Available online at www.sciencedirect.com
SCIENCE DIRECT®
Electronic Notes in Theoretical Computer Science
ELSEVIER Electronic Notes in Theoretical Computer Science 132 (2005) 73–93
www.elsevier.com/locate/entcs

A Proof Calculus for Natural Semantics Based on Greatest Fixed Point Semantics

Sabine Glesner

Institute for Program Structures and Data Organization
University of Karlsruhe, 76128 Karlsruhe, Germany
http://www.info.uni-karlsruhe.de/~glesner

Abstract

Formal semantics of programming languages needs to model the potentially infinite state transition behavior of programs as well as the computation of their final results simultaneously. This requirement is essential in correctness proofs for compilers. We show that a greatest fixed point interpretation of natural semantics is able to model both aspects equally well. Technically, we infer this interpretation of natural semantics based on an easily omprehensible introduction to the dual definition and proof principles of induction and coinduction. Furthermore, we develop a proof calculus based on it and demonstrate its application for two typical problems.

Keywords: Formal semantics, formal compiler correctness, natural semantics, coinductive/greatest fixed point interpretation, proof calculus.

1 The Need for Greatest Fixed Point Semantics

Programming language semantics incorporates two dual aspects: The execution of a program triggers a potentially infinite state transition sequence. If this transition sequence terminates, then it defines the final result of program execution. A formalism for the semantics of programming languages should model both aspects simultaneously. If the execution of a program terminates, then its final result should be defined based on the finite state transition sequence. Moreover, a semantics formalism should specify a more meaningful semantics than just "undefined" for non-terminating programs. This requirement is essential in practical applications. Many programs (e.g. operating systems, data bases, control software in embedded systems or reactive systems) are not intended to terminate while still having a very special semantics.

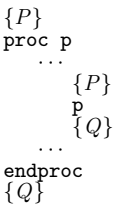
doi:10.1016/j.entcs.2005.02.011

We show that a greatest fixed point interpretation of natural semantics is able to model both aspects simultaneously. This greatest fixed point interpretation gives rise to a proof calculus consisting of inductive and coinductive proof rules. It can be used in the formal reasoning about programming languages. As examples, we consider two applications. The first concerns the correctness proofs of translations, e.g. in compilers. Thereby one needs to prove that the observable behavior of the translated programs is preserved. This is a stronger requirement than just preserving their final results. The second example regards proofs for properties of programming languages, e.g. type safety. They need to consider terminating and non-terminating programs.

Our proof calculus is based on the well-established trend that a combination of algebraic and coalgebraic methods can be used successfully in the specification of and reasoning about programming languages, especially for potentially non-terminating processes. We restate the corresponding proof principles of induction and coinduction in a simple form which is yet powerful enough to model deterministic, possibly infinite computations. We describe the two dual definition and proof principles, in contrast to the common literature utilizing category theory, in a purely set-theoretic and easily comprehensible manner. We show that the state transition behavior of programs must be defined coinductively and that the final result is defined inductively on top of it. While automated theorem provers, e.g. Isabelle [13], have the potential to reason coinductively, the standard practice does not use it. All automated as well as "paper and pencil" proofs based on natural semantics exploit induction and, hence, do only hold for terminating computations. The results of this paper demonstrate that this is not sufficient and can easily be replaced by coinductive reasoning.

The Insufficiency of Induction Proofs

Let us start with a motivation why induction is not the appropriate proof principle for infinite computations.



Consider one of the well-known proof rules of the Hoare calculus [6]. If one wants to prove that a recursive procedure p is correct wrt. a precondition P and a postcondition Q, then one assumes that for all recursive calls of p within the body of p, precondition P and postcondition Q hold. If p always terminates, then this is an induction proof. The recursion depth of the inner calls is always smaller than the recursion depth of p itself. If the procedure p does not terminate, it is no longer a valid induction proof. The state transition sequence in the inner procedure's body is infinitely long as well as the state transition sequence of the outer procedure. Hence, we do not have an induction premise about a strictly smaller

state transition sequence. Both state transition sequences have the same set-theoretic size. Nevertheless, it is still a valid coinductive proof showing that at each procedure entry, the precondition is fulfilled. Thereby we assume that the precondition holds in the initial state and prove that it also holds when entering the inner procedure. In this case, we cannot say anything about the postcondition because the program point at which it should hold is never reached.

The proof for the validity of the Hoare calculus rule in the non-terminating case uses induction to show that no contradiction can be observed. This reasoning is the basis for coinduction. An inductive argument shows that, for all finite prefixes of the potentially infinite state transition sequence, the precondition P is valid at each entry of procedure `p`. Then it concludes that this property (P valid at each entry of `p`) holds also for the infinite state transition sequence. If this were not the case, then there would be a finite prefix not fulfilling P, hence contradicting the result of the induction proof.

The Hoare calculus rule for procedures is essentially an overlay of two rules. The first considers the terminating case with a postcondition. The second models the non-terminating case where the precondition holds at each procedure entry. We will see the same overlay of rules for natural semantics.

2 Natural Semantics

Natural semantics [9] is a deductive method to define the semantics of programs. Axioms and inference rules specify semantic properties wrt. the abstract syntax. The semantics of an abstract syntax tree is defined as a state transition from the initial state into the final state. This state transition is defined compositionally in terms of the state transitions of the direct subtrees of the abstract syntax tree. Consider e.g. the rules for the while-loop:

$$\frac{\mathsf{Eval}(cond,\sigma) = false}{< \textbf{while } cond \textbf{ do } S \textbf{ end}, \sigma > \rightarrow \sigma} \qquad \frac{\mathsf{Eval}(cond,\sigma) = true, \quad < S, \sigma > \rightarrow \sigma', \quad < \textbf{while } cond \textbf{ do } S \textbf{ end}, \sigma' > \rightarrow \sigma''}{< \textbf{while } cond \textbf{ do } S \textbf{ end}, \sigma > \rightarrow \sigma''}$$

These two rules express that the body S of the loop is executed depending on the value of the condition *cond*. If it is executed, then the entire loop is executed recursively again. In the traditional setting, which we revise in this paper, this kind of semantic description is only used for terminating computations. In this case, the second rule says that there exists a state transition from σ to σ'' if the condition *cond* evaluates to true, if the body S is executed by a state transition from σ to σ' and if there is a state transition from σ' to σ'' describing the recursive execution of the loop.

In general, given a production $X_0 ::= X_1 \cdots X_n$ of the abstract syntax, a corresponding inference rule has the following form, whereby $X_{l_k} \in \{X_1, \ldots, X_n\}$, $1 \leq k \leq m$ and $X_{i_j} \in \{X_0, X_1, \ldots, X_n\}$, $1 \leq j \leq r$:

$$\frac{\mathsf{Eval}(X_{l_1}, \sigma_0) = value_1, \ldots, \mathsf{Eval}(X_{l_m}, \sigma_0) = value_m, \quad < X_{i_1}, \sigma_0 > \rightarrow \sigma_1, \cdots, < X_{i_r}, \sigma_{r-1} > \rightarrow \sigma_r}{< X_0, \sigma_0 > \rightarrow \sigma_r}$$

The assumptions of an inference rule consist of two parts, evaluation conditions $\mathsf{Eval}(X_{l_k}, \sigma_0)$ and state transitions of the direct subprograms $< X_{i_j}, \sigma_{j-1} > \rightarrow \sigma_j$. The evaluation conditions decide about the applicability of the rule in a given state σ_0. In the while-loop example, they express the value $\mathsf{Eval}(cond, \sigma_0)$ of the condition $cond$. The state transitions in the assumptions describe the semantics of the subprograms. The entire state transition for the loop is expressed in the conclusion. An axiom is an inference rule with only evaluation conditions in its assumptions but no state transitions, i.e. $r = 0$.

Data structures are needed to define the values of the evaluation conditions and the states reached during program execution. These data structures are typically defined inductively by a term algebra over a fixed set of constructor functions. Additional (defined) functions are specified by equations defining recursively the effect of these functions on the constructor terms.

Natural semantics specifications describe derivation trees. Their root nodes are marked with the program to be executed and with the initial and final state of computation. The successors of the root are marked either with direct subtrees of the program or the program itself (in recursive definitions). Furthermore, the successors are marked with state transitions as defined by the inference rules: The entire state transition from the initial state σ into the final state σ' of the root node is split up into a sequence $\sigma = \sigma_0 \rightarrow \sigma_1 \rightarrow \cdots \rightarrow \sigma_r = \sigma'$ of state transitions. Each individual state transition $\sigma_{i-1} \rightarrow \sigma_i$, $1 \leq i \leq r$, is described by exactly one of the subtrees of the derivation tree. The order on these subtrees is specified implicitly by the linear order of the states $\sigma_0 \rightarrow \sigma_1 \cdots \rightarrow \sigma_r$.

Traditionally, natural semantics specifications are interpreted with finite derivation trees because only then, a unique final state exists. This traditional view corresponds to an inductive or least fixed point interpretation. In this paper, we argue why a greatest fixed point or coinductive interpretation is more appropriate. It also allows for a semantics for non-terminating programs while not changing the usual inductive semantics for terminating programs.

3 Induction and Coinduction

Let $\mathcal{D}$ be an abstract data type, $\mathcal{S}$ some fixed set, $d_l \in \mathcal{D}$ for $l \in \{1, \ldots r_j\}$, $j \in \{1, \ldots, q\}$ and $s_k \in \mathcal{S}$ for $k \in \{1, \ldots, w_j\}$, $j \in \{1, \ldots, q\}$ or $k \in \{1, \ldots, t_i\}$, $i \in \{1, \ldots, p\}$ defined by: $d \in \mathcal{D}$ iff

$$\begin{aligned} d \quad ::= \quad & Base_1(s_1, \ldots s_{t_1}) \mid \cdots \mid Base_p(s_1, \ldots s_{t_p}) \mid \\ & Con_1(d_1, \ldots, d_{r_1}, s_1, \ldots s_{w_1}) \mid \cdots \mid Con_q(d_1, \ldots, d_{r_q}, s_1, \ldots s_{w_q}) \end{aligned}$$

This definition specifies a universe $\mathcal{D}$ of trees whose nodes are marked with one of the base or constructor symbols $Base_i$, $i \in \{1, \ldots, p\}$ or Con_j, $j \in \{1, \ldots, q\}$ and with the corresponding sequence of values $s_1, \ldots s_{t_i}$ or $s_1, \ldots s_{w_j}$. Formally, this set $\mathcal{D}$ of marked trees is defined by two recursive conditions: $d \in \mathcal{D}$ iff:

- If $d = Base_i(s_1, \ldots s_{t_i})$, $1 \leq i \leq p$, then $root(d)$ has no successor nodes. In this case, the marking of $root(d)$ is defined as $mark(root(d)) = (Base_i, s_1, \ldots s_{t_i})$.
- If $d = Con_j(d_1, \ldots, d_{r_j}, s_1, \ldots s_{w_j})$, $1 \leq j \leq q$, then $root(d)$ has r_j direct subtrees $d_1, \ldots, d_{r_j}$ such that $d_1, \ldots, d_{r_j} \in \mathcal{D}$. In this case, the marking of $root(d)$ is defined as $mark(root(d)) = (Con_j, s_1, \ldots s_{w_j})$.

This definition does not only specify trees of finite height but also trees of infinite height. For space reasons, we do not prove that the set $\mathcal{D}$ exists. Such a proof can be found e.g. in [2], showing that the closure ordinal of $\mathcal{D}$ is ω. For sake of readability, as an abbreviation, we write $mark(d)$ for a given tree d instead of $mark(root(d))$, where $root(d)$ denotes the root node of tree d.[1] The universe $\mathcal{D}$ of marked trees induces the complete lattice $(\mathcal{P}(\mathcal{D}), \subseteq)$ where $\mathcal{P}(\mathcal{D})$ denotes the powerset of $\mathcal{D}$ and $\subseteq$ the inclusion relation on sets.

Definition 3.1 [Specification *Spec*] A specification *Spec* defines a unary predicate *Spec* on the universe of an abstract data type $\mathcal{D}$ by stating exactly one equation for each base $Base_i$, $1 \leq i \leq p$, and each constructor Con_j, $1 \leq j \leq q$:

$$\begin{aligned} & Spec(Base_i(s_1, \ldots, s_{t_i})) \equiv true \wedge ok_{Base_i}(s_1, \ldots, s_{t_i}) \\ & Spec(Con_j(d_1, \ldots, d_{r_j}, s_1, \ldots s_{w_j}) \equiv Spec(d_1) \wedge \cdots \wedge Spec(d_{r_j}) \\ & \quad \cdots \wedge ok_{Con_j}(s_1, \ldots, s_{w_j}, mark(d_1), \ldots, mark(d_{r_j})) \end{aligned}$$

Thereby, the predicates ok_{Base_i} and ok_{Con_j}, $1 \leq i \leq p$ and $1 \leq j \leq q$, define restrictions on the allowed combinations of markings of neighbored nodes in the elements of $\mathcal{D}$. The exact definitions of ok_{Base_i} and ok_{Con_j} depend on the concrete specification. E.g. in the context of natural semantics, they are

[1] Note that $mark(d)$ is **not** a recursive function denoting the markings of all nodes in tree d. $mark(d)$ only specifies the marking of the root node of d.

implicitly specified by the axioms and inference rules of the natural semantics, cf. also subsection 4.1 where we state the corresponding details. For now, we only require them to be decidable. The predicate *Spec* defines implicitly a function $spec : \mathcal{P}(\mathcal{D}) \to \mathcal{P}(\mathcal{D})$:

$$spec(X) = \{x \in X \mid \exists i \in \{1,\dots,p\}.x = Base_i(s_1,\dots s_{t_i}) \ \wedge ok_{Base_i}(s_1,\dots,s_{t_i}) \vee$$
$$\exists j \in \{1,\dots,q\}.x = Con_j(d_1,\dots,d_{r_j},s_1,\dots,s_{w_j}) \ \wedge \ d_1 \in X \wedge \dots \wedge d_{r_j} \in X \wedge$$
$$ok_{Con_j}(s_1,\dots,s_{w_j},mark(d_1),\dots,mark(d_{r_j}))\}$$

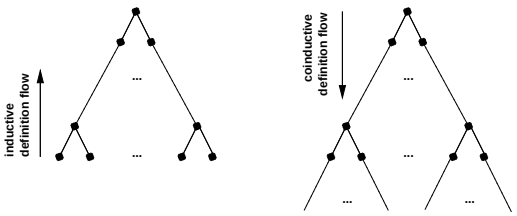
Theorem 3.2 (Monotonicity of *spec*) *The specification function spec: $\mathcal{P}(\mathcal{D}) \to \mathcal{P}(\mathcal{D})$ is monotone on the lattice $(\mathcal{P}(\mathcal{D}),\subseteq)$, i.e., if $X \subseteq Y$ for $X,Y \in \mathcal{P}(\mathcal{D})$, then $spec(X) \subseteq spec(Y)$.*

Proof. By contradiction: Assume there exists $z \in spec(X)$, $z \notin spec(Y)$. Then there exists $x \in X$ such that $spec(\{x\}) = \{z\}$. Since $X \subseteq Y$, it follows that $x \in Y$ and $spec(\{x\}) = \{z\} \subseteq spec(Y)$, contradicting the assumption. □

Tarski's fixed point theorem states that each monotone function f on a complete lattice has a least and a greatest fixed point, denoted by $lfp(f)$ and $gfp(f)$. Hence, we conclude that $lfp(spec)$ and $gfp(spec)$ exist. The least fixed point is called initial algebra, the greatest fixed point final coalgebra.

A specification *Spec* restricts the valid markings of the nodes of the trees in the universe $\mathcal{D}$ of an abstract data type. The least fixed point $lfp(spec)$ is the set of all finite trees whose markings are valid wrt. the specification. (Short outline of a proof: It is obviously a fixed point. Consider a set strictly smaller: Then the "missing element" can always be constructed by a finite construction sequence.) The greatest fixed point $gfp(spec)$ is the set of all trees with finite and infinite height whose markings are valid wrt. the specification. (Short outline of a proof: Each tree in $\mathcal{D}$ not contained in this set has at least two neighbored nodes whose markings are not valid wrt. the markings of its predecessor or successor nodes.)

A priori there is no direction in the specification. It is not determined if a marking is defined in terms of the markings of its successors or of its predecessor. In principle, two definition schemata are possible: In the inductive definition schema, we specify valid markings for the bases. Then we state how they are propagated through the entire tree by defining how the markings of a node are derived from the markings of its child nodes. The reverse direction is also possible and gives us the coinductive definition princi-



ple. Starting at the root node of a tree, we specify how its marking is propagated through the tree. Therefore we define how the marking of a node is derived from the marking of its predecessor. The first principle is structural induction and defines unique markings on finite trees. The second principle works also for infinite trees. Even though a tree might not be finite, the coinductive definition specifies a possibly infinite marking process well-defined at each step.

The inductive definition principle corresponds directly with the inductive proof principle. It states that some predicate Q holds for all elements in the least fixed point $lfp(spec)$. An inductive proof is entirely constructive. Q can only be verified for elements which can be constructed.

There is also a coinductive proof principle which corresponds directly with the coinductive definition principle. It can be used to prove properties of elements in the greatest fixed point. We need these two versions:

Theorem 3.3 (Unary Coinduction Principle) *Let $d \in gfp(spec)$, Q a predicate on the markings of the nodes of d. $Q(mark(k))$ holds for all nodes $k \in d$ if*

- $Q(mark(root(d)))$ *and*
- *if* $\forall j \in \{1, \ldots, q\} \,.\, (d = Con_j(d_1, \ldots, d_{r_j}, s_1, \ldots, s_{w_j}) \Rightarrow (Q(mark(\ Con_j(d_1, \ldots, d_{r_j}, s_1, \ldots, s_{w_j}))) \Rightarrow Q(mark(d_1)) \wedge \cdots \wedge Q(mark(d_{r_j}))))$

The two conditions in theorem 3.3 provide us with a proof principle to verify that all markings in a tree $d \in gfp(spec)$ fulfill a given predicate Q. Therefore we need to prove that Q holds for the marking of the root node of d (first condition) as well as for all nodes which can possibly be reached from this root node (second condition). This is achieved by proving that whenever Q holds for the marking of an inner node, then it also holds for the markings of its direct successor nodes. In contrast to definition 3.1, there are no recursive proof obligations like $Spec(Con_j(\cdots)) \equiv Spec(d_1) \wedge \cdots \wedge Spec(d_{r_j}) \ldots$. Here we only need to prove a non-recursive statement about the finitely many constructors $Con_1, \ldots, Con_j$ of $\mathcal{D}$ and their possible successors. As a consequence of theorem 3.3, we then get a statement about the infinitely many trees in the greatest fixed point $gfp(spec)$ (many of which are of infinite height) and their markings. In practical applications, we verify the two conditions of theorem 3.3 by utilizing the specification $spec$ and its definitions of the predicates ok_{Base_i} for $1 \leq i \leq p$ and ok_{Con_j} for $1 \leq j \leq q$, cf. also section 4 and 5.

Proof. Proof of theorem 3.3. By contradiction: Assume there exists a node $k \in d$ such that $\neg Q(mark(k))$. W.l.o.g. let k be a node with minimal distance to the root node of d such that $\neg Q(mark(k))$. Let pos be the position of this node k, i.e. $k = d\,|_{pos}$. (Each node in a tree can be spec-

ified by a list of navigation numbers denoting the path from the root on which it can be reached.) Since we assume that $Q(mark(root(d)))$ holds, the list pos contains at least one element: $pos = [l \mid pos']$. Since we assume that k is a smallest node such that $\neg Q(mark(k))$, $Q(mark(d\ |_{pos'}))$ follows. But $d\ |_{pos'} = Con_j(d_1, \ldots, d_{r_j}, s_1, \ldots, s_{w_j})$ for some $j \in \{1, \ldots, q\}$ and $d\ |_{pos} \in \{d_1, \ldots, d_{r_j}\}$. From the third assumption in theorem 3.3 we infer that $Q(mark(d_l))$ for all $l \in \{1, \ldots, r_j\}$, in particular $Q(mark(k))$ in contradiction to the assumption $\neg Q(mark(k))$. Hence, $Q(mark(k))$ for all $k \in d$. □

Theorem 3.4 (Binary Coinduction Principle) *Let $d, d' \in gfp(spec)$. $d = d'$ if*

- *for some $i \in \{1, \ldots, p\}$: $d = Base_i(s_1, \ldots, s_{t_i})$ and $d' = Base_i(s_1, \ldots, s_{t_i})$ or if for some $j \in \{1, \ldots, q\}$: $d = Con_j(d_1, \ldots, d_{r_j}, s_1, \ldots, s_{w_j})$ and $d' = Con_j(d'_1, \ldots, d'_{r_j}, s_1, \ldots, s_{w_j})$ and*
- *if for all terms $t_1, t_2 \in gfp(spec)$ and for all $j \in \{1, \ldots, q\}$: if $t_1 = Con_j(d_1, \ldots, d_{r_j}, s_1, \ldots, s_{w_j})$ and $t_2 = Con_j(d'_1, \ldots, d'_{r_j}, s_1, \ldots, s_{w_j})$ implies that for all $l \in \{1, \ldots, r_j\}$: $mark(d_l) = mark(d'_l)$.*

Proof. Analogous to the proof of theorem 3.3: By contradiction: Assume that $d \neq d'$. Then there exists a position $pos = [l \mid pos']$ of minimal length such that $mark(d\ |_{pos}) \neq mark(d'\ |_{pos})$ and $mark(d\ |_{pos'}) = mark(d'\ |_{pos'})$. But then the second condition in theorem 3.4 implies that $mark(d\ |_{pos}) = mark(d'\ |_{pos})$ which is a contradiction to the assumption $d \neq d'$. Hence $d = d'$. □

As theorem 3.3, theorem 3.4 states two **non**-recursive conditions which allow us to reason about recursive, possibly infinite structures. When reasoning about the semantics of programming languages, we use the unary coinduction principle to prove statements about possibly infinite state transition sequences of program executions. Moreover, we use the binary coinduction principle to compare programs by comparing their state transition sequences.

4 Interpretations of Natural Semantics

We start with the observation that each natural semantics defines an abstract data type. Then we show that each natural semantics is a specification in the sense of definition 3.1. We prove that the least fixed point of such a specification describes the execution of all terminating programs while the greatest fixed point defines also a semantics for all non-terminating computations.

4.1 Derivation Trees of Natural Semantics

A big-step semantics defines execution of programs in a top-down fashion: the state transitions of an entire abstract syntax tree are composed from the state transitions of its direct subtrees and, in recursive definitions, also from its own state transitions. It is important to observe that a big-step semantics defines individual state transitions only at the leaves of an abstract syntax tree. For all inner nodes, the inference rules specify how to compose the overall state transition sequence in the conclusion from the state transitions of the assumptions. Hence, we can regard each inference rule as a recursive procedure that is applicable if its evaluation conditions are fulfilled and that calls recursively further axioms or inference rules. The execution of a program defines a possibly infinite derivation tree. Its inner nodes correspond to the application of inference rules and its leaves represent the application of axioms. We define this idea formally:

First we define the markings of the nodes in a derivation tree. Let $\overline{\mathbf{Prog}}$ be all abstract syntax trees. Let $\mathbf{Prog} = \{prog \mid \exists\ prog' \in \overline{\mathbf{Prog}}\ .\ prog = prog' \vee prog \text{ is a subtree of } prog'\}$ be all abstract syntax trees and their subtrees. Let $\mathcal{S}$ be the data structures used in a natural semantics to represent the states (cp. section 2). In a derivation tree, each node is marked with $(P, prog, s, s')$ where P is its base or constructor, $prog \in \mathbf{Prog}$ is a program, $s \in \mathcal{S}$ is the initial state, and $s' \in \mathcal{S}$ the final state.

Let $A_1, \ldots, A_p$ be the axioms and $R_1, \ldots, R_q$ be the inference rules of a natural semantics specification, each belonging uniquely to one production $X_0 ::= X_1 \cdots X_n$ of the abstract syntax and each of the form

$$\frac{\mathsf{Eval}(X_{l_1}, \sigma_0) = value_1, \ldots, \quad \mathsf{Eval}(X_{l_{m_i}}, \sigma_0) = value_{m_i}}{< X_0, \sigma_0 > \rightarrow\ \sigma_1} \qquad \frac{\mathsf{Eval}(X_{l_1}, \sigma_0) = value_1, \ldots, \mathsf{Eval}(X_{l_{m_j}}, \sigma_0) = value_{m_j}, \quad < X_{i_1}, \sigma_0 > \rightarrow\ \sigma_1, \cdots, < X_{i_{r_j}}, \sigma_{r_j - 1} > \rightarrow\ \sigma_{r_j}}{< X_0, \sigma_0 > \rightarrow\ \sigma_{r_j}}$$

The abstract data type $\mathcal{D}$ of derivation trees is defined as follows whereby $prog \in \mathbf{Prog}$ and $s, s' \in \mathcal{S}$: $d \in \mathcal{D}$ iff

$$\begin{aligned} d \quad ::= \quad & A_1(prog, s, s') \mid \cdots \mid A_p(prog, s, s') \mid \\ & R_1(d_1, \ldots, d_{r_1}, prog, s, s') \mid \cdots \mid R_q(d_1, \ldots, d_{r_q}, prog, s, s') \end{aligned}$$

The predicate *Spec* is defined by stating one equation for each axiom and each inference rule. This is the version for A_i, $1 \leq i \leq p$:

$$\begin{aligned} & Spec(A_i(prog, s, s')) \equiv \\ & \quad root(prog) = X_0\ \wedge root(prog\ |_{[1]}) = X_1 \wedge \cdots \wedge root(prog\ |_{[n]}) = X_n\ \wedge \\ & \quad \exists \text{ substitution } \tau\ .\ (\tau(\sigma_0) = s \wedge \tau(\sigma_1) = s' \wedge \end{aligned}$$

$$\mathsf{Eval}(prog\mid_{[l_1]}, \tau(\sigma_0)) = value_1 \wedge \cdots \wedge \mathsf{Eval}(prog\mid_{[l_{m_i}]}, \tau(\sigma_0)) = value_{m_i})$$

The first line specifies that axiom A_i belongs to production $X_0 ::= X_1 \cdots X_n$ and can only be applied to programs of that form. $prog\mid_{[i]}$ denotes the i-th direct subtree of $prog$. The second line describes that the general states σ_0 and σ_1 which may contain variables as placeholders can be mapped to the concrete states s and s' by applying a substitution τ. The last line specifies that the evaluation conditions must be fulfilled in the state $s = \tau(\sigma_0)$.

The version of *Spec* for an inference rule R_j, $1 \leq j \leq q$ needs additional conditions for the recursive correctness requirements. The first line is the recursive constraint requiring that all subtrees fulfill the specification. The last three lines require that each direct subtree is marked either with the same program or a direct subtree of the program. Furthermore, it is specified that the final state of the k-th subtree is the initial state of the $k+1$-st subtree, $1 \leq k \leq j-1$. The remaining requirements are the same as for an axiom:

$$\begin{aligned}
&Spec(R_j(d_1, \ldots, d_{r_j}, prog, s, s')) \equiv Spec(d_1) \wedge \cdots \wedge Spec(d_{r_j}) \wedge \\
&\quad root(prog) = X_0 \ \wedge root(prog\mid_{[1]}) = X_1 \wedge \cdots \wedge root(prog\mid_{[n]}) = X_n \wedge \\
&\quad \exists \text{ substitution } \tau \,.\, (\tau(\sigma_0) = s \wedge \tau(\sigma_{r_j}) = s' \wedge \\
&\quad \mathsf{Eval}(prog\mid_{[l_1]}, \tau(\sigma_0)) = value_1 \wedge \cdots \wedge \mathsf{Eval}(prog\mid_{[l_{m_i}]}, \tau(\sigma_0)) = value_{m_i} \wedge \\
&\quad \forall l \in \{1, \ldots, r_j\} \,.\, (mark(d_l) = (prog', s_1, s_2) \Rightarrow \\
&\quad (i_l = 0 \Rightarrow prog = prog' \wedge i_l > 0 \Rightarrow prog\mid_{[i_l]} = prog') \wedge \\
&\quad root(prog') = X_{i_l} \wedge \tau(\sigma_{l-1}) = s_1 \wedge \tau(\sigma_l) = s_2))
\end{aligned}$$

Spec is a specification wrt. definition 3.1. Hence, there exists a monotone specification function *spec* with least and greatest fixed point, $lfp(spec)$ and $gfp(spec)$, on the set $\mathcal{D}$ of marked derivation trees. If $d \in \mathcal{D}$ is marked with $(P, prog, s, s')$ for $P \in \{A_1, \ldots, A_p, R_1, \ldots, R_q\}$, then we say that d is a derivation tree for *prog* and s and that s is the initial and s' the final state of d.

A priori, there is no direction in the definition of the markings. Nevertheless, in all existing specifications, such a direction exists. For each marking $(P, prog, s, s')$ of a derivation tree, $P \in \{A_1, \ldots, A_p, R_1, \ldots, R_q\}$, the program *prog* and the initial state s are defined coinductively while the final state s' is defined inductively. Even if the execution does not terminate and the final state is not uniquely defined, the state transitions performed during execution are still uniquely determined.

Definition 4.1 A natural semantics specification is deterministic if

- for all $prog \in \mathbf{Prog}$, $s \in \mathcal{S}$, there exists exactly one axiom or inference rule whose evaluation conditions are fulfilled in state s and which is applicable to *prog* (i.e. if the axiom or inference rule belongs to the production $X_0 ::= X_1 \cdots X_n$, then $root(prog) = X_0$ and for $l \in \{1, \ldots, n\}$, $root(prog\mid_{[l]}) = X_l$).
- For each axiom and inference rule, if *prog* and initial state s are known, then

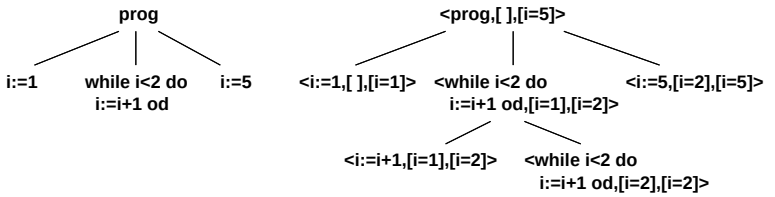


Fig. 1. Semantics of a Terminating Program

all evaluation conditions can be computed by a terminating computation.

- For each axiom, if *prog* and the initial state s are given, then the final state s' can be computed uniquely, also by a terminating computation.

One can also consider the case that there are specifications such that no final state can be computed because, e.g., there might be no applicable rule. Such a case is called a *stuck computation*. To keep the presentation simple, we do not discuss such situations here.

4.2 Classical Inductive Interpretation

The classical interpretation of natural semantics defines semantics only for terminating programs. We give an example for a terminating computation. Then we prove that for all terminating programs, the final state is unique.

Example 4.2 [Terminating Execution] Assume that a state during execution is a list of pairs of variables with their current values. Assume further that the program *prog* as given on the left-hand side in figure 1 is to be executed in state [], i.e., no variable has been assigned a value yet. Then the semantics of the program is represented by the derivation tree d on the right-hand side in figure 1. This example demonstrates the two-level hierarchy of coinductive and inductive structures in program semantics: The program *prog* and the initial state s are defined coinductively. Their definition starts at the root of the derivation tree and is propagated through the tree until its leaves are reached. At the leaves, the coinductive part of semantics, i.e. the state transition behavior, is connected with the inductive part, i.e. the computation of the final state. The definition of the final state is inductive since it starts at the leaves and proceeds along the derivation tree structure up to the root.

Theorem 4.3 *Let Spec be a deterministic natural semantics, spec the corresponding specification function and lfp(spec) its least fixed point on the set $\mathcal{D}$ of marked derivation trees. Let $\mathcal{S}$ be the set of states and* **Prog** *the set of abstract syntax trees or subtrees thereof. For each program prog* $\in$ **Prog**, *$s_0 \in \mathcal{S}$, if the execution of prog starting in s_0 terminates, then there exists*

exactly one derivation tree $d \in \mathcal{D}$ for prog and s_0. The final state of d is uniquely determined.

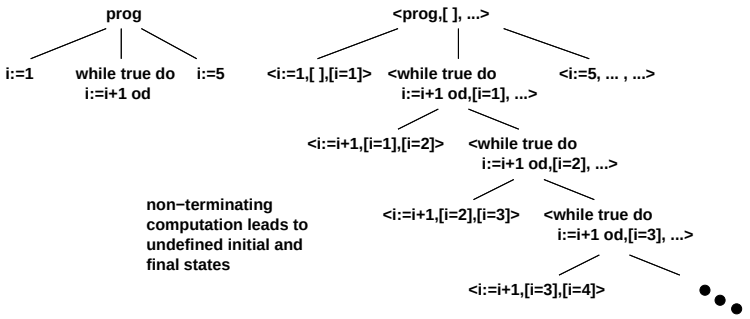
Proof. By Induction on the (finite) structure of d:
Induction Base: If there is an axiom A_i such that its evaluation conditions are fulfilled in s_0 and which is applicable to *prog*, then there exists a unique final state $s' \in \mathcal{S}$ such that $< prog, s_0 > \rightarrow s'$. Because *Spec* is deterministic, no other axiom or inference rule is applicable, hence d is uniquely determined.
Induction Step: Let $R_j, 1 \leq j \leq q$, be the unique inference rule applicable to *prog* whose evaluation conditions are fulfilled. If this rule has r_j assumptions, then the derivation tree d for *prog* and s_0 has r_j direct subtrees. The first subtree is uniquely determined because it is a derivation tree for some program *prog'* and s_0 where either $prog' = prog$ or *prog'* is a direct subtree of *prog*, as specified by R_j. Due to the induction hypothesis, there exists a unique state s_1 which is the final state of the first direct subtree of d, $< X_{l_1}, s_0 > \rightarrow s_1$. s_1 is also the initial state for the second subtree of d. By repeating this reasoning, we conclude that all direct subtrees of d have unique initial and final states. The unique final state of the last subtree of d is also the final state of d. Hence, the derivation tree d for *prog* and s_0 is uniquely determined. □

4.3 Coinductive Interpretation

If a program *prog* does not terminate when started in an initial state s_0, then the derivation tree for *prog* and s_0 has an infinite height. This means that the coinductive and the inductive definition flow of the semantics cannot be connected since there are no leaves. In consequence, there is no unique derivation tree for *prog* and s_0. As an illustration, consider this example:

Example 4.4 [Non-Terminating Execution] As in example 4.2, each state is a list of pairs of variables and their current value. The semantics of the program with the non-terminating while-loop on the left-hand side is represented by the infinite derivation tree on the right-hand side. The annotation "..." means that the value of the respective initial or final state is not uniquely determined. Thus, there are several derivation trees for *prog* and $s_0 = []$, all for which the relation between the markings of parent and child nodes is valid wrt. the specification. Even though not all states are uniquely defined, these derivation trees define a unique infinite state transition sequence, as we prove below.

Definition 4.5 [State Transition Sequence] Let *Spec* be a deterministic natural semantics specification, *spec* the corresponding specification function, *prog* be a program, and s_0 the initial state of computation. Let $d \in gfp(spec)$ be a derivation tree of *prog* and s_0. Then the state transition sequence of d, *prog* and s_0 is defined as follows:



$state\_sequence(R_j(d_1, \ldots, d_{r_j}, prog, s_0, s)) =$
$\quad append([s_0], state\_sequence(d_1), \ldots, state\_sequence(d_n))$
$state\_sequence(A_i(prog, s_0, s)) = [s_0, s]$ where s is the uniquely determined final state (cf. third case in definition 4.1).

Lemma 4.6 *Let Spec be a deterministic natural semantics specification, spec the corresponding specification function, prog be a program, and s_0 the initial state of computation. Let $d \in gfp(spec)$ be a derivation tree of prog and s_0. Then the state transition sequence $state\_sequence(d, prog, s_0)$ of d, prog and s_0 is uniquely defined.*

Proof. If d has finite height, then it follows from theorem 4.3. If d has infinite height and direct subtrees $d_1, \ldots, d_r$, then let d_i, $1 \leq i \leq r$ be the first subtree of infinite height. All subtrees $d_1, \ldots, d_{i-1}$ have finite height and unique initial and final states. d_i has a unique initial state. By using the unary coinduction principle (In theorem 3.3, let Q be the property that the roots of all finite subtrees as well as the first subtree with infinite height have uniquely determined initial states), we conclude that d_i has a unique state transition sequence. Since this sequence is infinite, its concatenation with the state transition sequences of the subtrees $d_{i+1}, \ldots, d_r$ does not have any effect. (The concatenation of an infinite list l with any other list l' is again the list l.) Hence, d has a well-defined state transition sequence. □

When programs do not terminate, then they have in general more than one derivation tree, as discussed in example 4.4. Nevertheless, their state transition sequences are always the same. To prove this, we first define the effective part $eff\_part(d)$ of a derivation tree which includes only those parts which can be reached, if one spends enough time, during computation.

Definition 4.7 [Effective Part of Derivation Tree] The effective part of a derivation tree d, $eff\_part(d)$, is the tree defined as follows: :

$eff\_part(d) = d$ if d has finite height,
$eff\_part(R_j(d_1, \ldots, d_{r_j}, prog, s, s')) =$

$R_j(d_1, \ldots, d_{l-1}, \mathit{eff\_part}(d_l), \mathit{prog}, s, \bot)$
where $l \in \{1, \ldots, r_j\}$, and $d_1, \ldots, d_{l-1}$ have finite height.

Theorem 4.8 (Unique Effective Parts) *Let Spec be a deterministic natural semantics specification, prog a program and s the initial state of program execution. Let spec be the corresponding specification function and $d, d' \in \mathit{gfp}(\mathit{spec})$ derivation trees for prog and s. Then $\mathit{eff\_part}(d) = \mathit{eff\_part}(d')$.*

Proof. If d and d' both have finite height, then its follows directly from theorem 4.3. Hence, assume that d or d' have infinite height and use the binary coinduction principle to prove that $\mathit{eff\_part}(d) = \mathit{eff\_part}(d')$. Therefore we prove the two conditions stated in theorem 3.4 for $\mathit{eff\_part}(d)$ and $\mathit{eff\_part}(d')$.
First condition: The case that $d = A_i(\mathit{prog}, s, s')$ or $d' = A_i(\mathit{prog}, s, s')$, $i \in \{1, \ldots, p\}$, does not exist because then, both d and d' are equal to $A_i(\mathit{prog}, s, s')$ (and finite) because *Spec* is deterministic. Hence, $d = R_j(d_1, \ldots, d_{r_j}, \mathit{prog}, s, s')$ and $d' = R_l(d'_1, \ldots, d'_{r_l}, \mathit{prog}, s, s'')$, $j, l \in \{1, \ldots, q\}$. Because *Spec* is deterministic, it follows that $R_j = R_l$. Hence, wlog., $d' = R_j(d'_1, \ldots, d'_{r_l}, \mathit{prog}, s, s'')$. Hence, we conclude that $\mathit{eff\_part}(d) = R_j(\mathit{eff\_part}(d_1), \ldots, \mathit{eff\_part}(d_{r_l}), \mathit{prog}, s, \bot)$ and $\mathit{eff\_part}(d') = R_j(\mathit{eff\_part}(d'_1), \ldots, \mathit{eff\_part}(d'_{r_l}), \mathit{prog}, s, \bot)$ which shows that the first condition of theorem 3.4 is fulfilled.
Second condition: We need to show that those markings of the direct subtrees of $\mathit{eff\_part}(d)$ and $\mathit{eff\_part}(d')$ which do not denote trees are the same. These markings are the constructor symbols (i.e. the applied inference rules), the program annotations (element of **Prog**) and the initial states in the markings of the direct subtrees of $\mathit{eff\_part}(d)$ and $\mathit{eff\_part}(d')$.
d has at least one infinite subtree d_l, $1 \leq l \leq r_j$. The subtrees $d_1, \ldots, d_{l-1}$ have finite height. d_1 has the same initial state as d'_1, so it follows that $d_1 = d'_1$. (For a proof by contradiction, assume that $d_1 \neq d'_1$. Then assume that there is a first position when traversing d_1 and d'_1 in left-to-right order at which d_1 and d'_1 differ. But this is a contradiction to *Spec* being deterministic). In particular, we conclude that $\mathit{mark}(d_1) = \mathit{mark}(d_1{}')$. With the same reasoning repeated, we prove that $d_k = d'_k$ for $2 \leq k \leq l-1$. The markings of d_l and d'_l do not need to be equal as the final state of a non-terminating computation is not uniquely determined. Nevertheless, the parts of their markings which influence their effective parts are the same: The final state of d_{l-1} and d'_{l-1} are the same so that also the initial states of d_l and d'_l are equal; the programs $\in$ **Prog** in the marking of d_l and d'_l are the same because the same inference rule is applied at d and d' (*Spec* is deterministic); and because *Spec* is deterministic, there is exactly one inference rule R_l which is applicable at d_l and d'_l. Hence, we have $d_l = (R_l(\ldots), \mathit{prog}, s_l, s'_l)$ and $d_l = R_l(\ldots), \mathit{prog}, s_l, s''_l)$. From this, it follows that $\mathit{eff\_part}(d_l) = (R_l(\ldots), \mathit{prog}, s_l, \bot)$ and $\mathit{eff\_part}(d'_l) = (R_l(\ldots), \mathit{prog}, s_l, \bot)$ which completes the proof of the second condition of the-

orem 3.4. Hence, we conclude that $\mathit{eff\_part}(d) = \mathit{eff\_part}(d')$. □

Corollary 4.9 (Unique State Transition Sequence) *Let Spec be a deterministic natural semantics, prog a program and s_0 the initial state of program execution. Let spec be the corresponding specification function and $d, d' \in \mathit{gfp}(\mathit{spec})$ be derivation trees for prog and s_0. Then $\mathit{state\_sequence}(d, \mathit{prog}, s_0) = \mathit{state\_sequence}(d', \mathit{prog}, s_0)$.*

Proof. This follows directly from theorem 4.8 and the construction used in the proof of lemma 4.6. □

Definition 4.10 [Semantics of a Program] Let *Spec* be a natural semantics, *spec* the corresponding specification function, and *prog* be a program. The semantics $\mathbf{Sem}(\mathit{prog})$ of *prog* is defined as the set of all derivation trees in $\mathit{gfp}(\mathit{spec})$ whose root is marked with *prog*:

$$\mathbf{Sem}(\mathit{prog}) = \{d \in \mathit{gfp}(\mathit{spec}) \mid \exists s, s' \in \mathcal{S}, P \in \{A_1, \ldots, A_p, R_1, \ldots, R_q\} \; . \; \mathit{mark}(\mathit{root}(d)) = (P, \mathit{prog}, s, s')\}$$

The semantics of *prog* for the initial state s_0 is the set

$$\mathbf{Sem}(\mathit{prog}, s_0) = \{d \in \mathbf{Sem}(\mathit{prog}) \mid \exists s' \in \mathcal{S}, P \in \{A_1, \ldots, A_p, R_1, \ldots, R_q\} \; . \; \mathit{mark}(\mathit{root}(d)) = (P, \mathit{prog}, s_0, s')\}$$

The set $\mathbf{Sem}(\mathit{prog}, s_0)$ might contain more than one derivation tree. In this case, the computation does not terminate. Subtrees of the derivation tree coming after (wrt. to a depth-first left-to-right order) the non-terminating subtree do not contribute to the infinite state transition sequence since they will never be reached. Nevertheless, the effective parts of all derivation trees in $\mathbf{Sem}(\mathit{prog}, s_0)$ are the same and contain exactly those parts of the derivation trees which contribute to the state transition sequence of the program.

5 Applications of the Proof Calculus

Natural semantics, if interpreted coinductively, combines both aspects of programming language semantics in a very elegant and theoretically simple way. It defines a unique effective part for each program and each initial state. For all terminating executions, it defines also a unique final state. For all non-terminating executions, it describes uniquely the infinite state transition sequence of program execution. In this section, we show how the unary and binary coinduction principles can be applied for programming languages.

Compiler Correctness A correct compiler should preserve the observable behavior of the translated programs. This requirement is essential. In many practical applications, programs do not terminate and are not even intended

to terminate (e.g. data bases, operating systems, software in embedded systems, reactive systems in general). If one wants to verify that software for such systems is translated correctly, the proof cannot be done by induction. The corresponding derivation trees and state transition sequences are not finite. Instead one needs a coinductive proof that the observable behavior, i.e. the state transition sequence is the same in the original and the translated program. The basis for coinductive reasoning is greatest fixed point semantics.

Example 5.1 [Verification of an Optimization] Consider the non-terminating program from example 4.4. An optimizing compiler might recognize that the while-loop does not terminate. Since the compiler is required to preserve the observable behavior, it cannot modify the while-loop. Nevertheless, the assignment `i:=5` will never be reached during any execution and can be eliminated. If the inference rules for the while-loop (cp. section 2) are interpreted inductively with the least fixed point, then such a transformation cannot be verified as being semantics-preserving. In the greatest fixed point interpretation, we can do a coinductive proof showing that the while-loop does not terminate and that the state transition sequences in the original and in the optimized program are the same. Therefore we need to use the binary coinduction principle to prove that the effective parts of the derivation trees d and d' of the original and the optimized program are the same. The reasoning is completely analogous as in the proof of theorem 4.8. First we do a case distinction if d and d' are both finite (which is trivial because of theorem 4.3). For the non-trivial case, at least d or d' of infinite height, we use the binary coinduction principle to prove that the effective parts of d and d' are the same. Therefore we need to verify the two conditions of theorem 3.4: According to the first condition, we need to show that the applied inference rules at the root node of the derivation trees, the initial states and the program are the same. This holds trivially (assuming that the natural semantics is deterministic). According to the second condition we need to show that the markings of the direct subtrees of d and d' which contribute to their effective parts are marked with the same initial state, the same program and the same applied inference rule or axiom. Therefore we do a case distinction (either d or d' is finite or both are infinite), the first case is trivial because of theorem 4.3. For the second case, we first consider the first l finite subtrees of d and show that d' has the same first l finite subtrees (by induction). Then we finish the proof by showing that the first subtrees of infinite height in d and d' contribute with the same applied inference rule, the same initial state and the same annotated program to the effective parts of d and d' which completes the proof of the two conditions in the binary coinduction principle. Therefore we conclude that the effective parts of the original loop-program and the optimized loop-program

are the same and, hence, their state transition sequences are also the same.

Properties of Programming Languages Assume that the semantics of a programming language is defined by a natural semantics. Assume also that we want to prove a certain property Q for the states reached during program execution. Such a property could e.g. be the type-safety of the language. To prove that Q holds in all states reached during execution, we need to show the following: Let p be an arbitrary program, let d be a derivation tree for p for an arbitrary but fixed initial state s, and consider the effective part $\mathit{eff\_part}(d)$ of d. Then we need to verify the two conditions of the unary coinduction principle (theorem 3.3) for $\mathit{eff\_part}(d)$, i.e. verify that Q holds for the states in all markings in the effective part of d (by assuming that Q holds trivially for the state $\perp$). The first condition requires us to verify that Q holds in arbitrary initial states. To verify the second condition, we need an interleaved inductive and coinductive reasoning because the initial states are defined coinductively while the final states (which are also initial states in neighboured derivation subtrees) are specified inductively depending on the initial states. Consider the first l subtrees of d which have finite height. To prove that Q holds for all their initial and their final states requires an inductive proof along the axioms and inference rules of the specification. If d has only l subtrees, then we are done at this point. Otherwise, consider the $l+1$-st subtree which is the last subtree contributing to the effective part of d. Its initial state is the uniquely determined final state of d_l, for which we have already shown that Q holds. Since $\mathit{eff\_part}(d)$ has no more subtrees, we have verified that the second condition of theorem 3.3 holds which completes the proof.

6 Related Work

The results of this paper contradict the common understanding that natural semantics can only describe terminating computations (cf. [10,14] or any other textbook or lecture notes of your choice) while structural operational semantics, also called small-step semantics, is additionally suited to describe non-terminating programs. Rather it is the common least fixed point interpretation of natural semantics that defines semantics only for terminating programs. Usually a greatest fixed point semantics is assumed implicitly for structural operational semantics but without drawing the conclusion that coinductive proof rules are necessary. For both specification formalisms, both interpretations are possible. A least fixed point interpretation of structural operational semantics defines semantics only for terminating programs by assigning them a finite state transition sequence and an "undefined" to all non-terminating programs. The only related research attempting to widen the interpretations of natural semantics is described in [7]. It defines a coinductive interpretation

of natural semantics by translating it into a small-step format. Induction is used to reason about the thereby defined finite and infinite state transition sequences. This is only a half-hearted approach as it does not separate between the coinductive character of the state transitions and the inductive nature of the final result defined on top of them. We want to emphasize that induction is not the appropriate proof method to reason about the state transition behavior of programs, see also our explanations about induction and coinduction for lists at the end of this section.

This insight has severe consequences as it reveals that most equivalence proofs for programs based on structural induction do hold only if the programs terminate. In particular, this holds for the research efforts in proving the static type safety of Java [3,16,18,17]. All proofs are based on inductive arguments and, hence, do only hold for terminating programs. Therefore, one would assume that the machine-checking approach needs extra assumptions when applying the inductive proof principle. Indeed, the inductive proofs did not work without further assumptions: In the machine-checking approach documented in detail in [17], the maximal recursive depth of evaluation is restricted to a finite number, cp. paragraphs 5.3.2 and 5.7.2 of [17]. The same assumption has been applied in the mechanical verification of the correctness of a compiling specification [4] using the PVS system [12], cp. section 4 of [4].

We have based our proof calculus on a simple exploitation of finite and infinite abstract data types. The set-theoretic basis for this straightforward development can be found e.g. in [2] which shows that coinductive interpretations of rule systems capture the behavior of finite and infinite state transition sequences. Most of the existing literature on algebras and coalgebras and their corresponding definition and proof principles induction and coinduction chooses a categorical setting, cp. e.g. [1,8]. Nevertheless, in most situations one needs only polynomial functors going from the category of sets and functions to itself. The theory of algebras and coalgebras for polynomial functors can be stated in set theory. Then, the difference between an initial algebra and a final coalgebra is reduced to the distinction between finite and infinite data structures, i.e. least and greatest fixed points. We believe that this set-theoretical setting allows for a more intuitive understanding and in turn for better applicability in practical situations. Our notation in section 3 is based on the exposition in appendix B titled “Induction and Coinduction” in [11]. While the explanations therein give a good understanding of least and greatest fixed points of specifications, they do not state proof rules like the unary or binary coinduction principle. Rather they state a proof rule that the elements of each post fixed point fulfill the specification. In the context of functional programming languages, coinduction and bisimulation (which cor-

responds to the binary coinduction principle stated in section 3) have been used to deal with non-terminating computations, cf. e.g. [5]. The research documented in [15] investigates how compiler optimizations can be verified. His approach uses natural semantics and is restricted to terminating computations only but is able to abstract from given state transition sequences so that optimizations do not need to yield programs showing the exactly same state transition sequence. It would be interesting in future work to investigate if the coinductive version of natural semantics presented here can be combined with these methods.

Finally a remark about lists as degenerated trees: When we reason about state transition sequences, we start at the root of the lists and infer properties for a child node from its parent node. This is coinduction. It is different from induction where we start at the leaf of the list and construct (finite) lists by using already constructed smaller lists. The inductive view is used for defining results of computations. Thereby we assume that the list-degenerated state transition tree has a leaf as base case. Since these dual definition and proof principles look so similar for lists, there is often no clear distinction between them. The difference is in the practically probably not very important, yet existing distinction between state transition sequences of unbounded length and sequences of infinite length. In the first case, one can deal with all finite sequences, no matter how large they are. In the second case, one can also deal with infinite sequences. To capture also the infinite sequences, one needs to use coinduction. Induction can only deal with finite state transition sequences of unbounded length but is not appropriate for infinite state transition sequences. This is strikingly documented in the machine-checking approaches discussed above which need extra assumptions restricting proofs to terminating computations only.

7 Conclusions

Our investigations are based on the observation that programming language semantics has two dual aspects, the state transition behavior and the computation of the final result. In consequence, programming language semantics needs to be defined by a two-layer hierarchy: First the potentially infinite state transition behavior is defined coinductively. On top of this coinductive structure, an inductive definition specifies the final result of computation. It is unique only if the state transition sequence is finite. This connection between the coinductive and the inductive structure of program semantics seems to be essential and not only a characteristics of natural semantics, whereupon its greatest fixed point interpretation demonstrates it particularly clearly. In this

sense, we have established natural semantics as a well-balanced formalism for the semantics of programming languages as it models both aspects sufficiently and evenly. Axiomatic semantics, in particular the Hoare calculus [6] is an equally balanced formalism. It defines the preconditions coinductively and the postconditions inductively. This implies that the postconditions do only hold if the execution terminates. Especially the rule for recursive procedures demonstrates this interleaved coinductive/inductive reasoning (cp. section 1).

We have based our proof calculus on a purely set-theoretic and simple introduction to induction and coinduction and their respective definition and proof principles. Based on the definition of an abstract data type, we have introduced specifications which might put further restrictions on the valid structures. In a least fixed point setting, we consider only finite data structures. In a greatest fixed point setting, also infinite data structures are included. Induction proves that only correct structures can be constructed. Coinduction proves that no contradiction can be observed. We think that such an easily comprehensible description helps in bridging the gap between theoretical developments in the field of formal semantics of programming languages on one side and practical applications in reasoning about properties of programs and programming languages on the other side. The machine-checking approaches and their proof restrictions discussed in section 6 clearly indicate the necessity to close this gap. The two example applications in section 5 show that the coinductive proof rules can be used in typical practical situations.

In future work we will investigate how natural semantics deals with nondeterministic and parallel computations and how greatest fixed point interpretations and corresponding proof rules can be established for these extensions.

Acknowledgement

The author would like to thank the anonymous reviewers for their helpful comments. This work was supported by a research grant within the "Eliteförderprogramm für Postdoktoranden der Landesstiftung Baden-Württemberg".

References

[1] Roland Backhouse, Roy Crole, and Jeremy Gibbons, editors. *Algebraic and Coalgebraic Methods in the Mathematics of Program Construction.* Springer, LNCS 2297, 2002. International Summer School and Workshop, Oxford, UK, April 10-14, 2000, Revised Lectures.

[2] Patrick Cousot and Radhia Cousot. Inductive Definitions, Semantics and Abstract Interpretation. In *Proc. 19th ACM Symposium on Principles of Programming Languages*, pages 84–94, 1992. ACM.

[3] Sophia Drossopoulou and Susan Eisenbach. Describing the Semantics of Java and Proving Type Soundness. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, page 41 ff. Springer Verlag, LNCS 1523, 1999.

[4] Axel Dold and Vincent Vialard. A Mechanically Verified Compiling Specification for a Lisp Compiler. In *Proceedings of the 21st Conference on Software Technology and Theoretical Computer Science (FSTTCS 2001)*, pages 144–155, 2001. Springer Verlag, LNCS 2245.

[5] Andrew D. Gordon. A Tutorial on Co-Induction and Functional Programming. In *Proceedings of the 1994 Glasgow Workshop on Functional Programming, September 1994*, Ayr, Scotland, 1995. Springer Workshops in Computing.

[6] C.A.R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12(10):576 – 580, October 1969.

[7] Husain Ibraheem and David A. Schmidt. Adapting Big-Step Semantics to Small-Step Style: Coinductive Interpretations and "Higher-Order" Derivations. In *Proceedings 2nd Workshop on Higher-Order Techniques in Operational Semantics (HOOTS2)*, Stanford, 1998. Elsevier ENTS.

[8] Bart Jacobs and Jan Rutten. A Tutorial on (Co)Algebras and (Co)Induction. *EATCS Bulletin*, 67:222–259, 1997.

[9] Gilles Kahn. Natural Semantics. In *Proceedings of the 4th Annual Symposium on Theoretical Aspects of Computer Science (STACS'87)*, pages 22–39, Passau, Germany, February 1987. Springer, LNCS 247.

[10] Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications: A Formal Introduction*. Published in 1992 by John Wiley & Sons, revised edition available at http://www.daimi.au.dk/~ hrn, 1999.

[11] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer, 1999.

[12] S. Owre, J.M. Rushby, and N. Shankar. PVS: A Prototype Verification System. In *Proc. 11th Int'l Conference on Automated Deduction CADE*, 1992. Springer-Verlag, Lecture Notes in Artificial Intelligence, vol. 607.

[13] Lawrence C. Paulson. A Fixedpoint Approach to (Co)Inductive and (Co)Datatype Definitions, 2002. Computer Laboratory, University of Cambridge, England.

[14] David Schmidt. Programming Language Semantics. In *CRC Handbook of Computer Science*, Boca Raton, USA, 1996. CRC Press.

[15] Igor A. Siveroni. *Correctness of Analysis-based Program Transformations of Functional Programming Languages*. PhD thesis, College of Computer Science, Northeastern University, Boston, USA, 2002.

[16] Don Syme. Proving Java Type Soundness. In *Formal Syntax and Semantics of Java*, page 83 ff. Springer Verlag, LNCS 1523, 1999.

[17] David von Oheimb. *Analyzing Java in Isabelle/HOL*. PhD thesis, Technische Universität München, Germany, 2001.

[18] David von Oheimb and Tobias Nipkow. Java_{light} is Type-Safe – Definitely. In *Proceedings of the 25th ACM Symposium on the Principles of Programming Languages*, January 19-21 1998. ACM Press.