Program Verification by Coinduction

Brandon Moore
Univeristy of Illinois at Urbana-Champaign
bmmoore@illinois.edu

Grigore Rosu
Univeristy of Illinois at Urbana-Champaign
bmmoore@illinois.edu

Abstract

We present a program verification framework based on coinduction, which makes it feasible to verify programs directly against an operational semantics, without requiring intermediates like axiomatic semantics or verification condition generators. Specifications can be written and proved using any predicates on the state space of the operational semantics.

We implement our approach in Coq, giving a certifying language-independent verification framework. The core proof system is implemented as a single module imported unchanged into proofs of programs in any semantics. A comfortable level of automation is provided by instantiating a simple heuristic with tactics for language-specific tasks such as finding the successor of a symbolic state, and for domain-specific reasoning about the predicates used in a particular specification. This approach also smoothly allows manual assistance at points the automation cannot handle.

We demonstrate the power of our approach by verifying algorithms as complicated as Schorr-Waite graph marking, and the versatility by instantiating it for object languages in several styles of semantics. Despite the greater flexibility and generality of our approach, proof size and proof/certificate-checking time compare favorably with Bedrock, another Coq-based certifying program verification framework.

1. Introduction

Formal verification is a powerful technique for ensuring program correctness, and approaches such as axiomatic semantics and verification condition generators make proofs feasible for many languages. However, these add substantially to the effort of formally specifying a language.

Even if only a proof system is desired, it is necessary to show that it is faithful to the desired object language. This generally requires an executable and thus testable semantics[1], and a proof of soundness. If two semantics are required, then every construct of the language must be defined twice. Some features, especially loops or mutual recursion, require proof rules substantially more complicated than corresponding rules of an operational semantics. A semantics for a mature language such as C[9], Java[5], or JavaScript[4] can take several years, even without giving multiple semantics.

In short, standard approaches are known to work well for many languages, but they only give a plan for designing and implementing a program verification framework for each new language, rather than theorems or systems that can be reused with many languages. To use a software engineering metaphor[2], Hoare Logic[13] is a design pattern[11] rather than a library.

This paper presents instead a single language-independent proof framework. The proof system requires only an operational semantics of a language, given as a transition relation on a set of configurations. The core of the approach is a single language-independent theorem which gives a coinduction principle for proving that certain claims about paths among configurations hold in a particular transition relation. Working in a proof assistant, the key theorem is proved once and then literally instantiated with different relations to support verification in different programming languages. Hoare-style specifications can be straightforwardly reduced to such claims, as shown in [19]. The key theorem applies also to nondeterministic semantics, but we focus in this paper on deterministic languages for simplicity.

Proof automation is essential for the practicality of any program verification approach. We demonstrate our proofs can be effectively automated, on examples including heap data structures and recursive functions. We describe our automation strategy, the division of the implementation into reusable and language-specific components, and how we incorporate manual assistance on problems which are almost completely automated.

We first present our approach and a detailed example proof to illustrate it, then present experimental results and discuss proof automation before returning to a general soundness and completeness result.

Minimal granularity

$\langle$`while (x != 0) {x-=1}` $\mid \mathtt{x} \mapsto 3\rangle$
$\langle$`while (x != 0) {x-=1}` $\mid \mathtt{x} \mapsto 2\rangle$
$\langle$`while (x != 0) {x-=1}` $\mid \mathtt{x} \mapsto 1\rangle$
$\langle$`while (x != 0) {x-=1}` $\mid \mathtt{x} \mapsto 0\rangle$
$\langle$`skip` $\mid \mathtt{x} \mapsto 0\rangle$

Modest granularity

$\langle$`while (x != 0) {x-=1}` $\mid \mathtt{x} \mapsto 1\rangle$
$\langle$`if (x != 0) {x-=1; while (x != 0) {x-=1}}` $\mid \mathtt{x} \mapsto 1\rangle$
$\langle$`if (1 != 0) {x-=1; while (x != 0) {x-=1}}` $\mid \mathtt{x} \mapsto 1\rangle$
$\langle$`if (true) {x-=1; while (x != 0) {x-=1}}` $\mid \mathtt{x} \mapsto 1\rangle$
$\langle$`x-=1; while (x != 0) {x-=1}` $\mid \mathtt{x} \mapsto 1\rangle$
$\langle$`x:=0; while (x != 0) {x-=1}` $\mid \mathtt{x} \mapsto 1\rangle$
$\langle$`skip; while (x != 0) {x-=1}` $\mid \mathtt{x} \mapsto 0\rangle$
$\langle$`while (x != 0) {x-=1}` $\mid \mathtt{x} \mapsto 0\rangle$
$\langle$`if (x != 0) {x-=1; while (x != 0) {x-=1}}` $\mid \mathtt{x} \mapsto 0\rangle$
$\langle$`if (0 != 0) {x-=1; while (x != 0) {x-=1}}` $\mid \mathtt{x} \mapsto 0\rangle$
$\langle$`if (false) {x-=1; while (x != 0) {x-=1}}` $\mid \mathtt{x} \mapsto 0\rangle$
$\langle$`skip` $\mid \mathtt{x} \mapsto 0\rangle$

Figure 1. Example executions of example loop

2. Examples

We will illustrate how we formulate semantics and specifications with the following simple loop.

```
while (x != 0) {x-=1}
```

[1] attempting to execute an axiomatic semantics by proof search as in [25] has not been demonstrated for realistic languages or test suites

[2] less metaphorical when we formalize our mathematics in proof assistants

Two potential execution traces of the loop are given in Figure 1. Both take sufficiently fine steps for our approach to work. The longer trace takes steps of about the size we would expect from an evaluation contexts or small step semantics. Finer steps would not be a problem. The "minimal" trace is the coarsest semantics for which our approach is useful.

2.1 Specification

We can see this loop can only exit normally with `x` having the value 0, and will do so starting from non-negative integers. The loop may run forever if the language includes unbounded integers, which we assume to illustrate non-termination.

The claim we wish to make is that any state entering the loop with the variable `x` defined will either run indefinitely or reach a state which has just exited the loop, assigns `x` the value 0, and leaves any other local variables unchanged from the initial state. In some imaginary Hoare logic with a "partial-correctness" interpretation this might be written as

$$\left\{\!\left\{ \top \right\}\!\right\} \texttt{ while (x != 0) \{x-=1\} } \left\{\!\left\{ \texttt{x}= 0 \right\}\!\right\}$$

This sort of notation is language-specific in several ways. Code written between the predicates is translated to some relation between the code of initial and final states, program variables are written directly in place of their values in the predicates, and some framing condition governs what parts of the state are assumed not to change. In the interests of a language-independent notation we will dispense with these convenient conventions, and write a specification directly in terms of the underlying states of a transition relation.

Suppose that a configuration of the transition system defining our example language consists of a pair $\langle T \mid \sigma \rangle$ of a current statement T and a store σ mapping identifiers to values. Then the desired specification is written

$$\forall n, T, \sigma \,.\, \langle \texttt{while (x!=0) \{x -= 1\}; } T \mid \texttt{x} \mapsto n, \sigma \rangle \\ \Rightarrow \{ \langle T \mid \texttt{x} \mapsto 0, \sigma \rangle \}$$

A *reachability claim* $c \Rightarrow P$ is valid if the initial configuration c can reach a state in the set P or can take an infinite number of steps. The leading universal quantification gives a collection of claims making up our specification.

In general, we suppose a semantics consists of a set C of configurations and a step relation $R \subseteq C \times C$. We also write $a \rightarrow_R b$ for $(a, b) \in R$, and drop the subscript when R is understood. Conversely, we subscript the claim arrow $\Rightarrow$ when it is necessary to be explicit about the transition relation. We define the set of valid claims $\text{valid}_R \subseteq C \times \mathcal{P}(C)$ as

$$\text{valid}_R \equiv \{(c, P) \mid c \Rightarrow_R P\}$$

This approach to specification reduces claims about particular programs to a uniform sort of claim about the language semantics itself, asking if certain sorts of paths exist among various states in the configuration space of the semantics.

2.2 Proof

Now we introduce the proofs of our system. We begin with a direct but informal argument for the correctness of the example specification, which relies on the simple and predictable behavior of the example loop. The proof is divided into local reasoning about short sequences of execution steps from symbolic configurations, and global arguments about how these path segments can be fit together. Then we introduce a lemma which replaces and generalizes the global argument, and requires less precise local reasoning than the original informal proof.

Assume an operational semantics for our language where

$$\langle \texttt{while (x!=0) \{x-=1\}; } T \mid \texttt{x} \mapsto n, \sigma \rangle \\ \rightarrow^+ \langle \texttt{while (x!=0) \{x-=1\}; } T \mid \texttt{x} \mapsto n - 1, \sigma \rangle$$

for $n \neq 0$ and any T and σ, and

$$\langle \texttt{while (x!=0) \{x-=1\}; } T \mid \texttt{x} \mapsto 0, \sigma \rangle \rightarrow^+ \langle T \mid \texttt{x} \mapsto 0, \sigma \rangle$$

From this we can informally argue that the specification is valid. For $n \geq 0$ we get a path reaching the target state by induction on n, and for $n < 0$ we can assemble an infinite path showing execution diverges, by concatenating the path segments starting with `x` equal to n, $n - 1$, $n - 2$, etc.

This is sound, but too specific. One problem is that this form of argument requires deciding in advance which configurations can diverge. It is also excessively specific to describe precisely the single state which will be reached after one loop iteration. Even in a deterministic language it is often most appropriate to give specifications that permit multiple results, such as not specifying the precise address where freshly heap-allocated results will be allocated at.

In fact it is sufficient to check that states of the form

$$\langle \texttt{while (x!=0) \{x-=1\}; } T \mid \texttt{x} \mapsto n, \sigma \rangle$$

with $n \neq 0$ reach some state of the form

$$\langle \texttt{while (x!=0) \{x-=1\}; } T \mid \texttt{x} \mapsto n', \sigma \rangle$$

in a positive number of steps, while still checking that the initial state with $n = 0$ reaches the desired target state.

It should be plausible this suffices: For any initial state we either reach an acceptable target state, or we can take some steps and reach another state that we claimed reaches the same target set. Simply wandering forward along these path segments will either eventually actually reach an acceptable target state, or extend forever to give an infinite path. This will be made precise in terms of coinduction when we prove the soundness of our approach in Section 5.

Now we introduce a proof lemma. Fix a transition relation R on domain C. We identify reachability claims about individual configurations with pairs in $C \times \mathcal{P}(C)$ and specifications with subsets $S \subset C \times \mathcal{P}(C)$. Define an operation

$$\text{step}_R(S) = \{(c, P) \mid c \in P \vee \exists d.(c, d) \in R \wedge (d, P) \in S\}$$

We will see later that the set of all valid claims is in fact the *greatest fixpoint* of this operation (in a lattice of sets ordered by inclusion). Given any monotone $F : \mathcal{P}(C \times \mathcal{P}(C)) \rightarrow \mathcal{P}(C \times \mathcal{P}(C))$, define the *$F$-closure* operation F^* by

$$F^*(X) = \mu Y \,.\, F(Y) \vee X$$

The operation $Y \mapsto F(Y) \vee X$ is monotone for any X. so all the fixpoints exist and we have a well-defined function $F^* : \mathcal{P}(C \times \mathcal{P}(C)) \rightarrow \mathcal{P}(C \times \mathcal{P}(C))$. With these definitions we state the lemma

Lemma 1. *$S \subseteq \text{valid}_R$ if $S \subseteq \text{step}_R(\text{step}_R^*(S))$*

We prove a generalization in Section 5. Now we prove our example, to demonstrate that proofs using this style of lemma are closely based on symbolic execution in the semantics.

As a set of claims, our example specification is

$$S \equiv \big\{ \big(\langle \texttt{while (x!=0) \{x-=1\}; } T \mid \texttt{x} \mapsto n, \sigma \rangle \\ , \{ \langle T \mid \texttt{x} \mapsto 0, \sigma \rangle \} \big) \quad \mid \forall n, T, \sigma \big\}$$

Applying the proof lemma, it suffices to show that

$$S \subseteq \text{step}_R(\text{step}_R^*(S)) \tag{1}$$

Suppose our transition relation R follows the "Modest granularity" trace in Figure 1. Then all of the initial configurations in

our specification have a first step which unfolds the while loop. In symbols, for any n, T, σ,

$$\langle \texttt{while (x!=0) \{x-=1\};}\ T \mid \texttt{x} \mapsto n, \sigma\rangle \rightarrow_R \langle \texttt{if (x!=0) \{x-=1; while(x!=0) \{x-=1\};}\ T \mid \texttt{x} \mapsto 0, \sigma\rangle$$

We use this in the proof of the original inclusion (1) by choosing the second case of the disjunction in step_R and instantiating d to match this step. Thus it suffices to show

$$\{(\langle \texttt{if (x!=0)\{x-=1; while(x!=0)\{x-=1\};}\ T \mid \texttt{x} \mapsto n, \sigma\rangle, \{\langle T \mid \texttt{x} \mapsto 0, \sigma\rangle\}) \mid \forall n, T, \sigma\} \subseteq \mathrm{step}_R^*(S)$$

These configurations all can take homologous steps to look up the program variable. Unfolding the fixpoint as $\mathrm{step}_R(\mathrm{step}_R^*(S)) \subseteq \mathrm{step}_R(\mathrm{step}_R^*(S)) \cup S = \mathrm{step}_R^*(S)$ exposes an application of step_R on the right hand side. Then we can take an execution step as before to reduce the goal to

$$\{(\langle \texttt{if (}n\texttt{!=0)\{x-=1; while(x!=0)\{x-=1\};}\ T \mid \texttt{x} \mapsto n, \sigma\rangle, \{\langle T \mid \texttt{x} \mapsto 0, \sigma\rangle\}) \mid \forall n, T, \sigma\} \subseteq \mathrm{step}_R^*(S)$$

more steps evaluate the if condition to leave the goal

$$\{(\langle \texttt{if (}n \neq 0\texttt{)\{x-=1; while(x!=0)\{x-=1\};}\ T \mid \texttt{x} \mapsto n, \sigma\rangle, \{\langle T \mid \texttt{x} \mapsto 0, \sigma\rangle\}) \mid \forall n, T, \sigma\} \subseteq \mathrm{step}_R^*(S)$$

The if-condition is now a single boolean constant, whose concrete value is given by the symbolic expression $n \neq 0$.

We can make a case distinction by using the fact that $A \subseteq X$ and $B \subseteq X$ imply $A \cup B \subseteq X$ for any sets A, B, X. We split our set of claims into the subset with $n = 0$,

$$\{(\langle \texttt{if (false)\{x-=1; while(x!=0)\{x-=1\};}\ T \mid \texttt{x} \mapsto 0, \sigma\rangle, \{\langle T \mid \texttt{x} \mapsto 0, \sigma\rangle\}) \mid \forall T, \sigma\} \subseteq \mathrm{step}_R^*(S)$$

and the rest,

$$\{(\langle \texttt{if (true)\{x-=1; while(x!=0)\{x-=1\};}\ T \mid \texttt{x} \mapsto n, \sigma\rangle, \{\langle T \mid \texttt{x} \mapsto 0, \sigma\rangle\}) \mid \forall T, n, \sigma, n \neq 0\} \subseteq \mathrm{step}_R^*(S)$$

Translating execution steps to proof steps as before leaves

$$\{(\langle T \mid \texttt{x} \mapsto 0, \sigma\rangle, \{\langle T \mid \texttt{x} \mapsto 0, \sigma\rangle\}) \mid \forall T, \sigma\} \subseteq \mathrm{step}_R^*(S)$$

and

$$\{(\langle \texttt{while (x!=0)\{x-=1\};}\ T \mid \texttt{x} \mapsto n-1, \sigma\rangle, \{\langle T \mid \texttt{x} \mapsto 0, \sigma\rangle\}) \mid \forall T, n, \sigma, n \neq 0\} \subseteq \mathrm{step}_R^*(S)$$

In the $n = 0$ case the current state matches the target state. We conclude by using the first case $c \in P$ in the definition of step_R to reduce to the trivial hypothesis

$$\langle T \mid \texttt{x} \mapsto 0, \sigma\rangle \in \{\langle T \mid \texttt{x} \mapsto 0, \sigma\rangle\}$$

For the non-zero case, we have arrived at a set of claims that are contained in the initial specification. We use this by unfolding the fixpoint $\mathrm{step}_R^*(S)$ differently, $S \subseteq \mathrm{step}_R(\mathrm{step}_R^*(S)) \cup S = \mathrm{step}_R^*(S)$. This reduces the goal to showing the inclusion

$$\{(\langle \texttt{while (x!=0)\{x =1\};}\ T \mid \texttt{x} \mapsto n-1, \sigma\rangle, \{\langle T \mid \texttt{x} \mapsto 0, \sigma\rangle\}) \mid \forall T, n, \sigma, n \neq 0\} \subseteq S$$

This holds by instantiating the variable n in the definition of S with the expression $n - 1$, and concludes the proof.

Lemma 1 can obviously be used with *any* operational semantics. This simple example served to illustrate in detail that a simple statement about subsets and fixpoints can be used for verification, and understood as a *language-independent proof system* allowing proof steps of finishing trivially, taking a symbolic step, and applying a claim from the specification being proved. Our full coinduction theorem allows a compositional choice of additional rules, and our soundness and relative completeness results in Section 5 state that our simple approach is as powerful as any other verification method can be. However, the crucial aspect of our approach is that it maintains these desirable results for all programming languages given by their operational semantics, at no additional definitional or proof effort. Compare that with the current state-of-the-art in mechanical verification, where in order to obtain a proof system for a new language given by its (trusted) operational semantics, an additional, axiomatic semantics needs to be defined, and language-specific and tedious soundness proofs need to be produced.

The question is: "Can this simple approach really work?"

	Size (lines)			Time (s)	
Example	Code	Spec.	Proof	Prove	Check
Simple					
undefined	2	3	1	3.5	2.0
average3	2	5	1	4.0	1.2
min	3	4	2	3.6	1.0
max	3	4	2	3.6	1.0
multiply	9	7	1	12.4	2.4
sum(rec)	6	8	6	5.5	1.3
sum(iter)	5	12	6	8.1	1.5
Lists					
head	1	6	1	2.6	1.2
tail	1	6	1	2.5	1.1
add	4	16	1	4.5	1.4
swap	9	13	1	19.6	4.2
dealloc	4	7	1	8.5	1.8
length(rec)	4	12	1	7.8	1.9
length(iter)	5	17	1	9.4	2.0
sum(rec)	6	7	1	9.3	2.1
sum(iter)	5	11	1	12.6	2.3
reverse	7	11	3	22.0	3.7
append	7	12	3	22.3	4.3
copy	13	23	3	101.5	15.5
delete	15	60	35	83.3	10.8
Trees					
height	8	7	3	26.7	4.2
size	5	7	1	11.4	2.5
find	5	12	2	20.9	3.2
mirror	6	16	1	24.2	5.6
dealloc	14	33	1	33.8	6.8
flatten(rec)	10	18	1	43.7	8.5
flatten(iter)	28	35	28	270.7	49.6
Schorr-Waite					
tree	14	91	116	105.1	14.4
graph	14	91	203	232.9	34.4

Table 1. Proof statistics

2.3 Experiments

The rest of this section describes a range of examples we have verified in Coq with our system. Overall statistics are presented in Table 1. Times were measured with the 64-bit Linux version of Coq 8.4pl2, on a laptop with a i7-3720QM processor and 1600Mhz memory. We describe the languages, predicates, and proof automation afterwards.

Size attempts to count content, ignoring comments, blank lines, and punctuation. We also ignore some fixed lines such as imports. The specification size includes any functions or predicates defined for a particular example in addition to the set of claims. The

proof size similarly includes any lemmas, tactics, and proof hint declarations in addition to the main body of the proof. The one-line proofs are those solved completely by our automation.

Proving time is measured by compiling the Coq file containing an example, and certificate checking time is measured by rechecking the resulting proof certificate file, while skipping checks of any included modules. Checking a proof certificate does not require proof search or executing proof tactic scripts.

The simple examples are those which do not use the heap. The minimum and maximum proofs required a hint to use standard lemmas about the min and max functions. The sum program adds numbers from 1 to n, and required an auxiliary lemma proving an arithmetical formula. We describe one example in each of the remaining categories in more detail.

2.4 Lists

The next group of examples deals with linked lists. We implement a list node as a record value containing the integer value of that node, and the address of the rest of the list (which is 0 to represent empty lists).

Here is a statement which returns a copy of an input list, leaving the input list unchanged.

```
if (x == 0)
  { return 0 }
else
  { y := alloc
  ; *y := {val = x->val, next = 0}
  ; iterx := x->next
  ; itery := y
  ; while (not (iterx == 0))
    { node := alloc
    ; *node := {val = iterx->val; next = 0}
    ; *itery := {val = itery->val; next = node}
    ; iterx := iterx->next
    ; itery := itery->next
    }
  ; return y
  }
```

Using abbreviated notation, the desired specification is

```
(<k> copy_code </k>
<store>... "p" |-> p ...</store>
<heap> asP hlist (rep_list A p), hrest </heap>
...)
 =>
(exists p2,
 <k> return p2 </k>
 <store> _ </store>
 <heap> rep_list A p2, hlist, hrest </heap>
 ...)
```

This says that if the code is executed in a state where the store binds program variable `p` to the address of a linked list in the heap holding the sequence of values in the abstract list `A`, it can only return with the address of a freshly-allocated copy of the list, and the rest of the heap unchanged.

For this program, repeating `rep_list A p` in the postcondition would not be a strong enough specification, because it would allow moving intermediate nodes in the input list, but the copy operation should be usable as a subroutine even in code that holds pointers to intermediate nodes in the input list. The `asP` pattern binds the variable `hlist` to the exact subheap that satisfies the pattern `rep_list A p`, which allows specifying that the input list is unchanged.

We have not implemented this abbreviated notation. Our complete specification also includes a claim about the loop, whose precondition asserts that a non-empty initial segment of the list has been copied, and `itery` holds the address of the last list node in the copied segment.

The complete proof script for this example is

```
Proof. list_solver.
rewrite app_ass in * |-.
list_run. Qed.
```

We see associativity of list append was not automatically applied. When the automated solver paused, it left a goal we abbreviate as

```
(<k> return v </k>
 <heap> rep_list ((A++[x])++y::B) v , H </heap>
 ...)
=>
(exists p2,
 <k> return p2 </k>
 <heap> rep_list (A++x::y::B) p2 , H </heap>
 ...)
```

The abstract list is described with an unexpectedly associated append, but the current state does satisfy the target. It was not necessary to automate reasoning about associativity or to manually complete this branch of the proof. After reassociating the expression membership in the target set can be shown automatically, and this is the first thing attempted when the `list_run` tactic resumes automation.

The three examples with three-line proof required this assistance with associativity. The delete example removes all copies of a value from a linked list. The specification defined the desired operation as a Coq function, and we did not automate reasoning about it.

The Bedrock[6] example `SinglyLinkedList.v` verifies a module defining length, reverse, and append functions on linked lists. It takes approximately 150 seconds to prove, and 50s to recheck. Our results in Table 1 may be an unfair comparison as Bedrock's language can only store a scalar value in a heap location and our code as presented in this section keeps an entire structure at an address. For a more even comparison we modified the program and specification to expect the value and next pointers at consecutive addresses. To ensure no undesired functional was used, we made a modified copy of our main language with records and built-in memory allocation removed, under the `bytewise` directory in our development. However, rather than costing performance, the ability to make a single-field update without copying the other field actually improved performance. The modified examples can be verified in respectively 6.5, 13, and 15 seconds. Rechecking these proof certificates take 1.7, 2.4, and 2.6 seconds.

2.5 Trees

The next data structure we deal with is a binary tree. Tree nodes are implemented as records containing a value and the addresses of left and right children.

Two examples flatten a binary tree into a linked list by a preorder traversal, deallocating the input tree.

One implementation is a recursive helper function which flattens a tree onto the front of a given list

```
(<k> flatten_code </k>
 <store>... "t" |-> t , "l" |-> l ...</store>
 <heap>rep_tree T t, rep_list A l, H</heap>
 ...)
=>
(exists v,
 <k> return v </k>
 <heap>rep_list (tree2list T ++ A) v, H</heap>
```

```
...)
```

Flattening is defined as a Coq function `tree2list`. This implementation was easily verified.

A more interesting implementation avoids recursion by using an explicit stack, implemented as a linked list of pointers to subtrees. To specify this list, the list predicate is generalized. Instead of just taking a list of integers, the generalized predicate takes a list of any type, plus a representation predicate for that type, and asserts that the values in the abstract list are represented by the corresponding numbers in the concrete list, along with some possibly-empty subheaps. Instantiating this with the representation predicate for trees gives a predicate `rep_gen_list rep_tree trees` for a stack of trees. Here is the full Coq specification of the loop:

```
loop_claim :
  forall c rest,
    kcell c
      = kra (KStmt tree_to_list_loop) rest ->
    (alloc_mark c > 0)%Z ->
  forall t l s tn ln sn,
    store c ~= "t" s|-> KInt t
            :* "l" s|-> KInt l
            :* "s" s|-> KInt s
            :* "tn" s|-> tn
            :* "ln" s|-> ln
            :* "sn" s|-> sn ->
  forall ts lv hrest,
    heap c |= rep_prop_list rep_tree ts s
           :* rep_list lv l
           :* hrest ->
  tree_to_list_spec c (fun c' =>
    exists l', exists store',
    kcell c' = rest
    /\ stk_equiv (stack c) (stack c')
    /\ store c' ~= "l" s|-> KInt l' :* store'
    /\ functions c ~= functions c'
    /\ (alloc_mark c' >= alloc_mark c)%Z
    /\ heap c' |=
      rep_list (trees2List (rev ts) ++ lv) l'
      :* hrest)
```

The proof of the height function required some assistance with the max function. The (exhaustive) find function required a manual case split on the results of looking for the target value in the left subtree. The recursive flatten function required more substantial assistance because we did not provide automation for the generalized list predicate.

2.6 Schorr-Waite

A standard example of complex invariants is the Schorr-Waite graph marking algorithm [22], which overwrites pointers to maintain a stack without using additional space but eventually restores the graph.

The code verified implements the algorithm as presented in [12]. This version is written for cons cells with two pointers, and is somewhat unusual in rotating pointers through all fields rather than only having one disturbed field at a time. This rotation ensures the first field of the current node is the next node to consider moving to, whether that move is descending to a child or ascending the implicit stack.

As with the earlier examples, heap data structures are handled by defining a representation predicate satisfied by a concrete subheap representing an abstract value. In particular, the predicate here describes a connected graph of cells reachable from a current pointer. To assert that the graph nodes are left in place, the abstract value also contains the exact address where each node is allocated.

Pgm ::= *FunDef**

FunDef ::= *Id* `(` *Id*$^*_{,}$ `)` `{` *Stmt* `}`

Exp ::= *Id* | *Int* | *Bool*
| `-` *Exp* | *Exp* `+` *Exp* | *Exp* `-` *Exp* | *Exp* `*` *Exp* | *Exp* `/` *Exp*
| `not` *Exp* | *Exp* `and` *Exp* | *Exp* `or` *Exp*
| *Exp* `<=` *Exp* | *Exp* `<` *Exp* | *Exp* `==` *Exp*
| *Id* `(` *Exp*$^*_{,}$ `)`
| `alloc` | `load` *Exp*
| *Exp* `.` *Id* | `build` *Map*

Stmt ::= `skip` | *Stmt* `;` *Stmt*
| `if` *Exp* `then` `{` *Stmt* `}` `else` `{` *Stmt* `}`
| `while` *Exp* `do` `{` *Stmt* `}`
| *Id* `:=` *Exp*
| `*` *Exp* `:=` *Exp* | `dealloc` *Exp*
| *Id* `(` *Exp*$^*_{,}$ `)` | `decl` *Id* | `return` *Exp* | `return`

Figure 2. HIMP syntax

The simpler proof shows that the given code correctly marks a tree, with no sharing or cycles. We extend this to general graphs by considering the tree resulting from a depth first traversal. Using dependent types we augment a tree with a list of earlier nodes, and allow explicit backreferences. The overall specification says the calling the marking function with a pointer to the root of an unmarked graph and a remaining heap will return with the connected graph marked but otherwise unchanged, and the remaining heap untouched. For verified garbage collection, we would prove a lemma that selecting a root divides any collection of cells into a connected graph and remaining unreachable nodes.

To specify the inner loop of the algorithm, we define the encoded stack and the untargeted part of the graph as a zipper[14] into the DFS tree.

This proof involves quite a lot of manual effort. However, it is almost entirely in manipulating our graph representation. As hoped, almost all execution steps are automated, even when predicates in a specification lack any automation.

A Schorr-Waite example in Bedrock [3] takes 22 minutes to prove and over 4 minutes to recheck. Our version proves in under 4 minutes and rechecks in 34s. Total file length is 564 in our system and 1408 lines in Bedrock, though some of the difference in time and space may come from the Bedrock proof reasoning directly about graphs as sets of nodes.

3. Example Languages and Predicates

The main language used in the examples is a structured imperative statement language, extended with functions, a heap, and records. We call it HIMP (from IMP with heap).

A configuration is a tuple with six slots. In addition to the currently executing code and current local variable environment, we add cells for a call stack, the heap, and a collection of function definitions. An increment-only counter is a simple deterministic way to choose fresh addresses.

For clarity of programs dealing with data structures, we allow records with named fields as values. An entire record may be stored in a heap location. Records may be nested. Fields and subrecords do not have their own addresses.

The syntax of the language is given in Figure 2. A program is simply a sequence of function definitions, which are transferred

[3] Obtained from https://github.com/duckki/bedrock-examples/ at commit 4086a4f63e57

Pgm ::= *FunDef**

FunDef ::= name : *Inst**

Inst ::= `Dup` n | `Roll` n | `Pop` | `Push` z | `BinOp` f | `While` *Inst* Inst**
| `If` *Inst* Inst** | `Load` | `Store` | `Call` name | `Ret`

Figure 3. Stack language syntax

to the table of functions before beginning execution by calling the function named "main". A function call first evaluates the arguments, and then pushes the current code and scope as a new stack frame, and starts executing the body of the function in a new scope initialized to map the formal variables to the argument values.

Heap addresses are values. Heap access is provided by a dereference expression '`load`' and a heap assignment statement `*` *Exp* `:=` *Exp*. The '`alloc`' expression allocates a fresh heap location, and the '`dealloc`' expression deallocates an existing location. Records access uses the field projection operator Exp `.` Id. The record construction expression '`build`' takes a map from field names to values and evaluates the expression to produce a record value.

The '`decl`' statement extends the local variable environment with a new uninitialized variable. No statement introduces a local scope, so it would suffice to allow variable declarations only at the start of functions. The remaining expressions and statements are entirely conventional.

3.1 Stack Language

We also implemented a simplified stack language, with syntax given in Figure 3. Here the local state is a stack of values rather than a map from identifiers, and it is not saved and restored when calling functions. However, the heap is still a map indexed by integer addresses, so we can implement data structures with the same memory layout as in other examples.

The argument to the '`Dup`' instruction is the natural number offset of an existing stack entry to copy. The argument to the '`Roll`' instruction is the number of further stack entries to lift over the top entry. The argument to `Push` is the value to push. The argument to `BinOp` is a binary operation to perform. To shorten the specification, the argument to `BinOp` can be any Coq function.

3.2 Semantic Styles and Automating Steps

The only part of a proof that directly involves the transition relation of a semantics is finding a successor for a configuration. To investigate the difficultly of automating these steps we defined a transition relation for a simple imperative language in three different styles: small-step[16], evaluation contexts[10], and K[17], and verified a simple loop.

A small-step semantics defines the transition relation inductively. For small expressions a successor can be found by resolution using the `auto` tactic, but the search depth must be increased and we may not be able to control backtracking.

An evaluation-contexts semantics avoids a recursive definition of the transition relation, but needs an auxiliary definition of contexts and a plugging function. Coq's native unification cannot decompose a term into context and redex to apply rules to configurations, so a custom tactic is required.

A K-style semantics extends the code in the configuration to a list of terms, and evaluates within subterms by having a transition that extracts the term to the front of the list, where it can be examined directly. This allows a non-recursive definition of transition, whose cases can be applied by unification. Given a tactic `domain_solver` which can solve the side conditions of rules, the proof tactic `econstructor(domain_solver)` will automatically use the first case of the transition relation whose side conditions (if any) can all be solved automatically.

3.3 Predicates

In general, a specification could use arbitrary predicates over configurations. Within this design space we based our approach on Matching Logic[20]. The most basic part of the matching logic approach is to provide for any term-forming operation a corresponding pattern forming operation. This matches a value if the value can be constructed by applying the term-former to values that respectively match the sub-patterns.

For maps, the join pattern takes two map predicates and accepts any map which is equivalent to the union of maps that respectively satisfy the predicates. The empty and item patterns are satisfied respectively by an empty map and a map with only the given binding.

A pattern existential quantifier matches a value if there is a choice of the bound variable for which the body pattern matches the value. The `constraint` patterns matches an empty heap, but only if a logical statement is true, and is used to put constraints on existentially quantified variables.

Using these basic patterns, we define the heap representation of linked lists and binary trees by recursion over the abstract value. The representation of list segments is

```
Fixpoint rep_seg (val : list Z) (tailptr : Z)
    (p : Z) : MapPattern k k :=
 (match val with
    | nil => constraint (p = tailptr%Z)
    | x :: xs => constraint (p <> 0%Z) :*
      (existP p',
      (p h|-> KStruct (Struct
         ("val" s|-> KInt x
         :* "next" s|-> KInt p')%Map))
      :* rep_seg xs tailptr p')
  end)%pattern.
```

An empty list segment is represented at address p in a heap only when the heap is empty and p is actually equal to the final target address. A non-empty list segment is represented in a heap when p is the address of a linked list node in the heap which carries the first value from the segment, and the next pointer of that node represents the rest of the list segment in the rest of the heap.

The generalized list predicate mentioned in Section 2.5 is

```
Fixpoint rep_prop_list {A}
  (P : A -> Z -> MapPattern k k) (val : list A)
  (p : Z) : MapPattern k k :=
  (match val with
    | nil => constraint (p = 0%Z)
    | x :: xs => constraint (p <> 0%Z) :*
        existP v, existP next,
        (p h|-> list_node v next
        :* P x v
        :* rep_prop_list P xs next)
  end)%pattern.
```

The value in the list node is existentially quantified, and `P x v` may match a subheap.

Trees are defined similarly, but recurse into two subtrees.

```
Fixpoint rep_tree t p : MapPattern k k :=
  match t with
    | Leaf => constraint (p = 0)%Z
    | Node v l r => constraint (p <> 0)%Z
      :* existP pl, existP pr,
       (p h|-> tree_node v pl pr
        :* rep_tree l pl :* rep_tree r pr)
  end%pattern.
```

With representation predicates defined in this style, whenever the abstract value is partially known, evaluation simplifies the predicate and exposes primitive heap assertions. Because of this, most of the effort for making use of representation predicates is the reusable work of writing tactics for handling the basic heap predicates.

Two cases which require some further automation are refining the abstract value when a case split fixes information about the concrete heap, and picking abstract values to match a claim of the specification to the current state.

4. Proof Automation

An essential component for any practical verification approach is proof automation. As the example proof in the previous section demonstrated, every computational step is reflected in the proof, but most are so simple they should not require human attention.

In particular, most steps do not introduce new symbolic variables, introduce new facts about existing variable, or require proving any facts about existing variables (beyond trivialities such as showing by simple evaluation that some predicate holds for some term regardless of the values of the symbolic variables).

The larger conceptual units in a proof, such as taking a step of computation, correspond to some sequence of lower-level proof steps, possibly leaving some further subgoal. A procedure implementing such an operation is known as a proof *tactic*. In particular, we implement our proof automation using the Ltac[8] tactic language in Coq.

Our results were obtained with a simple overall heuristic, based on the heuristic of the MatchC[18] system. The general idea is to handle a proof goal of supporting a claim (c, P) in a way that makes as much progress along program execution as possible. Here are the possible steps, in order of expected progress. First, immediately show $c \in P$ if possible. Second, if some claim of the specification covers c, take a step by transitivity. If no claim applies, take one computational step according to the rules of the language. Finally, if none of those cases apply, then it is necessary to make a case distinction breaking c down into smaller cases, so some execution rules (or other steps) apply.

The last case arises mostly from If-statements and other conditional constructs, where a symbolic boolean value given by an expression like $x > 0$ must often be split into a true and false case to allow execution to proceed. Automatically completing a proof is attempted simply by repeatedly trying to make progress as above as many times a possible, recursing down all alternatives of a case split.

As we implement this strategy in an interactive proof assistant, it is easy to integrate manual assistance simply by having the tactics leave a subgoal for the user. The user may resume automation by reinvoking the appropriate tactic.

4.1 Reusability

The core logical content of our approach is a theorem that can be proved once in a given proof assistant and reused for any language. The procedural knowledge in our proof automation tactics unfortunately require more customization, and contribute to the incremental cost of supporting new language or programs. We require a definition of the transition relation, but consider that part of the cost of having a formal semantics of a language at all, rather than counting it as a cost of supporting verification.

The main loop of our heuristic can be implemented as a parameterized tactic, so only tactics performing or used by the individual steps need to be customized. Further more, defining semantics and specifications in a particular style (as non-recursive inductively-defined predicates) allows a generic tactic to select appropriate claims for transitivity and rules for execution, given only a tactic that can solve the hypotheses of the appropriate constructors. With non-recursive definitions of transitions and specifications, these hypotheses only involve reasoning about the domains making up individual configurations, which is independent of the number of rules in the semantics or number of clauses in the specification being proved.

This domain reasoning involves three sorts of predicates. Some deal with domains such as integers and finite maps which are general enough that many semantics can share the same definitions. Others deal with language specific predicates such as checking whether a subexpression is considered to be completely evaluated. However, to have an executable semantics these predicates must return concrete results on concrete terms, and we have found that the symbolic states in our example are sufficiently concrete that expression simplification reduces these conditions to simple boolean equalities, so these first two categories can be handled in reusable ways. Finally, some predicates are specific to particular examples, such as the definition of the heap representation of linked lists and trees that we mention in Section 3.3.

A necessary subtlety in the automation of transitivity steps is recognizing when a claim of the specification might be applicable even though it could not be automatically applied. For example, if the specification of a loop has a precondition which is not handled automatically, falling through to taking a symbolic execution into another iteration of the loop would result in proof automation diverging. For our example languages it suffices to check if the currently executing code matches that in the specification.

The remaining case of the overall heuristic is making a case split. This is generally necessary only at a configuration where an if-expression or other conditional construct is examining a boolean value which is know only as a symbolic expression such as $x > 0 \wedge y \leq 10$. We currently rely on handwritten patterns to recognize these cases and extract the expression to split on. It may be possible to generate these patterns by anti-unification. Some additional proof handling is necessary to turn boolean facts into more useful decomposed hypotheses, such as $(x > 0 \wedge y \leq 10) = \mathrm{true}$ into $x > 0$ and $y \leq 10$.

Overall, the work necessary to implement useful proof automation is almost independent of the number of rules of the language semantics or specification involved, but depends only on the number and complexity of the domain predicates used in defining the language language and the specification to be proved.

5. Soundness and Completeness

We present our results without generalizing beyond the setting of monotone functions on a complete lattice (with completeness used only to justify the existence of certain fixpoints). In particular, the lattice of subsets of the set of all claims. Some readers may find it helpful to regard preorders as categories and draw on intuitions from recursion schemes for algebraic data types, but we do not need the generality and confine categorical language to footnotes [4].

Results which depend only on the lattice structure use lattice notation rather than set notation, such as $x \leq y$ rather than $x \subseteq y$, $x \vee y$ instead of $x \cup y$, etc. We lift join and union pointwise to operators, so $(F \vee G)(x) = F(x) \vee G(x)$. A least fixpoint expression $\mu X \,.\, F(X)$ denotes the least fixpoint of the operation $X \mapsto F(X)$, and is only used when we know or immediately show F monotone.

For a monotone function F, μF and νF respectively denote the least and greatest fixpoints of F. An *F-closed* set is a set A such that $F(A) \leq A$. Dually, an *F-stable* set is a set A such that $A \leq F(A)$. A defining property of least fixpoints is *induction*: μF is a subset of any F-closed set. Dually, a defining property of greatest fixpoints is *coinduction*: any F-stable set is a subset of νF.

[4] Monotone functions are the functors of a preorder category.

The Knaster-Tarski Theorem [23] states that any monotone function on a complete lattice has a complete lattice of fixpoints, and as a corollary that least and greatest fixpoints (not necessarily distinct) exist.

A *closure operation* on a lattice is a monotone function C satisfying the additional properties of being *extensive* ($\forall X, X \leq C(X)$) and *idempotent* ($\forall X, C(C(X)) = C(X)$).[5]. Now we are finished recalling preliminaries.

5.1 Soundness

First, we justify the name of the F-*closure* operation (defined immediately before Lemma 1):

Lemma 2. *For any monotone F, F^* is the least closure operator at least as large as F.* [6]

Proof. First, establish that F^* is in fact a closure operator.

For monotonicity, fix $A \leq B$. Then $F^*(A) \leq F^*(B)$ follows by induction from

$$F(F^*(B)) \vee A \leq F(F^*(B)) \vee B = F^*(B)$$

For extensiveness, $X \leq F(F^*(X)) \vee X = F^*(X)$

For idempotence, we need $F^*(F^*(X)) = X$. Extensiveness gives $F^*(X) \leq F^*(F^*(X))$ so it suffices to show $F^*(F^*(X)) \leq F^*(X)$. This follows by induction from

$$\begin{aligned} F(F^*(X)) \vee F^*(X) &\leq F(F^*(X)) \vee X \vee F^*(X) \\ &\leq F^*(X) \vee F^*(X) \\ &= F^*(X) \end{aligned}$$

Second, show that $F \leq F^*$. Making use of extensiveness,

$$F(X) \leq F(F^*(X)) \leq F(F^*(X)) \vee X = F^*(X)$$

Third, show F^* is the least such closure operator. Suppose G is another closure operator with $F \leq G$. Fixing X, $F^*(X) \leq G(X)$ holds by induction from $F(G(X)) \vee X \leq G(G(X)) \vee G(X) \leq G(X)$ □

One corollary is that the $-^*$ operation is itself monotone. If $F \leq G$, then G^* is a closure operator and $F \leq G \leq G^*$ so $F^* \leq G^*$.

Now we are ready to state and prove our key coinduction theorem. We define G *is sound for* F to mean F and G are monotone and $\forall X, G(F(X)) \subseteq F(G^*(X))$.

Theorem 1 (Coinduction with Rules)**.** *If G is sound for F, then for any X, $X \leq F(G^*(X))$ implies $X \leq \nu F$.*

Proof. As G^* is a closure operation $X \leq G^*(X)$, so it suffices to show $G^*(X) \leq \nu F$. This follows by coinduction from $G^*(X) \leq F(G^*(X))$, which follows by induction from

$$G(F(G^*(X))) \vee X \leq F(G^*(X)).$$

By idempotence this is equivalent to the instance

$$G(F(G^*(X))) \vee X \leq F(G^*(G^*(X)))$$

of the hypothesis that G is sound for F. □

Note that F is always sound for F, and that if G and H are sound for F then $G \vee H$ is also sound for F by monotonicity of $-^*$ and the inclusions $G, H \leq G \vee H$. So, we say that G is a *valid derived rule for* F if $G \vee F$ is sound for R, and note that this property is also preserved under union.

Lemma 3. $\mathrm{valid}_R = \nu\, \mathrm{step}_R$

Proof. First we show valid_R is step_R-stable. Suppose $(c, P) \in \mathrm{valid}_R$. If $c \in P$, then $(c, P) \in \mathrm{step}_R(\mathrm{valid}_R)$ immediately. Otherwise either c can take an infinite number of steps in R, or reaches a successor in P. In either case there is a one-step successor d with $c \to_R d$, and also $(d, P) \in \mathrm{valid}_R$, so $(c, P) \in \mathrm{step}_R(\mathrm{valid}_R)$.

Next we show valid_R is the largest step_R-stable set. Let X be a set with $X \subseteq \mathrm{step}_R(X)$. For any $(c, P) \in X$, we make a case distinction based on whether c can take an infinite number of steps in R. If so then $(c, P) \in \mathrm{valid}_R$ already. If not, then relation R is well-founded on the set of configurations reachable from c. By well-founded induction we can assume that $(d, P) \in \mathrm{valid}_R$ for any d with $c \to_R d$ and $(d, P) \in X$. As $(c, P) \in X \subseteq \mathrm{step}_R(X)$ we have either $c \in P$, in which case $(c, P) \in \mathrm{valid}_R$ immediately, or that there exists a d with $c \to_R d$ and $(d, P) \in X$. In this case $(d, P) \in \mathrm{valid}_R$ by the inductive hypothesis. By the termination assumption, d reaches a configuration in P. Then c reaches a configuration in P as well, by following the step $c \to_R d$ with the same path. □

By Lemma 3, instantiating Theorem 1 using step_R for F and G gives Lemma 1. As we saw in Section 2.2, this allows multiple steps of symbolic execution while proving a clause of the specification, and appealing coinductively to claims which exactly satisfy the current goal. For larger examples we need to be able to continue the proof after appealing to the specification of loops or subroutines. As with multiple-step symbolic execution, we can capture this reasoning principle as a function for use with Theorem 1.

Lemma 4. *Let* trans *be the function on claims defined as*

$$\mathrm{trans}(X) = \{(c, P) \mid \exists Q.(c, Q) \in X \wedge \forall d \in Q, (d, P) \in X\}.$$

Then $\mathrm{trans} \cup \mathrm{step}_R$ *is sound for* step_R*, for any R.*

Proof. Suppose $(c, P) \in \mathrm{trans}(\mathrm{step}_R(X))$. Then there exists some Q such that $(c, Q) \in \mathrm{step}_R(X)$ and for any $d \in Q$, $(d, P) \in \mathrm{step}_R(X)$. Now, either $c \in Q$ or there is some d with $c \to_R d$ and $(d, Q) \in X$. In the first case, $c \in Q$ so $(c, P) \in \mathrm{step}_R(X)$, which is a subset of $\mathrm{step}_R((\mathrm{trans} \vee \mathrm{step}_R)^*(X))$. In the second case we have $(d, P) \in \mathrm{trans}(X \vee \mathrm{step}_R(X))$, so $(c, P) \in \mathrm{step}_R(\mathrm{trans}(X \vee \mathrm{step}_R(X)))$, which is also a subset of $\mathrm{step}_R((\mathrm{trans} \vee \mathrm{step}_R)^*(X))$. □

5.2 Relative Completeness

We are not presenting a syntactic proof system, so we carefully consider how to formulate relative completeness. Intuitively, we want to formulate the condition that any desired set X of valid claims can be proven with our approach. This satisfied if $X \subseteq \mathrm{valid}_R$ is the conclusion of an application of Theorem 1. However, this condition is too strict. Proving of a Hoare triple may necessarily require providing additional loop invariants. Likewise, proving a specification of interest in our system may require also making claims about loops and auxiliary functions. In our system this is done by enlarging the original set of claims. The correct notion of relative completeness is thus to ask whether the desired set X of valid claims is contained in some larger set S for which we can conclude $S \subseteq \mathrm{valid}_R$ as an application of Theorem 1.

For any set X and any choice of G we can in fact take the set valid_R of all true claims. As part of showing valid_R is a fixpoint we established that $\mathrm{valid}_R \subseteq \mathrm{step}_R(\mathrm{valid}_R)$. By monotonicity and extensiveness,

$$\mathrm{valid}_R \subseteq \mathrm{step}_R(F^*(\mathrm{valid}_R))$$

This leaves only the goal of showing $X \subseteq S = \mathrm{valid}_R$.

One might complain that this is trivial, but then one should complain all the more about a conventional relative completeness

[5] This is a specialization of the definition of a monad to the preorder category.

[6] "least closure operator" is a specialization of "free monad".

result. Any general purpose specification language is necessarily undecidable, so no syntactic proof system can be complete. Instead, any relatively complete proof system has a rule with a hypothesis of semantic validity in some predicate language, and the relative completeness argument consists of tediously showing how to Gödel-encode validity into the predicate language, and showing that the rules of the proof system are strong enough to make use of such a complicated predicate. We obtain an equally strong result.

6. Related Work

A number of prominent tools such as Why[3], Boogie[1, 15], and Bedrock[6, 7] provide program verification for a fixed language, and support other languages by translation if at all. For example, Frama-C and Krakatoa respectively attempt to verify C and Java by translation through Why, Spec# and Havoc respectively verify C# and C by translation through Boogie. We are not aware of soundness proofs for these translations.

All of these systems are based on a verification condition generator for their programming language. Bedrock is closest in architecture and guarantees to our system, as it is implemented in Coq and verification results in a Coq proof certificate that the specification is sound with respect to a semantics of the object language. Bedrock supports dynamically created code, and modular verification of higher-order functions, which we have not yet attempted. Bedrock also makes more aggressive attempts at complete automation, which seem to be quite effective, but costs increased runtime. Most fundamentally, Bedrock is built around a verification condition generator for a fixed target language.

In sharp contrast to the above approaches, we believe that a small-step operational semantics suffices for program verification, without a need to define any other semantics, or verification condition generators, for the same language. A language-independent, sound and (relatively) complete coinductive proof method then allows us to verify reachability properties of programs using directly the operational semantics. Both the required human effort and the performance of the verification task compare favorably with the closest approach based on the above, Bedrock, at the same time providing the same high confidence in correctness: the trust base consists of the operational semantics of the language only.

A closely related approach to program verification based on operational semantics is reachability logic [21]. Reachability logic offers a language-independent proof system for verifying reachability properties given a definition of an operational semantics as a collection of rewrite rules. Our work in this paper resulted, in fact, from an attempt to understand the soundness proof of the reachability logic proof system in terms of coinduction.

A categorical generalization of our key lemma was introduced as a recursion scheme in "Recursion Schemes from Comonads"[24]. The titular recursion scheme defines functions from an initial algebra, and is parameterized also over a comonad satisfying a distributive law. Dualizing the construction to get "Corecursion Schemes from Monads" (named λ-coiteration in [2]) and specializing to preorder categories results in a lemma for showing that sets are contained in a greatest fixpoint. Choosing to use a free monad gives our Theorem 1, except with the hypothesis

$$\forall X, G^*(F(X) \subseteq F(G^*(x))$$

This is however equivalent to our condition

$$\forall X, G(F(X) \subseteq F(G^*(x))$$

It seems a number of weaker results are folklore. For example, the Isabelle standard library includes a lemma

$$mono(f) \wedge A \subseteq f(\mu x \,.\, f(x) \cup A \cup \nu f) \implies A \subseteq \nu(f)$$

which is an instance of our rule with the choice

$$g(x) = f(x) \cup \nu f$$

7. Future Work

We have not used our coinductive approach for verifying higher-order specifications. Verifying code that accepts objects or closures generally requires preconditions making requirements about the behavior of calling into those arguments. In some cases (e.g, when all subclasses are known in advance) this can be split into a limited set of allowed arguments and individual proofs. For the general case, however, a specification may need to require that a whole set of claims about the arguments are valid. We believe our approach can be extended to such cases by parameterizing higher-order specifications over set(s) of claims, and replacing "is valid" in such preconditions with set membership.

The language-independence of our approach may be particularly useful to allow verifying programs against slightly modified semantics. Augmenting a semantics with cost counting might allow proofs about performance or memory costs in addition to simple functional correctness. Simply tracking the number of execution steps may suffice for simple realtime systems. Modifying the semantics to become stuck when such a counter expires gives an easy approach to total correctness. We could also extend a language semantics with descriptions of a surrounding system, such as operating system services or network servers.

References

[1] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Proceedings of the 4th International Conference on Formal Methods for Components and Objects*, FMCO'05, pages 364–387, Berlin, Heidelberg, 2006. Springer-Verlag. ISBN 3-540-36749-7, 978-3-540-36749-9. doi: 10.1007/11804192_17. URL `http://dx.doi.org/10.1007/11804192_17`.

[2] Falk Bartels. *On Generalised Coinduction and Probabilistic Specification Formats: Distributive Laws in Coalgebraic Modelling*. PhD thesis, Vrije Universiteit Amsterdam, 2004.

[3] François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. Why3: Shepherd your herd of provers. In *Boogie 2011: First International Workshop on Intermediate Verification Languages*, pages 53–64, Wrocław, Poland, August 2011. URL `http://proval.lri.fr/publications/boogie11final.pdf`.

[4] Martin Bodin, Arthur Charguéraud, Daniele Filaretti, Philippa Gardner, Sergio Maffeis, Daiva Naudziuniene, Alan Schmitt, and Gareth Smith. A trusted mechanised javascript specification. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 87–100, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2544-8. doi: 10.1145/2535838.2535876. URL `http://doi.acm.org/10.1145/2535838.2535876`.

[5] Denis Bogdănaş and Grigore Roşu. K-Java: A Complete Semantics of Java. In *Proceedings of the 42nd Symposium on Principles of Programming Languages (POPL'15)*. ACM, 2015.

[6] Adam Chlipala. Mostly-automated verification of low-level programs in computational separation logic. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 234–245, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0663-8. doi: 10.1145/1993498.1993526. URL `http://doi.acm.org/10.1145/1993498.1993526`.

[7] Adam Chlipala. The bedrock structured programming system: Combining generative metaprogramming and hoare logic in an extensible program verifier. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, pages 391–402, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2326-0. doi: 10.1145/2500365.2500592. URL `http://doi.acm.org/10.1145/2500365.2500592`.

[8] David Delahaye. A tactic language for the system coq. In *Proceedings of the 7th International Conference on Logic for Programming and Automated Reasoning*, LPAR'00, pages 85–95, Berlin, Heidelberg, 2000. Springer-Verlag. ISBN 3-540-41285-9. URL http://dl.acm.org/citation.cfm?id=1765236.1765246.

[9] Chucky Ellison. *A Formal Semantics of C with Applications*. PhD thesis, University of Illinois, July 2012.

[10] Matthias Felleisen and Robert Hieb. The revised report on the syntactic theories of sequential control and state. *Th. Comp. Sci.*, 103(2):235–271, 1992.

[11] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995. ISBN 0-201-63361-2.

[12] David Gries. The schorr-waite graph marking algorithm. *Acta Informatica*, 11(3):223–232, 1979. ISSN 0001-5903. doi: 10.1007/BF00289068.

[13] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, October 1969. ISSN 0001-0782. doi: 10.1145/363235.363259. URL http://doi.acm.org/10.1145/363235.363259.

[14] Gérard Huet. The zipper. *Journal of Functional Programming*, 7:549–554, 9 1997. ISSN 1469-7653. URL http://journals.cambridge.org/article_S0956796897002864.

[15] K. Rustan M. Leino. This is boogie 2. Technical report, Microsoft Research, June 2008. URL http://research.microsoft.com/apps/pubs/default.aspx?id=147643.

[16] Gordon D. Plotkin. A structural approach to operational semantics. *Journal of Logic and Algebraic Programming*, 60-61:17–139, 2004. doi: 10.1016/j.jlap.2004.05.001. Original version: University of Aarhus Technical Report DAIMI FN-19, 1981.

[17] Grigore Roşu and Traian Florin Şerbănuţă. K overview and SIMPLE case study. *Electronic Notes in Theoretical Computer Science*, 304 (0):3 – 56, 2014. ISSN 1571-0661. doi: 10.1016/j.entcs.2014.05.002. URL http://www.sciencedirect.com/science/article/pii/S1571066114000383. Proceedings of the Second International Workshop on the K Framework and its Applications (K 2011).

[18] Grigore Roşu and Andrei Ştefănescu. Checking reachability using matching logic. In *Proceedings of the 27th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'12)*, pages 555–574. ACM, 2012.

[19] Grigore Roşu and Andrei Ştefănescu. From hoare logic to matching logic reachability. In *Proceedings of the 18th International Symposium on Formal Methods (FM'12)*, volume 7436 of *LNCS*, pages 387–402. Springer, Aug 2012. doi: 10.1007/978-3-642-32759-9_32.

[20] Grigore Roşu, Chucky Ellison, and Wolfram Schulte. Matching logic: An alternative to Hoare/Floyd logic. In Michael Johnson and Dusko Pavlovic, editors, *Proceedings of the 13th International Conference on Algebraic Methodology And Software Technology (AMAST '10)*, volume 6486 of *Lecture Notes in Computer Science*, pages 142–162, 2010. ISBN 978-3-642-17795-8.

[21] Grigore Roşu, Andrei Ştefănescu, Ştefan Ciobâcă, and Brandon M. Moore. One-path reachability logic. In *Proceedings of the 28th Symposium on Logic in Computer Science (LICS'13)*, pages 358–367. IEEE, June 2013.

[22] H. Schorr and W. M. Waite. An efficient machine-independent procedure for garbage collection in various list structures. *Commun. ACM*, 10(8):501–506, August 1967. ISSN 0001-0782. doi: 10.1145/363534.363554. URL http://doi.acm.org/10.1145/363534.363554.

[23] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5(2):285–309, 1955. URL http://projecteuclid.org/euclid.pjm/1103044538.

[24] Tarmo Uustalu, Varmo Vene, and Alberto Pardo. Recursion schemes from comonads. *Nordic J. of Computing*, 8(3):366–390, September 2001. ISSN 1236-6064. URL http://dl.acm.org/citation.cfm?id=766517.766523.

[25] Y. Yang, G. Gopalakrishnan, G. Lindstrom, and K. Slind. Nemos: a framework for axiomatic and executable specifications of memory consistency models. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, pages 31–, April 2004. doi: 10.1109/IPDPS.2004.1302944.