Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 176 (2007) 61–77

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Coinductive Verification of Program Optimizations Using Similarity Relations

Sabine Glesner   Johannes Leitner   Jan Olaf Blech

Institute for Software Engineering and Theoretical Computer Science, FR 5-6, Technical University of Berlin, 10587 Berlin, Germany
Email: {glesner|leitner|blech}@cs.tu-berlin.de
URL: *http://pes.cs.tu-berlin.de/*

Abstract

Formal verification methods have gained increased importance due to their ability to guarantee system correctness and improve reliability. Nevertheless, the question how proofs are to be formalized in theorem provers is far from being trivial, yet very important as one needs to spend much more time on verification if the formalization was not cleverly chosen. In this paper, we develop and compare two different possibilities to express coinductive proofs in the theorem prover Isabelle/HOL. Coinduction is a proof method that allows for the verification of properties of also non-terminating state-transition systems. Since coinduction is not as widely used as other proof techniques as e.g. induction, there are much fewer "recipes" available how to formalize corresponding proofs and there are also fewer proof strategies implemented in theorem provers for coinduction. In this paper, we investigate formalizations for coinductive proofs of properties on state transition sequences. In particular, we compare two different possibilities for their formalization and show their equivalence. The first of these two formalizations captures the mathematical intuition, while the second can be used more easily in a theorem prover. We have formally verified the equivalence of these criteria in Isabelle/HOL, thus establishing a coalgebraic verification framework. To demonstrate that our verification framework is suitable for the verification of compiler optimizations, we have introduced three different, rather simple transformations that capture typical problems in the verification of optimizing compilers, even for non-terminating source programs.

Keywords: coinduction, operational semantics, compiler verification, theorem prover, Isabelle/HOL

1 Introduction

Formal verification within interactive or automated theorem provers has become more and more important during the last ten years due to several reasons: First of all, machine proofs guarantee that no special cases have been overlooked and, hence, that the verified properties will indeed hold under all specified circumstances. Secondly, mechanized theorem provers allow for managing even large correctness proofs for large systems that cannot be managed by humans without machine help due to sheer complexity. And last but not least, the efficiency and user-friendliness of theorem provers has improved so much over the last years that they can be used in real-life verification problems. Nevertheless, there are still many unsolved problems. Especially the question how proofs are to be formalized in a theorem

doi:10.1016/j.entcs.2006.02.037

prover is not trivial at all. An unclever formalization might lead to clumsy proofs, thus complicating the proof unnecessarily.

In this paper, we investigate correctness proofs for transformations, i.e. proofs that show that transformations preserve the semantics of the systems modified by them. In particular, we address non-refining transformations that do not preserve the syntactical structure of programs. A transformation is called *refining* if it does not alter the structure of programs but only specifies in more detail how the individual computations are to be executed. Non-refining transformations are important in optimizing compilers because they allow for code transformations that do not adhere to the original program structure. Correctness proofs for non-refining transformations of entire programs pose the problem that they cannot be composed directly from the correctness proofs of their parts. Moreover, we address the problem of verifying transformations of non-terminating programs. For these purposes, we need a notion of correctness that captures the state transition behavior of programs.

As application examples for our methodology, we consider optimizing compilers that typically do need to transform programs by non-refining optimizations in order to achieve the best possible speed-up. The area of compiler verification has been a major area of research during the past decade. Nevertheless, most of this research focuses on rather simple, refining transformations which preserve the structure of programs as well as on terminating programs. Especially for compilers, the restriction to terminating programs is not adequate because many non-terminating programs (e.g. operating systems, data bases, software in reactive systems) exist which need to be compiled correctly such that their state transition behavior is preserved.

Our approach is based on the observation that each program defines a state transition system and can be considered as an element of a suitable final coalgebra. We present a framework for defining programming language semantics by assigning each program such an element of a coalgebra, i.e. a function that transforms input states into successor states. For each initial state and program, this semantics specifies an element of such a final coalgebra, namely a coinductively defined (lazy) list of a potentially infinite number of program execution states. We also present our formalization of this coalgebraic semantics in Isabelle/HOL.

Furthermore, we show that this notion of programming language semantics is sufficiently powerful for the verification of also non-refining program transformations. In general, we consider two programs as being semantically equivalent iff their semantics, i.e. their corresponding elements of a suitable coalgebra are the same. Nevertheless, in many cases, one wants to regard two programs as semantically equivalent even though their state transition sequences (for an arbitrary but fixed initial state) are not completely identical but only similar. To be able to also capture these cases, we define two *similarity relations* on state transition sequences: *abstractions* and *collapsings*. For the latter, we state two different definitions and prove their equivalence within the theorem prover Isabelle/HOL. The first definition captures very well the mathematical intuition while the second definition is much better suited for the use within Isabelle/HOL. This is a typical example for the

$i := 1;$ **while** *true* **do** $i := i + 1$ **od**; $i := 5$	$i := 1;$ **while** *true* **do** $i := i + 1$ **od**

Fig. 1. Eliminating Unreachable Code

experience that the choice of formalization in a theorem prover is of utmost importance. With our equivalence proof, we bridge the gap between an intuitive notion and one for efficient use in a theorem prover, thus simplifying the construction of mechanized proofs considerably.

We have formalized our framework for the coinductive definition of programming language semantics together with the equivalence proof for the two definitions of collapsings within the theorem prover Isabelle/HOL [15]. Moreover, also within Isabelle/HOL, we have instantiated our framework with a simple imperative programming language together with example proofs for program equivalence for typical cases. With these example proofs, we show that our framework is flexible enough for the verification of a variety of optimizing program transformations in compilers that are not refining.

This paper is structured as follows: In Section 2, as a motivation, we discuss typical examples of compiler transformations and the problems arising in their verification. We also indicate that a coalgebraic definition of program semantics together with similarity relations as introduced in this paper is sufficiently powerful to verify these transformations. In Section 3, we give a short introduction to the theory of (co)algebras and (co)induction tailored to our needs. In Section 4, we define semantics of imperative programming languages coalgebraically by assigning each program and each initial state an element of a suitable final coalgebra. The notion of similarity relations is given in Section 5. We demonstrate their application in the verification of program transformations in Section 6. Finally we discuss related work in Section 7 and conclude in Section 8. Our Isabelle formalization is presented along the way throughout this paper.

2 Motivation: Verifying Compiler Transformations

As simple, nevertheless typical examples for the situations arising in the verification of compiler transformations, let us consider three programs and optimizations thereof. The first example, cf. Figure 1, deals with a non-terminating program that contains unreachable code (after the while-loop). Since the while loop does not terminate, the last assignment $i := 5$ will never be reached and could be eliminated. To prove that the program pruned in this way behaves as the original one, we need to coinductively show that the semantics of both of them are in a bisimulation relation which implies that they are equal (cf. Section 6). Induction as proof principle would not suffice as it allows us only to prove statements over finite state transition sequences, cf. also our detailed discussion of the insufficiency of induction in Section 7.

The second example concerns the shifting of loop invariant code out of loop bodies, cf. Figure 2. Since the assignment $l := 0$ does not depend on any value computed or modified in the loop body, this assignment can be executed already

```
$i := 0$;
while $i < 1000$ do
    $l := 0$;
    $a[i] := i$;
    $i := i + 1$  od
```

```
$i := 0$;
$l := 0$;
while $i < 1000$ do
    $a[i] := i$;
    $i := i + 1$  od
```

Fig. 2. Shifting of Loop-Invariant Code

```
$i := 0$;  $s := 0$;
while $i <= N$ do
    $s := s + i$;  $i := i + 1$
od;
$Sum := s$
```

```
$Sum := N * (N + 1)/2$
```

Fig. 3. Two Different Programs Both Computing the Gauss Sum

before entering the loop, thus computing it only once instead of in every iteration of the loop. The verification of this transformation is not as straight-forward as in the first example because the state transition sequences are not completely equal but might differ in the number of repetitions of the same state. We clearly do not want to distinguish between the two sequences but instead want to be able to collapse any finite number of identical states. This idea is captured in the notion of *collapsings*, cf. Section 5.

Finally, consider the two programs in Figure 3 (with N being a non-negative integer), both computing the Gauss sum. Their semantics, i.e. their state transition sequences, are different because the first program contains computations which involve the variables i and s while the second does not need any auxiliary variables. Hence, from a strict semantic point of view, these two programs cannot be semantically identical. Nevertheless, if one forgets about variables i and s, i.e., if one projects each state on only that part one is interested in, the resulting state transition sequences are similar and can be collapsed into the same. In Section 5, in particular in Subsection 5.2, we define this idea formally by introducing the notion of being similar modulo an abstraction function.

3 (Co)Algebras and (Co)Induction

In recent years, coalgebraic methods, in particular coinduction, have gained increased interest and importance in the specification of and reasoning about state-based systems [10]. In Subsection 3.1 we summarize the most important concepts in this area. In Subsection 3.2 we show how the coinductive definition and proof principle can be used in the theorem prover Isabelle/HOL [15].

3.1 Coalgebras and the Coinductive Proof Principle

Algebras and coalgebras are defined with respect to functors. Given a functor T and a set X, a T-algebra is a set supplied with a T-structure, i.e. is defined as a function $a : T(X) \to X$. For example, the structure of the natural numbers is defined as an T-algebra $[0, S] : 1 + \mathbb{N} \to \mathbb{N}$ for the functor $T(X) = 1 + X$. *Initial T-algebras* are characterized by the fact that there exists a unique homomorphism f from the initial T-algebra into any other T-algebra. Initial T-algebras are the least fixed point of the functor T.

Dually, T-coalgebras are defined as functions $c : X \to T(X)$. If one thinks of the elements in X as being states, then a coalgebra maps a given state $x \in X$ into one or several successor states together with observations that can be made in the state x. In this setting, a state-based system is characterized by the observations that can be made during its run. For example, a deterministic, not necessarily terminating transition system is described by a T-coalgebra $[\mathit{stop}, \langle \mathit{value}, \mathit{next}\rangle] : X \to T(X)$ for the functor $T(X) = 1 + A \times X$ where A is an arbitrary non-empty set of observations. Given a state $x \in X$, $[\mathit{stop}, \langle \mathit{value}, \mathit{next}\rangle](x)$ is either the terminating state *stop* in which no observation is possible, or there exists the successor state $\mathit{next}(x)$ and the observation $\mathit{value}(x) \in A$.

Final T-coalgebras, as the dual concept to initial algebras, are characterized by the existence of a unique homomorphism from any other T-coalgebra into the final T-coalgebra. For the functor $T(X) = 1 + A \times X$, the final coalgebra is $[\mathit{empty}, \langle \mathit{head}, \mathit{tail}\rangle] : A^\infty \to A \times A^\infty$. A^∞ is the set containing all finite and infinite sequences with elements from A. Final T-coalgebras, if they exist, are the greatest fixed point of the functor T. For polynomial functors and even for the finite power set functor, final coalgebras exist. Polynomial functors are completely sufficient for our purposes.

Coinduction is – as well as induction – a definition and proof principle. The definition principle uses the fact that homomorphisms from arbitrary T-coalgebras into the final T-coalgebra exist, while the proof principle uses their uniqueness. Especially bisimulation is an important coinductive proof rule. It says that each binary relation on a final coalgebra that is closed under the operations of the coalgebra is contained in the equality relation.

3.2 Coalgebras and Coinduction in Isabelle/HOL

Coalgebraic types are available in Isabelle/HOL in the extension described in [17]. This extension makes use of coinductively defined sets, a definition principle available in Isabelle/HOL, and uses it to define lazy lists. As an example, consider the coinductive definition of possibly infinite lists as stated in [17]. The following definition specifies the set of lazy lists *llist*(A) over a given set A.

coinductive $\mathit{llist}(A)$
  intros
    $\mathit{NIL\_I} : \; \mathit{NIL} \in \mathit{llist}(A)$
    $\mathit{CONS\_I} : \; [a \in A \; ; \; M \in \mathit{llist}(A)] \implies \mathit{CONS} \; a \; M \in \mathit{llist}(A)$

Based on this definition, the coinductive data type *llist* is derived. For details of this specification, we refer to [17].

For reasoning about equality on coalgebraic types, we use the concept of bisimulations. A bisimulation is a binary relation $\sim$ on a coalgebraic type that is closed under the operations of the coalgebra. For lazy lists, this means that from $CONS\ a\ l\ \sim\ CONS\ a'\ l'$, we can deduce that $a = a'$ and $l\ \sim\ l'$. In Isabelle/HOL, we have stated this definition where we use the symbol $\diamondsuit$ to denote the empty list and $x \leadsto L$ as abbreviation for $CONS\ x\ L$ (read "$\leadsto$" as "leads to" in the sense that one state is transformed into another):

```
 bisimulation :: "('a llist × 'a llist) set ⇒ bool"
 bisimulation R == ∀ x ∈ R. (  x=(♢,♢) ∨
 ( ∃ x1 x2 L1 L2. x = ( x1⤳L1 , x2⤳L2 ) ∧ x1=x2 ∧ (L1,L2)∈R ) )
```

The lazy list package for Isabelle [17] provides a slightly different form of bisimulations:

```
llist_equalityI:
      ⟦(l1, l2) ∈ r; r ⊆ llistD_Fun (r ∪ range (λx. (x, x)))⟧
       ⟹ l1 = l2
```

Since our definition is closer to the one introduced in [10], we prefer it in our proofs. We have easily shown that it implies the one required to apply *llist_equalityI* (for the detailed Isabelle proof, we refer to [12]):

```
pauls_equiv :
    bisimulation r ⟹ r ⊆ llistD_Fun (r ∪ range (λx. (x, x)))
```

To use coalgebraically defined types, we need to be able to define not necessarily terminating recursive functions. For lazy lists, *llist_corec* allows us to define a function $'a \longrightarrow' b\ llist$ as follows where f is some partially defined function $f :' a \rightharpoonup' b \times\ 'a$ and $\_1$ and $\_2$ the usual projection functions:

$$llist\_corec\ x\ f = \begin{cases} \diamondsuit & \text{if } f\ x = \perp \\ (f\ x)_1 \cdot llist\_corec\ (f\ x)_2\ f & \text{else} \end{cases}$$

These definitions are the basis for our coalgebraic framework for the semantics of imperative programming languages and for correctness proofs of compiler transformations.

4 Programs as Elements of Coalgebras

Operational approaches to the formal semantics of imperative programming languages typically define the semantics of a given program as the sequence of states reached during the run of the program or, more generally in case of indeterminism, as a state transition system. Hence, it is a natural consequence to regard programs as elements of suitable coalgebras, namely as functions that take a state as input and output a new state together with possible observations. This view is in line with the intention of the two classical approaches to operational semantics which are abstract state machines (ASMs) [6] and structural operational semantics (SOS) [18]. We concentrate here on SOS but all our developments can be applied to ASMs as well. This holds because every SOS semantics can be transformed into an equivalent ASM semantics and vice versa [4].

4.1 Structural operational semantics (SOS)

Structural operational semantics (SOS), also called small-step semantics, concentrates on individual steps of program execution and how these single steps are integrated in the overall execution. Assumptions of inference rules formalize smaller steps while their embedding into the larger program context is defined in the conclusion. Individual steps are described by axioms. Such an individual step is either termination of execution $< p, \sigma > \rightarrow \ \sigma'$ in the final state σ' or it is a state transition $< p, \sigma > \rightarrow < p', \sigma' >$ denoting that the execution of p in state σ yields a new program p' to be executed in the succeeding state σ'. p' is often called *continuation*. The conclusions of inference rules define the embedding of such program parts into their larger context. As typical examples for small-step definitions, consider these inference rules:

$$\frac{<S_1,\sigma> \ \rightarrow \ <S_1',\sigma'>}{<S_1;S_2,\sigma> \ \rightarrow \ <S_1';S_2,\sigma'>} \qquad \frac{\mathsf{Eval}(\mathit{cond})=\mathit{true}}{<\textbf{if} \ \mathit{cond} \ \textbf{then} \ S_1 \ \textbf{else} \ S_2,\sigma> \ \rightarrow \ <S_1,\sigma>}$$

$$\frac{<S_1,\sigma> \ \rightarrow \ \sigma'}{<S_1;S_2,\sigma> \ \rightarrow \ <S_2,\sigma'>} \qquad \frac{\mathsf{Eval}(\mathit{cond})=\mathit{false}}{<\textbf{if} \ \mathit{cond} \ \textbf{then} \ S_1 \ \textbf{else} \ S_2,\sigma> \ \rightarrow \ <S_2,\sigma>}$$

$$< \textbf{skip}, \sigma > \ \rightarrow \ \sigma$$

$$< \textbf{while} \ \mathit{cond} \ \textbf{do} \ S, \sigma > \ \rightarrow < \textbf{if} \ \mathit{cond} \ \textbf{then} \ (S; \textbf{while} \ \mathit{cond} \ \textbf{do} \ S) \ \textbf{else} \ \textbf{skip}, \sigma >$$

The first two inference rules in the left column describe how the execution of a sequence of statements S_1 is integrated into a larger context, namely the sequence of statements $S_1; S_2$. The first two rules on the right-hand side specify the execution of the if-statement. The first axiom defines the effect of the skip-statement. The last axiom describes the while-loop by reducing its semantics to the semantics of the if-statement. This semantics is deterministic, as a simple case distinction over the possible states and continuation programs shows.

In a small-step semantics, the program to be executed is an explicit part of the state. Each state $< p, \sigma >$ contains a continuation program p. In the initial state, p is the original program while in the final state, p is simply the empty program. The axioms and inference rules of a small-step semantics define how to rewrite this program during each state transition.

4.2 Coalgebraic Semantics for SOS

We define the operational semantics of programming languages by a function that maps each program together with an initial state to an element of the final coalgebra of the functor $T(X) = 1 + A \times X$ where A is the set of states reached during computation. The carrier set of this coalgebra is A^∞. This means that the semantics of a program is described by a finite state transition list if program execution terminates and by an infinite state transition list in case of non-termination.

SOS defines semantics of programming languages as a function that maps tuples $< p, a >$ to tuples $< p', a' >$, thereby denoting that the execution of p in state a yields a new program p' to be executed in the succeeding state a', or, in case that program execution terminates, as a mapping of $< p, a >$ to a' which is a final state. Hence, each SOS semantics can be considered as a function that takes a program p and an initial state a as input and iteratively defines a finite or infinite list with elements of A.

Hence, each SOS semantics corresponds to a coalgebra with the coalgebra operation $[\mathsf{stop}, \langle \mathsf{value}, \mathsf{next} \rangle]$. stop represents program termination. Otherwise the current state is observable and denoted by the function value. The rest of the observable state transition sequence is obtained by applying the function next to the current state. Given an SOS semantics, we define a function $[\![SOS]\!] : A \times P \to A^\infty$ coinductively as the unique coalgebra homomorphism in the diagram below:

$$\begin{tikzcd}
A \times P \arrow[r, dashed, "{[\![SOS]\!]}"] \arrow[d, "{[\mathsf{stop},\langle \mathsf{value},\mathsf{next}\rangle]}"'] & A^\infty \arrow[d, "{[\mathsf{empty},\langle \mathsf{head},\mathsf{tail}\rangle]}"] \\
1 + A \times (A \times P) \arrow[r, dashed, "{\mathsf{id}+(id\times[\![SOS]\!])}"'] & 1 + A \times A^\infty
\end{tikzcd}$$

The defining equations of $[\![SOS]\!]$ are:

$$[\![SOS]\!](a,p) = \begin{cases} \mathsf{cons}(\mathsf{value}(a,p), \mathsf{next}(a,p)) & \text{if } [\mathsf{stop}, \langle \mathsf{value}, \mathsf{next} \rangle](a) \neq \mathsf{stop} \\ & \text{i.e. if } < p, a > \to < p', a' > \\ & \quad\text{or if } < p, a > \to a' \\ () & \text{if } [\mathsf{stop}, \langle \mathsf{value}, \mathsf{next} \rangle](a) = \mathsf{stop} \\ & \text{i.e. if no successor state exists} \end{cases}$$

This definition assigns each program and each initial state an element of the final coalgebra A^∞ which is the final coalgebra of the functor $T(X) = 1 + A \times X$.

5 Similarity Relations

As demonstrated by the examples in Section 2, we might consider the behavior of two different programs being equivalent even though their state transition sequences are not identical. In this section, we define two kinds of similarity relations on lists. The first, which we call *collapsings*, allows us to collapse any finite number of consecutive identical states to just one single state in a given state transition sequence. Collapsings are vital for verifying compiler optimizations that change the number of execution steps in a certain program part. A very simple example is the elimination of no-ops or skip statements. A slightly more complicated example on machine code level is the replacement of several simple operations by another more complicated equivalent operation or the bundling of simple operations so that they can be executed in one step. This is especially important when verifying optimizations for modern VLIW (very long instruction word) processors like the Intel Itanium architecture. These optimizations can easily be proved correct with the notion of collapsings. In Subsection 5.1, we introduce two different definitions for collapsings and summarize the proof for their equivalence in Isabelle/HOL. The second kind of similarity relations, *abstractions*, allows us to simplify state transition sequences by applying an abstraction function element-wise to the contained states. Abstractions are defined in Subsection 5.2.

5.1 Collapsing Relations

Collapsings are binary relations on sequences which capture the idea that we want to consider any finite subsequence of consecutive equal states to be equivalent to only one occurrence of that state by collapsing this finite sequence to one single state. In principle, there are two ways to approach the definition of collapsing relations. In the rest of this subsection, we present our two alternative definitions of collapsings and their equivalence proof within Isabelle/HOL.

5.1.1 Similarity using a merging function

Our first approach to collapsings is based on a unique minimal form for a sequence, called the *merging of the sequence*. This is the sequence in which all finite repetitions of an element are collapsed to one single occurrence of that element. Two lists are considered similar if their mergings are contained in a bisimulation. The definition of merging functions requires formalizations of finite prefixes and subsequences of potentially infinite sequences. In particular, we require the notion of a *maximum finite prefix* which is the (length-maximal) prefix of equal elements of a sequence. In Isabelle/HOL, this can be expressed by the following predicate:

```
is_max_prefix p L ≡
  p prefixes L ∧
  ∃x. set p = {x} ∧
  ∀other. ( other prefixes L ∧ ∃x. set other = {x})
           ⟶ size other ≤ size p
```

Since our sequences are potentially infinite, such a prefix does not necessarily need to exist. E.g. if our list is the infinite repetition of a single state[1], no maximum finite prefix exists.

In our formalization, we have also defined the function `split_finite_prefix` which splits the maximum prefix off of a sequence (if it exists), returning the prefix element and the rest of the list, or `None` if the list is empty:

```
split_finite_prefix L ≡
case L of ◇ ⇒ None
| x ⇝ L2 ⇒
 if ∃p. is_max_prefix p L then
        Some (x, cut_maxn (size (max_prefix L)) L)
    else Some (x, L2)
```

[1] For abbreviation, such a list is called boring.

If the list does not have such a prefix, only the first element is split off. Applying this function corecursively to a lazy list yields the desired merging function:

```
merging L ≡ llist_corec L split_finite_prefix
```

The merging function defined in this way does exactly what we want: It collapses sequences of equal elements into one single occurrence of that element. If the list is an infinite repetition of some x, then the merging function behaves as a simple copy function since a maximum prefix never exists. Note that the merging function is not computable because `split_finite_prefix` is not computable. This is also the reason why the proof that two lists are similar cannot be automated. For such a proof, one needs to find a bisimulation which contains the pair of their mergings. In concrete cases, this can be a tedious task. For finite lists, we have the possibility to explicitely name all the maximum simple prefixes in order to collapse them into single (equal) elements. For infinite lists, we must exhibit some structure in our lists, such that whenever we remove a maximum simple prefix from both lists, we can do so again, etc. Reasoning about similarity of infinite lists becomes easier when using a directly defined relation, as introduced below.

5.1.2 Defining a similarity relation directly

Alternatively, we coinductively define a relation $\approx$ using the following introduction rules where $l \cong p$ denotes the case that both l and p are simple x-prefixes, i.e. finite repetitions of an element x:[2]

```
coinductive "absRel"
intros
 nil : "◇ ≈ ◇"
 step : "⟦  p≅q ; L ≈ M ⟧ ⟹ p@L ≈ q@M"
```

This definition expresses that the empty list is similar to the empty list, and if two lists are similar then we can prepend two simple x-prefixes of different length but with the same element x to them. Although this definition is short, concise and – as we shall see lateron – easy to use in proofs, one cannot instantly see that it is equivalent to our first definition of collapsings based on the merging function. The most striking problem is that we cannot "walk back". This means that when $P \approx Q$ holds for some P and Q, we only know that there is *some* split of P and Q such that the respective prefixes of P and Q are finite x-prefixes for some x and that the suffixes are again similar but there is no way to constructively determine this prefix-suffix-pair, though. Proofs for such properties of coinductive as well as inductive sets often make use of the *cases* rule, the logical equivalent of "walking back" an introduction step. Applying this rule to a statement of the form $A \approx B$ only yields the weak statement that such a split exists, which is not sufficient for most uses. To still make this definition useful, we prove a strengthened cases rule: We show that whenever we split our sequences at the position of the first change, i.e. after the maximum prefix defined in the last section, we retain similarity:

```
lemma case_strong :
 "⟦p@L≈q@M ; is_max_prefix p p@L ; is_max_prefix q q@M ⟧ ⟹ L≈M "
```

[2] The symbol @ denotes the append operation on lists.

With this lemma, we have proved abstract properties (like transitivity) of our relation $\approx$, cf. [12] for details. Moreover, this rule is an important bridge between our two definitions of similarity which we need to prove their equivalence.

5.1.3 Proving equivalence

The two definitions for collapsings are equivalent:

$$merging(L_1) = merging(L_2) \iff L_1 \approx L_2$$

One direction of the proof is simple. We show that every lazy list is similar to its own merging. A shortened version of this direction of the proof is given here in ISAR [20] notation:

```
lemma abs_sim :  shows "A ≈ merging A"
proof -
 let ?X = "⋃L.  {( L, merging L ) }"
 have "∀x. x ∈ ?X ⟶ ( x = ( ◇ , ◇ ) ∨
        ( ∃ L M p q. x = (p@L, q@M) ∧ p≅q ∧ (L, M) ∈ ?X ∪ absRel ) )"
  proof -
    { fix x assume "x ∈ ?X"
      then obtain L where x_form : "x = ( L, merging L )" by auto
      hence "x = ( ◇ , ◇ ) ∨
        ( ∃ L M p q. x = (p@L, q@M) ∧ p≅q ∧ (L, M) ∈ ?X ∪ absRel )"
      proof cases
        assume "L=◇" show ?thesis by simp add: lnil_abs_invariant
      next
        assume L_not_empty : "L≠◇"
        then obtain l and M where L_form : "L=l⇝M" by simp add: 13
        thus ?thesis
        proof cases
          assume "boring L" hence "x = ([l]@M , [l]@M)" by auto
          thus ?thesis by ( unfold sameRel_def , auto intro : sim_refl )
        next
          assume "¬ boring L"
          then obtain pmax where is_mp: "is_max_prefix pmax L" ...
          moreover then obtain suffix where lsplit: "L=(pmax@suffix)" ...
          moreover from is_mp and L_form have pml : "set pmax = {l}"   ...
          ultimately have "merging L = l ⇝ merging suffix"   ...
          hence " x = (pmax@suffix, [l]@(merging suffix) )" ...
          moreover have "pmax ≅ [l]" by simp add: sameRel_def
          ultimately show ?thesis by ( auto  )
        qed
      qed }
    thus ?thesis by blast
  qed
  moreover have "(A, merging A) ∈ ?X" by auto
  ultimately show ?thesis by ( rule_tac X="?X" in absRel.coinduct, simp )
qed
```

Having thus proven that $L \approx merging(L)$, we are able to complete the proof for one direction, namely $merging(L_1) = merging(L_2) \implies L_1 \approx L2$, since $\approx$ is an equivalence relation. The other direction poses more difficulty. Fortunately, we have our "strengthened cases rule". Using it, we show that

$$X = \{(merging(x), merging(y)) \mid x \approx y\}$$

is a bisimulation, which requires that

$$(merging(x), merging(y)) = \begin{cases} (\Diamond, \Diamond) \\ \textbf{or} \\ (a \leadsto merging(x'), a \leadsto merging(y')) \\ \text{where } x' \approx y' \end{cases}$$

Since we now know that we can remove a maximum finite α-prefix and the lists remain similar, we choose the corresponding suffixes as x' and y', respectively (and α for a). Due to the definition of the merging function as the removal of just this maximum prefix, the second requirement is also true.

The equivalence between the two definitions of collapsings turns out to be immensely useful. In our experiments, we found that while it is relatively simple to use the coinduction rule of our similarity relation (the second definition variant), it is much more difficult to show that the image of two different lazy lists under a corecursively defined function (the merging function in the first definition alternative) is equal.

5.2 Abstractions from Irrelevant Details

Reconsider the two programs computing the Gauss sum in Figure 3 which are not semantically identical, nevertheless similar if one ignores the variables i and s and projects each state on only that part one is interested in. In this subsection, we introduce the idea of simplifying states by abstraction functions formally.

Definition 5.1 [Abstractions on Sequences] Let $f : A \to A$ be a function that modifies states in A. Then we define coinductively a function $T_f : A^\infty \to A^\infty$ by

$$\mathsf{head}(T_f(l)) = T_f(\mathsf{head}(l)) \quad \text{and} \quad \mathsf{tail}(T_f(l)) = T_f(\mathsf{tail}(l)).$$

T_f transforms a given sequence by applying f to each element in the original sequence. $T_f(l)$ is called an abstraction of l with respect to f. $\diamond$

Definition 5.2 [Semantic Equivalence modulo Abstraction] Let l and k be state transition sequences, $l, k \in A^\infty$. l is semantically equivalent to k modulo the abstraction functions f and g if $T_f(l)$ and $T_g(k)$ are similar. $\diamond$

With this definition, we prove in Subsection 6.3 that the semantics of the two programs in Figure 3 are semantically equal if one forgets about the values of the auxiliary variables i and s.

6 Correctness of Program Transformations

In this section, we demonstrate the usefulness of our coalgebraic approach to programming language semantics and correctness proofs of program transformations by verifying the transformations discussed by the motivating examples in Section 2.

For this purpose, we have specified the semantics of a simple while-language in Isabelle/HOL by formalizing the rules given in Subsection 4.1. A state is defined as a function $\mathit{State} : \mathit{Var} \rightarrow \mathbb{N}$ that maps variables to natural numbers. Moreover, we have specified the function *step* that formalizes execution of a single statement and returns the new statement and new state where computation continues: $\mathit{step} : (\mathit{Statement}, \mathit{State}) \rightharpoonup (\mathit{Statement}, \mathit{State})$. Its result is $\perp$ if there is no successor state. This transition function allows us to corecursively define the operational semantics of programs in our example language by assigning each configuration (each state and continuation program) a potentially infinite state transition sequence:

```
state_sequence :: "Configuration ⇒ State item llist"
state_sequence_def: "state_sequence C == llist_corec C step"
```

In this section, we show that the transformations introduced in Section 2 can be verified based on this coalgebraic semantics and by employing the similarity relations defined in Section 5. In Subsection 6.1, we start with a standard bisimulation proof to show that unreachable code can be eliminated, cf. Figure 1. We continue in Subsection 6.2 by using collapsings to verify the shifting of loop-invariant code out of loops, cf. Figure 2. Finally, in Subsection 6.3, we demonstrate how abstractions can be used when transformations involve the elimination of program variables, cf. Figure 3.

6.1 Correctness Proofs involving the Classical Coinductive Case

Consider the following program template:

$P_S := \textbf{while}\ \mathit{true}\ \textbf{do}\ i := i + 1\ \textbf{od};\ S$

where S is an arbitrary statement obviously never reached. Hence, the semantics of P_S should be independent of S. We prove this by verifying that the two state transition sequences of arbitrary instantiations S_1 and S_2 of S are equal:

$$\forall_{S_1, S_2, \sigma \in \mathit{State}}.\ \mathit{seq}(P_{S_1}, \sigma) = \mathit{seq}(P_{S_2}, \sigma)$$

Proving this statement by showing that the two state transition sequences are in a bisimulation relation is straightforward. The configuration of P_S can only have one of two continuations: P_S itself, when the body of the loop has just been evaluated, and $P_S^E := (i := i+1;\ \textbf{while}\ \mathit{true}\ \textbf{do}\ i := i+1\ \textbf{od};\ S)$, when the condition which is always true has been checked in the preceding step. During program execution, the continuation of this program infinitely alternates between these two continuations and the thereby defined state transition sequence is independent of S. For details of the coinductive proof in Isabelle/HOL that $\mathit{seq}(P_{S_1}, \sigma) = \mathit{seq}(P_{S_2}, \sigma)$ cf. [12].

6.2 Correctness Proofs involving Collapsings

In the preceding subsection, the state sequences of the two programs were actually equal. Obviously, this is not always the case. As example consider the extraction of loop-invariant code in Figure 2. For its verification, we need to show (where P_1

and P_2 are the two programs in Figure 2):

$$\forall_{\sigma \in State}\ seq(P_1, \sigma) \approx seq(P_2, \sigma)$$

In such correctness proofs of program transformations within Isabelle/HOL, the relation $\approx$ (cf. Subsection 5.1) comes in handy. To show that two lazy lists are similar, it suffices to show that there exists a set of lazy list pairs that contains (x, y) such that the coinduction rule of the relation $\approx$ holds:

```
absRel.coinduct :  ⟦(a , b) ∈ X;    ⋀z. z ∈ X ⟹ z = (◇, ◇)
  ∨ (∃L M p q. z = (p@L, q@M) ∧ p≅q ∧ ((L, M) ∈ X ∨ L ≈ M) )⟧
  ⟹ a ≈ b
```

It is often straightforward to find such a set X. In most cases, the pairs consisting of the traces of the two programs to be shown to be equivalent (all state lists that the programs pass) can be constructively specified and have the desired properties even if they are infinite. These proofs typically make use of the fact that one can "stay in X forever". In our example verification, we choose:

$$X = \bigcup_{\sigma} \{(seq(P_1, \sigma), seq(P_2, \sigma))\}$$

In Isabelle/HOL, we have verified that $\forall_{(x,y) \in X}\ x \approx y$ which is the desired result. For proof details, we refer to [12].

6.3 Correctness Proofs involving Abstractions

With the definition of semantic equivalence modulo abstractions, cf. Definition 5.2, we have a method to relate different state transition sequences with each other. The utilized abstraction functions can rename variables, project onto only the interesting ones, or put the computed values into relation (e.g. if one program computes its results with respect to metric and the other with respect to non-metric units), just to mention a few of the useful possibilities. Together with collapsings, they allow us to relate even infinite state transition sequences.

Example 6.1 Consider again the programs in Figure 3. Their semantics is described by finite and infinite state transition sequences in A^∞ whereby each state is a mapping of the variables contained in the program to their current values (which are undef if no definition has occurred yet). We define an abstraction function $f : A \to A$ by

$$f(a) = \{(x = v) \mid (x = v) \in a\ \wedge\ x \notin \{i, s\}\}$$

This function projects each state to its relevant part by ignoring the current values for the non-interesting variables i and s.

It is straightforward to define a collapsing which contains $(T_f(l), T_{id}(k))$ whereby $id : A \to A$ denotes the identity function on A, l the semantics of the first program and k the semantics of the second program in Figure 3. $\diamond$

7 Related Work

Proving semantic equivalence of programs and systems, resp., has been intensively investigated in the area of compiler verification. Early research on compiler verification was carried out in the Boyer-Moore theorem prover considering the translation of the programming language Piton [14]. The german *Verifix* project investigated the construction of correct compilers without performance loss, see [3] for an overview. Recent work has concentrated on transformations taking place in compiler frontends. The formal verification of the translation from Java to Java byte code and formal byte code verification was investigated in [11]. This latter work was preceeded by the work on the formalization of Java and the proof of its type safety within the theorem prover Isabelle/HOL [16]

Lately, also coalgebraic methods have been used successfully in the specification of and reasoning about programming languages and systems. In [7,9], the semantics of object-oriented programming languages has been defined coalgebraically. The goal of the VFiasco project [8] is the verification of an operating system with coalgebraic methods. [19] describes the coalgebraic class specification language CCSL.

Coalgebraic proof methods are not the only formalism capturing the characteristics of semantics for non-terminating programs. One can also use labeled transition systems [13]. Bisimulation can be used within both formalisms. Our notion of bisimulation with collapsings (operating on coalgebraic datatypes) und the notion of weak bisimulation [13] (operating on labeled transition systems) may be used for the same purposes: defining program equivalence up to observable steps.

In our own work [1] we have proved a dead code elimination algorithm as used in compilers correct using bisimulation on Kripke structures. In [5], we describe how coalgebras and coinduction may be used in compiler verification. Finally, our work on formalizing and transforming data flow dependent computations [2] also shows, as the work presented in this paper, that the choice of formalization is vital for the proof success when using theorem provers.

8 Conclusions

We have presented a novel framework for the coalgebraic verification of program equivalence. By assigning each program an element of a suitable final coalgebra, we have defined semantics also for non-terminating programs. By verifying simple, yet typical, also non-refining optimizations in compilers, we have shown that our framework is able to formalize correctness proofs for transformations found in modern optimizing compilers. We have also shown how we have formalized this framework together with the correctness proofs within the theorem prover Isabelle/HOL. In particular, we have presented two different notions of correctness and proved their equivalence within Isabelle/HOL. While the first notion captures very well the mathematical intuition, the second is better suited for mechanized proofs. With these results we have contributed to the question how formalizations within theorem provers are to be chosen in order to simplify mechanized proofs and to reduce verification costs.

To be able to also relate programs with not exactly the same state transition behavior, we have introduced two kinds of similarity relations (*collapsings* and *semantic equivalence modulo abstractions*). With the notion of collapsings, we relate pairs of state transition sequences which contain corresponding finite subsequences of consecutive identical states which might be of different length. In a collapsing, we regard each such finite subsequence with identical states as a single state. In addition, the notion of semantic equivalence modulo abstractions allows us to relate pairs of state transition sequences which both can be brought down to a common denominator by applying abstraction functions element-wise on them. These two notions, collapsings together with abstractions on state transition sequences, are a powerful instrument in the verification of semantic equivalence. With them, our definition of semantic equivalence is entirely based on the semantics side, not on any syntactic criterion. Every transformation is correct (with respect to some abstraction functions) if the abstractions of the state transition sequences of the original and the transformed program are contained in a suitable collapsing. Even two syntactically completely unrelated programs can be semantically equivalent with respect to a given abstraction. The question how the abstraction function is to be chosen depends on the context, i.e. on the question what we want to consider as being semantically equivalent. We are convinced that this framework for the verification of program or system transformations, resp., can also be applied in other areas of software engineering as well. It is subject of future work to further explore this.

References

[1] Jan Olaf Blech, Lars Gesellensetter, and Sabine Glesner. Formal Verification of Dead Code Elimination in Isabelle/HOL. In *Proc.3rd IEEE Int'l Conf. on Software Engineering and Formal Methods*, 2005. IEEE Comp. Soc. Press.

[2] Jan Olaf Blech, Sabine Glesner, Johannes Leitner, and Steffen Mülling. A comparison between two formal correctness proofs in Isabelle/HOL. In *Proc. Workshop Compiler Optimization meets Compiler Verification (COCV 2005)*. Elsevier, April 2005.

[3] Sabine Glesner, Gerhard Goos, and Wolf Zimmermann. Verifix: Konstruktion und Architektur verifizierender Übersetzer (Verifix: Construction and Architecture of Verifying Compilers). *it - Information Technology*, 46:265–276, 2004. Print ISSN: 1611-2776.

[4] Sabine Glesner. ASMs versus Natural Semantics: A Comparison with New Insights. In *Abstract State Machines - Advances in Theory and Applications, Proc. 10th Int'l Workshop, ASM 2003*, 2003. Springer LNCS Vol. 2589.

[5] Sabine Glesner. A Proof Calculus for Natural Semantics Based on Greatest Fixed Point Semantics. In *Proc. Workshop Compiler Optimization meets Compiler Verification (COCV 2004)*, 2004. Elsevier, Electronic Notes in Theoretical Computer Science (ENTCS).

[6] Yuri Gurevich. Evolving Algebras 1993: Lipari Guide. In *Specification and Validation Methods*, pages 231–243. Oxford University Press, 1995.

[7] Ulrich Hensel, Marieke Huisman, Bart Jacobs, and Hendrik Tews. Reasoning about Classes in Object-Oriented Languages: Logical Models and Tools. In *Progr. Lang. and Systems - ESOP'98*, 1998. Springer LNCS Vol. 1381.

[8] Michael Hohmuth, Shane G. Stephens, and Hendrik Tews. Applying source-code verification to a microkernel – The VFiasco project. In *Proceedings of the 10th SIGOPS European Workshop*, 2002.

[9] Marieke Huisman. *Reasoning about Java programs in higher order logic using PVS and Isabelle.* PhD thesis, University of Nijmegen, 2001.

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

[11] Gerwin Klein and Tobias Nipkow. Verified Bytecode Verifiers. *Theoretical Computer Science*, 298:583–626, 2003.

[12] Johannes Leitner. Coalgebraic Methods in the Verification of Optimizing Program Transformations Using Theorem Provers. Minor Thesis (*Studienarbeit*), University of Karlsruhe, 2005.

[13] Robin Milner. *Communication and concurrency.* Prentice Hall International (UK) Ltd., Hertfordshire, UK, 1995.

[14] J. Strother Moore. A Mechanically Verified Language Implementation. *Journal of Automated Reasoning*, 5(4):461–492, 1989.

[15] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic.* Springer, LNCS Vol. 2283, 2002.

[16] Tobias Nipkow and David von Oheimb. Java$_{light}$ is Type-Safe – Definitely. In *Proc. 25th ACM Symp. Principles of Progr. Languages*, 1998. ACM Press.

[17] Lawrence C. Paulson. A Fixedpoint Approach to (Co)Inductive and (Co)Datatype Definitions, 2004. www.cl.cam.ac.uk/ Research/HVG/Isabelle/dist/Isabelle2004/doc/ind-defs.pdf.

[18] Gordon D. Plotkin. A structural approach to operational semantics. Report DAIMI FN-19, Comp. Sc. Department, Aarhus University, Denmark, 1981.

[19] Jan Rothe, Hendrik Tews, and Bart Jacobs. The Coalgebraic Class Specification Language CCSL. *J. Univ. Comp. Sc. (J.UCS)*, 7(2):175–193, 2001.

[20] Markus Wenzel. *Isabelle/Isar – A Versatile Environment for Human-Readable Formal Proof Documents.* PhD thesis, Institut für Informatik, Technische Universität München, 2002.