard approach for proving type soundness using big-step semantics is to provide inductive inference rules to define a predicate $a \Rightarrow \mathsf{err}$ characterizing terms that go wrong because of a type error, and prove the statement "if $\emptyset \vdash a : \tau$, then it is not the case that $a \Rightarrow \mathsf{err}$" [21]. This approach is not

fully satisfactory for two reasons: (1) extra rules must be provided to define $a \Rightarrow \mathsf{err}$, which increases the size of the semantics; (2) there is a risk that the rules for $a \Rightarrow \mathsf{err}$ are incomplete and miss some cases of "going wrong", in which case the type soundness statement does not guarantee that well-typed terms either evaluate to a value or diverge.

Let us revisit these trade-offs in the light of our characterizations of divergence and coevaluation. We can now formally state what it means for a term to evaluate or to diverge. This leads to the following alternate statement of type soundness:

Theorem 49 (Type soundness, 5) *If $\emptyset \vdash a : \tau$, then either $a \stackrel{\infty}{\Rightarrow}$ or there exists v such that $a \Rightarrow v$.*

By excluded middle, either $\exists v.\ a \Rightarrow v$ or $\forall v,\ \neg(a \Rightarrow v)$. Theorem 49 therefore follows from lemma 50 below, which is a big-step analogue to the progress lemma.

Lemma 50 (Progress, big-step style) *If $\emptyset \vdash a : \tau$ and $\forall v,\ \neg(a \Rightarrow v)$, then $a \stackrel{\infty}{\Rightarrow}$.*

Proof (classical). The proof is by coinduction and case analysis over a. The cases $a = x$, $a = c$ and $a = \lambda x.b$ lead to contradictions: variables have no types in the empty environment; constants and abstractions evaluate to themselves. The interesting case is therefore $a = a_1\ a_2$. By excluded middle, either a_1 evaluates to some value v_1, or not. In the latter case, $a \stackrel{\infty}{\Rightarrow}$ follows from rule ($\stackrel{\infty}{\Rightarrow}$-app-l) and from $a_1 \stackrel{\infty}{\Rightarrow}$, which we obtain by coinduction hypothesis. In the former case, v_1 has a function type $\tau' \to \tau$ by lemma 48, and therefore $v_1 = \lambda x.b$ for some x and b. Moreover, $\{x : \tau'\} \vdash b : \tau$. Using excluded middle again, either a_2 evaluates to some value v_2, or not. In the latter case, $a \stackrel{\infty}{\Rightarrow}$ follows from rule ($\stackrel{\infty}{\Rightarrow}$-app-r) and the coinduction hypothesis. In the former case, $\emptyset \vdash v_2 : \tau'$. Since typing is stable by substitution, $\emptyset \vdash b[x \leftarrow v_2] : \tau$. Using excluded middle for the third time, it must be that $\forall v.\ \neg(b[x \leftarrow v_2] \Rightarrow v)$, otherwise a would evaluate to some value. The result $a \stackrel{\infty}{\Rightarrow}$ then follows from rule ($\stackrel{\infty}{\Rightarrow}$-app-f) and the coinduction hypothesis. □

The proof above is an original alternative to the standard approach of showing $\neg(a \Rightarrow \mathsf{err})$ for all well-typed terms a. From a methodological standpoint, our proof addresses one of the shortcomings of the standard approach, namely the risk of not putting in enough error rules. If we forget some divergence rules, the proof of lemma 50 will, in all likelihood, not go through. Therefore, this novel approach to proving type soundness using big-step semantics appears rather robust with respect to mistakes in the specification of the semantics.

The other methodological shortcoming remains, however: just like the "not goes wrong" approach, our approach requires more evaluation rules than just

those for normal evaluations, namely the rules for divergence. This can easily double the size of the specification of a dynamic semantics, which is a concern for realistic languages where the normal evaluation rules number in dozens.

The coevaluation relation $\overset{\text{co}}{\Rightarrow}$ is attractive for this pragmatic reason, as it has the same number of rules as normal evaluation. Of course, we have seen that $a \overset{\text{co}}{\Rightarrow} v$ is not equivalent to $a \Rightarrow v \vee a \overset{\infty}{\Rightarrow}$, but the example we gave was for a diverging term a that is not typeable and where an early diverging evaluation "hides" a later evaluation that goes wrong. Since type systems ensure that all subterms of a term do not go wrong, we could hope that the following conjecture holds:

Conjecture 1 (Type soundness, 6) *If $\emptyset \vdash a : \tau$, there exists v such that $a \overset{\text{co}}{\Rightarrow} v$.*

We were able to prove this conjecture for some uninteresting but nonetheless non-normalizing type systems, such as simply-typed λ-calculus without recursive types, but with a predefined constant of type `int` $\to$ `int` that diverges when applied. However, the conjecture is false for simply-typed λ-calculus with recursive types, and probably for all type systems with a general fixpoint operator. Andrzej Filinski provided the following counterexample. Consider

$$Y\ F\ 0 \qquad \text{where} \qquad F = \lambda f.\lambda x.\ (\lambda g.\lambda y.\ g\ y)\ (f\ x)$$

or, in more readable ML notation

```
let rec f x = (let g = f x in fun y -> g y) in f 0
```

The term $Y\ F\ 0$ is well-typed with type $\tau \to \tau'$, yet it fails to coevaluate: the only possible value v such that $Y\ F\ 0 \overset{\text{co}}{\Rightarrow} v$ would be an infinite term, $\lambda y.\ (\lambda y.\ (\lambda y.\ \ldots\ y)\ \ y)\ y$.

9 Compiler correctness proofs

We now return to the original motivation of this work: proving that compilers preserve the semantics of source programs (including diverging ones), using big-step semantics. We demonstrate this approach on the compilation of call-by-value λ-calculus down to a simple abstract machine.

Our abstract machine uses closures and environments indexed by de Bruijn indices. It is therefore convenient to reformulate the big-step evaluation predicates in these terms. Variables, written x_n, are now identified by their de Bruijn indices n. Values (which are no longer a subset of terms) and environments are defined as:

Values:	v	$::=$	c	integer values
		$\mid$	$(\lambda a)[e]$	function closures
Environments:	e	$::=$	$\epsilon \mid v.e$	sequences of values

As in section 3, we define three evaluation relations by the inference rules given below.

$e \vdash a \Rightarrow v$	finite evaluations (inductive)
$e \vdash a \stackrel{\infty}{\Rightarrow}$	infinite evaluations (coinductive)
$e \vdash a \stackrel{\mathsf{co}}{\Rightarrow} v$	coevaluations (coinductive)

$$\frac{e = v_1 \ldots v_n \ldots}{e \vdash x_n \Rightarrow v_n} \qquad e \vdash c \Rightarrow c \qquad e \vdash \lambda a \Rightarrow (\lambda a)[e]$$

$$\frac{e \vdash a_1 \Rightarrow (\lambda b)[e'] \quad e \vdash a_2 \Rightarrow v_2 \quad v_2.e' \vdash b \Rightarrow v}{e \vdash a_1\ a_2 \Rightarrow v}$$

$$\frac{e \vdash a_1 \stackrel{\infty}{\Rightarrow}}{e \vdash a_1\ a_2 \stackrel{\infty}{\Rightarrow}} \qquad \frac{e \vdash a_1 \Rightarrow v \quad e \vdash a_2 \stackrel{\infty}{\Rightarrow}}{e \vdash a_1\ a_2 \stackrel{\infty}{\Rightarrow}}$$

$$\frac{e \vdash a_1 \Rightarrow (\lambda b)[e'] \quad e \vdash a_2 \Rightarrow v \quad v.e' \vdash b \stackrel{\infty}{\Rightarrow}}{e \vdash a_1\ a_2 \stackrel{\infty}{\Rightarrow}}$$

$$\frac{e = v_1 \ldots v_n \ldots}{e \vdash x_n \stackrel{\mathsf{co}}{\Rightarrow} v_n} \qquad e \vdash c \stackrel{\mathsf{co}}{\Rightarrow} c \qquad e \vdash \lambda a \stackrel{\mathsf{co}}{\Rightarrow} (\lambda a)[e]$$

$$\frac{e \vdash a_1 \stackrel{\mathsf{co}}{\Rightarrow} (\lambda b)[e'] \quad e \vdash a_2 \stackrel{\mathsf{co}}{\Rightarrow} v_2 \quad v_2.e' \vdash b \stackrel{\mathsf{co}}{\Rightarrow} v}{e \vdash a_1\ a_2 \stackrel{\mathsf{co}}{\Rightarrow} v}$$

We will not formally study these relations, but note that they enjoy the same properties as the environment-less relations studied in section 3.

The abstract machine we use as target of compilation follows the call-by-value strategy and the "eval-apply" model [22]. It is close in spirit to the SECD, CAM, FAM and CEK machines [23,24,25,26]. The machine state has three components: a code sequence, a stack and an environment. The syntax for these components is as follows.

Instructions:	I	::=	$\texttt{Var}(n)$	push the value of variable number n
		|	$\texttt{Const}(c)$	push the constant c
		|	$\texttt{Clos}(C)$	push a closure for code C
		|	$\texttt{App}$	perform a function application
		|	$\texttt{Ret}$	return to calling function
Code:	C	::=	$\epsilon \mid I, C$	instruction sequences
Values:	V	::=	c	constant values
		|	$C[E]$	code closures
Environments:	E	::=	$\epsilon \mid V.E$	
Stacks:	S	::=	ϵ	empty stack
		|	$V.S$	pushing a value
		|	$(C, E).S$	pushing a return frame

The behaviour of the abstract machine is defined as a transition relation $C; S; E \rightarrow C'; S'; E'$ that relates the machine states $(C; S; E)$ and $(C'; S'; E')$ respectively before and after the execution of the first instruction of the code C. The transitions are as follows.

State before transition			State after transition			
Code	Stack	Env.	Code	Stack	Env.	
$\texttt{Var}(n), C$	S	E	C	$V_n.S$	E	if $E = V_1 \ldots V_n \ldots$
$\texttt{Const}(c), C$	S	E	C	$c.S$	E	
$\texttt{Clos}(C'), C$	S	E	C	$C'[E].S$	E	
$\texttt{App}, C$	$V.C'[E'].S$	E	C'	$(C, E).S$	$V.E'$	
$\texttt{Ret}, C$	$V.(C', E').S$	E	C'	$V.S$	E'	

As in section 4, we consider the following closures of the one-step transition relation:

$C; S; E \xrightarrow{*} C'; S'; E'$	zero, one or several transitions (inductive)
$C; S; E \xrightarrow{+} C'; S'; E'$	one or several transitions (inductive)
$C; S; E \xrightarrow{\infty}$	infinitely many transitions (coinductive)
$C; S; E \xrightarrow{\mathsf{co}*} C'; S'; E'$	zero, one, several or infinitely many transitions (coinductive)

The compilation scheme from terms to code is straightforward:

$$\begin{aligned} [\![x_n]\!] &= \mathtt{Var}(n) \\ [\![c]\!] &= \mathtt{Const}(c) \\ [\![\lambda a]\!] &= \mathtt{Clos}([\![a]\!], \mathtt{Ret}) \\ [\![a_1\ a_2]\!] &= [\![a_1]\!], [\![a_2]\!], \mathtt{App} \end{aligned}$$

The intended effect for the code $[\![a]\!]$ is to evaluate the term a and push its value at the top of the machine stack, leaving the rest of the stack and the environment unchanged.

9.3 Proofs of semantic preservation

We expect the compilation to abstract machine code to preserve the behaviour of the source term, in the following general sense. Consider a closed term a and start the abstract machine in the initial state corresponding to a. If a diverges, the machine should perform infinitely many transitions. If a evaluates to the value v, the machine should reach a final state corresponding to v in a finite number of transitions. Here, the initial state corresponding to a is $[\![a]\!]; \epsilon; \epsilon$. The final state corresponding to the result value v is $\epsilon; [\![v]\!].\epsilon; \epsilon$, that is, the code has been entirely consumed and the machine value $[\![v]\!]$ corresponding to the source-level value v is left on top of the stack. The correspondence between source-level values and machine values, as well as between source-level environments and machine environments, is defined by:

$$[\![c]\!] = c \qquad [\![(\lambda a)[e]]\!] = ([\![a]\!], \mathtt{Ret})[[\![e]\!]] \qquad [\![v_1 \dots v_n]\!] = [\![v_1]\!] \dots [\![v_n]\!]$$

Semantic preservation is easy to show for terminating terms a using the big-step semantics. We just need to strengthen the statement of preservation so that it lends itself to induction over the derivation of $e \vdash a \Rightarrow v$.

Theorem 51 *If $e \vdash a \Rightarrow v$, then $([\![a]\!], C);\ S;\ [\![e]\!] \xrightarrow{+} C;\ [\![v]\!].S;\ [\![e]\!]$ for all codes C and stacks S.*

Proof. By induction on the derivation of $e \vdash a \Rightarrow v$. The base cases where a is a variable, a constant or an abstraction are straightforward. The inductive case is $a = a_1\ a_2$ with $e \vdash a_1 \Rightarrow (\lambda b)[e']$ and $e \vdash a_2 \Rightarrow v_2$ and $v_2.e' \vdash b \Rightarrow v$. We build the following sequence of machine transitions:

$$
\begin{aligned}
& (\llbracket a_1 \rrbracket, \llbracket a_2 \rrbracket, \mathtt{App}, C);\ S;\ \llbracket e \rrbracket \\
& \qquad \text{(induction hypothesis applied to the evaluation of } a_1\text{)} \\
\xrightarrow{+}\ & (\llbracket a_2 \rrbracket, \mathtt{App}, C);\ \llbracket (\lambda b)[e'] \rrbracket.S;\ \llbracket e \rrbracket \\
& \qquad \text{(induction hypothesis applied to the evaluation of } a_2\text{)} \\
\xrightarrow{+}\ & (\mathtt{App}, C);\ \llbracket v_2 \rrbracket.\llbracket (\lambda b)[e'] \rrbracket.S;\ \llbracket e \rrbracket \\
& \qquad \text{(}\mathtt{App}\text{ transition, since } \llbracket (\lambda b)[e'] \rrbracket = (\llbracket b \rrbracket, \mathtt{Ret})[\llbracket e' \rrbracket]\text{)} \\
\rightarrow\ & (\llbracket b \rrbracket, \mathtt{Ret});\ (C, \llbracket e \rrbracket).S;\ \llbracket v_2 \rrbracket.\llbracket e' \rrbracket \\
& \qquad \text{(induction hypothesis applied to the evaluation of } b\text{)} \\
\xrightarrow{+}\ & \mathtt{Ret};\ \llbracket v \rrbracket.(C, \llbracket e \rrbracket).S;\ \llbracket v_2 \rrbracket.\llbracket e' \rrbracket \\
& \qquad \text{(}\mathtt{Ret}\text{ transition)} \\
\rightarrow\ & C;\ \llbracket v \rrbracket.S;\ \llbracket e \rrbracket
\end{aligned}
$$

The result follows by transitivity of $\xrightarrow{+}$. □

It is impossible, however, to prove semantic preservation for diverging terms using only the standard big-step semantics, since it does not describe divergence. This led several authors to prove semantic preservation for compilation to abstract machines using small-step semantics with explicit substitutions [27,7]. To this end, they prove a simulation result between machine transitions and source-level reductions: every machine transition corresponds to zero or one source-level reductions. To make the correspondence precise, they need to define a *decompilation* relation that maps intermediate machine states back to source-level terms. However, decompilation relations are difficult to define, especially for optimizing compilation schemes; see [28, section 4.3] for an example.

The coinductive big-step semantics studied in this article provide a simpler way to prove semantic preservation for non-terminating terms. Namely, the following two theorems hold, showing that compilation preserves divergence and coevaluation as characterized by the $\overset{\infty}{\Rightarrow}$ and $\overset{\mathsf{co}}{\Rightarrow}$ predicates.

Theorem 52 *If* $e \vdash a \overset{\infty}{\Rightarrow}$ *, then* $(\llbracket a \rrbracket, C);\ S;\ \llbracket e \rrbracket \overset{\infty}{\rightarrow}$ *for all codes* C *and stacks* S*.*

Theorem 53 *If* $e \vdash a \overset{\mathsf{co}}{\Rightarrow} v$*, then* $(\llbracket a \rrbracket, C);\ S;\ \llbracket e \rrbracket \overset{\mathsf{co}^*}{\rightarrow} C;\ \llbracket v \rrbracket.S;\ \llbracket e \rrbracket$ *for all codes* C *and stacks* S*.*

Both theorems cannot be proved directly by coinduction and case analysis over a. The problem is in the application case $a = a_1\ a_2$, where the code component of the initial machine state is of the form $\llbracket a_1 \rrbracket, \llbracket a_2 \rrbracket, \mathtt{App}, C$. It is

not possible to invoke the coinduction hypothesis to reason over the execution of $[\![a_1]\!]$, because this use of the coinduction hypothesis is not guarded by an inference rule for the $\stackrel{\infty}{\rightarrow}$ relation, or in other terms because no machine instruction is executed before invoking the hypothesis. In the approach to coinduction based on systems of equations presented in section 2.2, the problem manifests itself as a non-guarded equation $x_j = x_{j'}$ when j is the judgment $([\![a_1\ a_2]\!], C);\ S;\ [\![e]\!] \stackrel{\infty}{\rightarrow}$ associated with the state $e \vdash a_1\ a_2 \stackrel{\text{co}}{\Rightarrow} v$, C and S, while j' is the equivalent judgment $([\![a_1]\!], ([\![a_2]\!], \texttt{App}, C));\ S;\ [\![e]\!] \stackrel{\infty}{\rightarrow}$ associated with the state $e \vdash a_1 \stackrel{\text{co}}{\Rightarrow} v$, $([\![a_2]\!], \texttt{App}, C)$ and S.

There are two ways to address this issue. The first is to modify the compilation scheme for applications, in order to insert a "no operation" instruction in front of the generated sequence: $[\![a_1\ a_2]\!] = \texttt{Nop}, [\![a_1]\!], [\![a_2]\!]$. The `Nop` operation has the obvious machine transition $(\texttt{Nop}, C);\ S;\ E \rightarrow C;\ S;\ E$. With this modification, the coinductive proof for lemma 52 performs a `Nop` transition before invoking the coinduction hypothesis to deal with the evaluation of $[\![a_1]\!]$. This makes the coinductive proof properly guarded.

Of course, it is inelegant to pepper the generated code with `Nop` instructions just to make one proof go through. We therefore use an alternate approach where the compilation scheme for applications is unchanged, but we exploit the fact that the number of such recursive calls that do not perform a machine transition is necessarily finite, because our term algebra is finite. More precisely, this number is the left application height $\|a\|$ of the term a being compiled, where $\|a\|$ is defined by

$$\|a_1\ a_2\| = \|a_1\| + 1 \qquad\qquad \|x\| = \|c\| = \|\lambda a\| = 0$$

To prove theorem 52, we follow the approach described by Bertot [29] in his coinductive presentation and proof of Eratosthenes' sieve algorithm. We first define the coinductive relation $\stackrel{\infty}{\underset{n}{\rightarrow}}$ where n is a nonnegative integer:

$$\frac{C; S; E \stackrel{\infty}{\underset{n}{\rightarrow}}}{C; S; E \stackrel{\infty}{\underset{n+1}{\rightarrow}}}\ (\stackrel{\infty}{\underset{n}{\rightarrow}}\text{-sleep})$$

$$\frac{C; S; E \stackrel{+}{\rightarrow} C'; S'; E' \qquad C'; S'; E' \stackrel{\infty}{\underset{n'}{\rightarrow}}}{C; S; E \stackrel{\infty}{\underset{n}{\rightarrow}}}\ (\stackrel{\infty}{\underset{n}{\rightarrow}}\text{-perform})$$

The relation $\stackrel{\infty}{\underset{n}{\rightarrow}}$ is similar to $\stackrel{\infty}{\rightarrow}$, but allows the abstract machine to remain in the same state, not performing any transitions, for at most n steps (rule $\stackrel{\infty}{\underset{n}{\rightarrow}}$-sleep). If n drops to zero, one or several transitions must be performed (rule

$\overset{\infty}{\underset{n}{\rightarrow}}$-perform). In exchange for performing at least one transition, the count n can be reset to any value n', allowing an arbitrary but finite number of non-transitions to be taken afterwards.

A proof by coinduction shows the following variant of theorem 52, using $\overset{\infty}{\underset{n}{\rightarrow}}$ with n equal to the left application height of the term under consideration.

Lemma 54 *If* $e \vdash a \overset{\infty}{\Rightarrow}$ *, then* $([\![a]\!], C);\ S;\ [\![e]\!] \overset{\infty}{\underset{\|a\|}{\rightarrow}}$

Proof. By coinduction and case analysis on the last rule used to derive $e \vdash a \overset{\infty}{\Rightarrow}$. In the first case, $a = a_1\ a_2$ and $e \vdash a_1 \overset{\infty}{\Rightarrow}$. Applying the coinduction hypothesis, we obtain $([\![a_1]\!], [\![a_2]\!], \mathtt{App}, C);\ S;\ [\![e]\!] \overset{\infty}{\underset{\|a_1\|}{\rightarrow}}$ and the result follows by one application of rule ($\overset{\infty}{\underset{n}{\rightarrow}}$-sleep), noticing that $\|a\| = \|a_1\| + 1$.

In the second case, $a = a_1\ a_2$, $e \vdash a_1 \Rightarrow v$ and $e \vdash a_2 \overset{\infty}{\Rightarrow}$. By lemma 51, we obtain $([\![a_1]\!], [\![a_2]\!], \mathtt{App}, C);\ S;\ [\![e]\!] \overset{+}{\rightarrow} ([\![a_2]\!], \mathtt{App}, C);\ [\![v_1]\!].S;\ [\![e]\!]$. Using the coinduction hypothesis, we also have $([\![a_2]\!], \mathtt{App}, C);\ [\![v_1]\!].S;\ [\![e]\!] \overset{\infty}{\underset{\|a_2\|}{\rightarrow}}$. The result follows by rule ($\overset{\infty}{\underset{n}{\rightarrow}}$-perform). The third case of divergence is similar and we omit it. □

We then show the following implication between $\overset{\infty}{\underset{n}{\rightarrow}}$ and $\overset{\infty}{\rightarrow}$.

Lemma 55 *If* $C; S; E \overset{\infty}{\underset{n}{\rightarrow}}$ *, then* $C; S; E \overset{\infty}{\rightarrow}$ *.*

Proof. We first show that $C; S; E \overset{\infty}{\underset{n}{\rightarrow}}$ implies the existence of n', C', S' and E' such that $C; S; E \rightarrow C'; S'; E'$ and $C'; S'; E' \overset{\infty}{\underset{n'}{\rightarrow}}$ by Peano induction over n. The result then follows by coinduction. □

Theorem 52 then follows from lemmas 54 and 55. We omit the proof of theorem 53, which is similar.

10 Related work

There are few instances of coinductive definitions and proofs for big-step semantics in the literature. Cousot and Cousot [8] proposed the coinductive big-step characterization of divergence that we use in this article and studied its applicability for abstract interpretation, as pursued later by Schmidt [30]. This approach was applied to call-by-name λ-calculus by Hughes and Moran [31] and by Crole [32], and to call-by-value λ-calculus by Grall [16].

Following up on [8], Cousot and Cousot recently introduced bi-inductive se-

mantics and applied it to the call-by-value λ-calculus [33]. Bi-inductive semantics are defined in terms of smallest fixed points with respect to a nonstandard ordering. This approach captures both terminating and diverging executions using a common set of inference rules. For instance, in the case of the call-by-value λ-calculus, a single inference rule replaces the two rules ($\Rightarrow$-app) and ($\overset{\infty}{\Rightarrow}$-app-f) of our presentation. It is not entirely clear yet how the bi-inductive approach could be mechanized in a proof assistant. Another difference with the present article is that Cousot and Cousot [33] start from a big-step trace semantics, then systematically derive the other semantics (big-step and small-step) by abstraction: this is an interesting alternative to our approach that separately deals with each semantics.

Gunter and Rémy [34] and Stoughton [35] have the same initial goal as us, namely describe both terminating and diverging computations with big-step semantics, but use increasing sequences of finite, incomplete derivations to do so, instead of infinite derivations. We do not know yet how their approach relates to our $\overset{\infty}{\Rightarrow}$ and $\overset{\mathsf{co}}{\Rightarrow}$ relations.

Milner and Tofte [11] and later Leroy and Rouaix [36] used coinduction in the context of big-step semantics for functional and imperative languages, not to describe diverging evaluations, but to capture safety properties over possibly cyclic memory stores.

Of course, coinductive techniques are routinely used in the context of small-step semantics, especially for the labeled transition systems arising from process calculi. The flavours of coinduction used there, especially proofs by bisimulations, are quite different from the present work. These techniques closely resemble the way coinduction can be used for defining the contextual equivalence in an operational setting [37] and the approximation order in the recursively defined domains involved in denotational semantics [38].

The infinitary λ-calculus [39,40] studies diverging computations from a very different angle: not only the authors use reduction semantics, but their terms are also infinite, and they use topological techniques (metrics, convergence, etc) instead of coinduction.

11 Conclusions

We investigated two coinductive approaches to giving big-step semantics for non-terminating computations. The first, based on [8] and using separate evaluation rules for terminating terms and diverging terms, appears very well-behaved: it corresponds exactly to finite and infinite reduction sequences, and lends itself well to type soundness proofs and to compiler correctness proofs.

The second approach, consisting in a coinductive interpretation of the standard evaluation rules, is less satisfactory: while amenable to compiler correctness proofs as well, it captures only a subset of the diverging computations of interest — and it is not yet clear which subset exactly.

To evaluate the applicability of the coinductive techniques presented here to languages other than small functional languages, we developed coinductive big-step semantics for three low-level imperative languages used in the Compcert verified compiler [41]: the source language Clight (a large subset of the C language) and the two intermediate languages C#minor and Cminor. These semantics characterize non-terminating programs and the traces of input/output events they perform. These semantics were used to mechanically prove that the first four passes of the Compcert compiler preserve the semantics of diverging programs. Some of the proofs use techniques similar to those presented in section 9.3 to combine co-inductive and inductive reasoning. The results of this experiment are encouraging. In particular, the addition of coinductive rules for divergence increases the size of the semantics by 40% only.

Acknowledgments

Andrzej Filinski disproved the conjecture from section 8.2 very shortly after it was stated. We thank Eduardo Bonelli, the anonymous reviewers for the ESOP 2006 conference, the participants of the 22nd meeting of IFIP Working Group 2.8 (Functional Programming), and the anonymous reviewers of this special issue for their feedback.

References

[1] G. Kahn, Natural semantics, in: STACS 87, 4th Annual Symposium on Theoretical Aspects of Computer Science, Vol. 247 of Lecture Notes in Computer Science, Springer, 1987, pp. 22–39.

[2] G. D. Plotkin, A structural approach to operational semantics, Tech. Rep. DAIMI FN-19, Aarhus University (1981).

[3] G. D. Plotkin, A structural approach to operational semantics, Journal of Logic and Algebraic Programming 60-61 (2004) 17–139.

[4] X. Leroy, Formal certification of a compiler back-end, or: programming a compiler with a proof assistant, in: 33rd symposium Principles of Programming Languages, ACM Press, 2006, pp. 42–54.

[5] G. Klein, T. Nipkow, A machine-checked model for a Java-like language, virtual machine and compiler, ACM Transactions on Programming Languages and Systems 28 (4) (2006) 619–695.

[6] M. Strecker, Compiler verification for C0, Tech. rep., Université Paul Sabatier, Toulouse (April 2005).

[7] T. Hardin, L. Maranget, B. Pagano, Functional runtimes within the lambda-sigma calculus, Journal of Functional Programming 8 (2) (1998) 131–176.

[8] P. Cousot, R. Cousot, Inductive definitions, semantics and abstract interpretation, in: 19th symposium Principles of Programming Languages, ACM Press, 1992, pp. 83–94.

[9] Coq development team, The Coq proof assistant, software and documentation available from `http://coq.inria.fr/` (1989–2008).

[10] Y. Bertot, P. Castéran, Interactive Theorem Proving and Program Development – Coq'Art: The Calculus of Inductive Constructions, EATCS Texts in Theoretical Computer Science, Springer, 2004.

[11] R. Milner, M. Tofte, Co-induction in relational semantics, Theoretical Computer Science 87 (1991) 209–220.

[12] V. Gapeyev, M. Levin, B. Pierce, Recursive subtyping revealed, Journal of Functional Programming 12 (6) (2003) 511–548.

[13] P. Aczel, An introduction to inductive definitions, in: J. Barwise (Ed.), Handbook of Mathematical Logic, Vol. 90 of Studies in Logics and the Foundations of Mathematics, North-Holland, 1977, pp. 739–782.

[14] A. Tarski, A lattice-theoretical fixpoint theorem and its applications, Pacific Journal of Mathematics 5 (2) (1955) 285–309.

[15] B. Courcelle, Arbres infinis et systèmes d'équations, R.A.I.R.O. Informatique Théorique 13 (1979) 31–48.

[16] H. Grall, Deux critères de sécurité pour l'exécution de code mobile, Ph.D. thesis, École Nationale des Ponts et Chaussées (Dec. 2003).

[17] L. Simon, A. Mallya, A. Bansal, G. Gupta, Coinductive logic programming, in: Logic Programming, 22nd International Conference, ICLP 2006, Vol. 4079 of Lecture Notes in Computer Science, Springer, 2006, pp. 330–345.

[18] E. Giménez, Codifying guarded definitions with recursive schemes, in: Types for Proofs and Programs. International Workshop TYPES '94, Vol. 996 of Lecture Notes in Computer Science, Springer, 1994, pp. 39–59.

[19] X. Leroy, H. Grall, Coinductive big-step operational semantics – the Coq development, available from `http://gallium.inria.fr/~xleroy/coindsem` (Feb. 2007).

[20] A. K. Wright, M. Felleisen, A syntactic approach to type soundness, Information and Computation 115 (1) (1994) 38–94.

[21] M. Tofte, Operational semantics and polymorphic type inference, PhD thesis CST-52-88, University of Edinburgh (1988).

[22] S. Marlow, S. Peyton Jones, Making a fast curry: push/enter vs. eval/apply for higher-order languages, Journal of Functional Programming 16 (4–5) (2006) 375–414.

[23] P. J. Landin, The mechanical evaluation of expressions, The Computer Journal 6 (1964) 308–320.

[24] G. Cousineau, P.-L. Curien, M. Mauny, The categorical abstract machine, Science of Computer Programming 8 (2) (1987) 173–202.

[25] L. Cardelli, The functional abstract machine, Polymorphism Newsletter 1 (1).

[26] M. Felleisen, D. P. Friedman, Control operators, the SECD machine and the λ-calculus, in: Formal Description of Programming Concepts III, North-Holland, 1986, pp. 131–141.

[27] M. Rittri, Proving the correctness of a virtual machine by a bisimulation, Licentiate thesis, Göteborg University (1988).

[28] B. Grégoire, Compilation des termes de preuves: un (nouveau) mariage entre Coq et OCaml, Ph.D. thesis, University Paris 7 (2003).

[29] Y. Bertot, Filters on coinductive streams, an application to Eratosthenes' sieve, in: Typed Lambda Calculi and Applications (TLCA'05), Vol. 3461 of Lecture Notes in Computer Science, Springer, 2005, pp. 102–115.

[30] D. A. Schmidt, Trace-based abstract interpretation of operational semantics, Lisp and Symbolic Computation 10 (3) (1998) 237–271.

[31] J. Hughes, A. Moran, Making choices lazily, in: Functional Programming Languages and Computer Architecture 1995, ACM Press, 1995, pp. 108–119.

[32] R. L. Crole, Lectures on [Co]Induction and [Co]Algebras, Tech. Rep. 1998/12, Department of Mathematics and Computer Science, University of Leicester (1998).

[33] P. Cousot, R. Cousot, Bi-inductive structural semantics (extended abstract), in: Workshop on Structural Operational Semantics 2007, Vol. 192 (1) of Electronic Notes in Theoretical Computer Science, Elsevier, 2007, pp. 29–44.

[34] C. A. Gunter, D. Rémy, A proof-theoretic assessment of runtime type errors, Research Report 11261-921230-43TM, AT&T Bell Laboratories (1993).

[35] A. Stoughton, An operational semantics framework supporting the incremental construction of derivation trees, in: Second Workshop on Higher-Order Operational Techniques in Semantics (HOOTS II), Vol. 10 of Electronic Notes in Theoretical Computer Science, Elsevier, 1998, pp. 122–133.

[36] X. Leroy, F. Rouaix, Security properties of typed applets, in: J. Vitek, C. Jensen (Eds.), Secure Internet Programming – Security issues for Mobile and Distributed Objects, Vol. 1603 of Lecture Notes in Computer Science, Springer, 1999, pp. 147–182.

[37] A. M. Pitts, Operationally-based theories of program equivalence, in: P. Dybjer, A. M. Pitts (Eds.), Semantics and Logics of Computation, Publications of the Newton Institute, Cambridge University Press, 1997, pp. 241–298.

[38] A. M. Pitts, A co-induction principle for recursively defined domains, Theoretical Computer Science 124 (2) (1994) 195–219.

[39] R. Kennaway, J. W. Klop, M. R. Sleep, F.-J. de Vries, Infinitary lambda calculus., Theoretical Computer Science 175 (1) (1997) 93–125.

[40] A. Berarducci, M. Dezani-Ciancaglini, Infinite lambda-calculus and types, Theoretical Computer Science 212 (1-2) (1999) 29–75.

[41] X. Leroy, The Compcert verified compiler: commented Coq development, Available at `http://compcert.inria.fr/doc/` (Mar. 2008).