Coinductive Program Verification

Thesis Proposal

Brandon Moore

University of Illinois

1 Problem Description

Every programming language should have a formal semantics, and sound verification tools. Giving an operational semantics is hardly more difficult than writing an interpreter, but traditional approaches to enabling program verification, such as defining and proving sound an axiomatic semantics, require considerably more sophistication.

For maximum confidence, a program verification tool should be proven sound, or at least produce proof certificate in a standard proof system showing that a validated formal semantics for the language implies that the program has the claimed properties.

The standard approach is to define an operational semantics for a language, and then define and prove equivalent an alternate semantics more suitable for verification, such as an axiomatic semantics.

We believe it is unnecessary to define language-specific proof systems, that certain simple coinductive semantics can make reasoning about programs directly in terms of an operational semantics simple, effective, and powerful.

2 Proposed Approach

I propose an approach to verifying partial correctness properties of programs by reasoning directly on the transition relation given by an operational semantics.

The key idea is to express the set of *all* correct claims as a greatest fixpoint. Then the coinduction principle for showing that a set is a subset of a greatest fixpoint can be used to prove a particular set of claims about a particular program are a subset of the set of true claims, and therefore true. The raw coinduction principle requires rather unwieldy evidence, so I add a notion of derived rule, and justify several derived rules which permit simple proofs.

The three parts of this approach are the particular choice of claims, the construction of truth as a greatest fixpoint, and the lemmas for coinduction with derived rules.

As a running example, I use a loop computing triangular numbers, in a simple non-procedural imperative language.

```
while (n > 0) {s = s + n; n =  n - 1}
```

Letting `TRI` abbreviate the loop, a valid Hoare triple is

$\{\mathrm{s} \ = \ s_0 \, , \mathrm{n} \ = \ n_0 \, , n_0 \geq 0\} \mathrm{TRI} \{\mathrm{s} \ = \ s_0 + \sum_{i=0}^{n_0} i\}$

With no side effects in expressions, executing an assignment step or resolving a conditional is a basic computational step, and a configuration can be simply a pair of code and a partial function from identifiers to numbers.

2.1 Reachability Claims

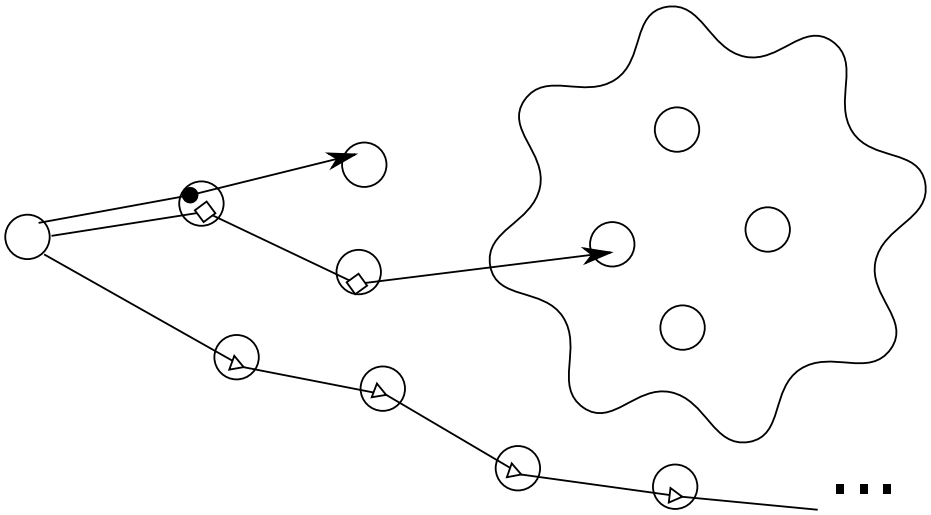

Figure 1. Reachability

I take as an elementary claim a pair (x, P) of a state and a set of states, read "x reaches P" for a "partially-correct" notion of reachability. In particular, x reaches P if it reaches after a finite number of steps in R some state $y \in P$, but also if x diverges. This is illustrated in fig. 1: the top path isn't evidence that the leftmost state reaches the circled region, but the middle path shows it by reaching the target in a finite number of steps, and the bottom shows it by diverging.

For nondeterministic languages, it may be better to use an "all-path" semantics, where the interpretation of a claim (x, P) is that every path from x that terminates, i.e. reaches a state with no successors, encountered somewhere a state in P. This can also be expressed as a greatest fixpoint, but I haven't worked with it as much.

This elementary notion of claims can be seen as a distillation of the notion in Reachability Logic[14]. There a partial correctness claim is represented as reachability rule, which is a pair

$$\varphi \Rightarrow \varphi'$$

of predicates on configurations. A reachability rule is true if every configuration γ satisfying φ under some valuation ρ, $\gamma, i.e. \rho \models \varphi$, either γ diverges or reaches in the transition system another configuration γ' with $\gamma', \rho \models \varphi'$. So, asserting a reachability rule is equivalent to asserting a set of claims

$$\{(\gamma, \{\gamma' \mid \gamma', \rho \models \varphi'\}) \mid \gamma, \rho \models \varphi\}$$

Work on reachability logic has shown that reachability rules are as expressive as Hoare triples for a basic imperative language [15], and given a "Matching Logic"[13] notation which makes predicates over configurations reasonably legible. A user friendly verifier should probably add some language specific abbreviations, e.g. MatchC[13] allows variable names to be used in formulas to stand for the value to which that variable is bound in the local environment of a state, and fills in the code component of a claim based on where in the program annotations are written.

The specification of the example loop is captured by

$$\langle \texttt{TRI}; R, \texttt{s} \mapsto s, \texttt{n} \mapsto n, S\rangle \wedge n \geq 0 \Rightarrow \langle R, \texttt{s} \mapsto s + \sum_{i=0}^{n} i, \texttt{n} \mapsto \_, S\rangle$$

2.2 Greatest Fixpoint

Let *cfg* be the set of configurations, $R \subseteq$ *cfg* $\times$ *cfg* the one step transition relation, and *reaches* $\subseteq$ *cfg* $\times \mathcal{P}$(*cfg*) the set of all true reachability claims. This is the greatest fixpoint of an operator on relations $step : \mathcal{P}(cfg \times \mathcal{P}(cfg)) \to \mathcal{P}(cfg \times \mathcal{P}(cfg))$, defined as follows

$$step[X] = \{(x, P) \mid x \in P \vee \exists y. x \; R \; y \wedge (y, P) \in X\}$$

This gives a coinduction principle:

$$X \subseteq step[X] \implies X \subseteq reaches$$

Unfortunately this requires mentioning each computation step. That is, for any claim (x, P) in a stable set X of claims, either $x \in P$ already, or there must exist a one-step successor y of x so that (y, P) is also a claim in X.

For example, to prove the example loop by direct coinduction, the set X of claims must include pairs where the initial configuration has code in each of the forms `TRI`; R, and `s = s + n;n = n - 1;TRI`; R, and `n = n - 1;TRI`; R, and R, rather than just of the initial form `TRI`; R.

2.3 Derived Rules

Directly giving a stable set means each claim can be supported by finding another claim in the set mentioning an immediate successor of each given claim, unless the initial point is already in the target set. To solve this, I introduce a semantic notion of a valid proof rule.

A first step toward this is a standard lemma (or at least one included in the Isabelle standard library) on greatest fixpoints. Given a set $X \subseteq A$ and a monotone function $F : \mathcal{P}(A) \to \mathcal{P}(A)$, to show that X is included in the greatest fixpoint νF of F, it suffices to take the least fixpoint $\mu S.F(S) \cup X$ of the operation $S \mapsto F(S) \cup X$, and show that

$$X \subseteq F[\mu S.F(S) \cup X]$$

This can be proved by way of a lemma that

$$X \subseteq F[\mu S.F(S) \cup X] \implies (\mu S.F(S) \cup X) \subseteq F(\mu S.F(S) \cup X)$$

Setting F to the *step* operator for verification, this means a set X of claims is supported if for each claim $(x, P) \in X$ there is a successor y after any number of steps $x\ R^*\ y$ which is either in the target, $y \in P$ or the subject of another claim in X, $(y, P) \in X$.

For the example loop, this makes the set containing just the translation of the desired reachability rule already a sufficient set of claims:

$$\begin{aligned} X_{\texttt{TRI}} = &\{(\langle \texttt{TRI}; R, \mathtt{s} \mapsto s, \mathtt{n} \mapsto n, S\rangle, \{\langle R, \mathtt{s} \mapsto s + \sum_{i=0}^{n} i, \mathtt{n} \mapsto n', S\rangle \mid n' \in \mathbb{Z}\}) \\ &\mid s, n \in \mathbb{Z}, n \geq 0, R \in \text{Stmt}, S \in (\text{Var} \setminus \{\mathtt{s}, \mathtt{n}\}) \hookrightarrow \mathbb{Z}\} \end{aligned}$$

Here the initial configurations with $n = 0$ can take one step to leave code R and a suitable state, so that portion of the set of claims is in $\mathit{step}[\mathit{step}[X]]]$. Code with $n > 0$ takes three steps to enter the loop and execute the two assignments, reaching a state whose code is again `TRI` but with a different store - however, checking that $\sum_{i=0}^{n} i = n + \sum_{i=0}^{n-1}$ shows this is another pair in the original set of claims. So, these states are all in $\mathit{step}[\mathit{step}[\mathit{step}[X]]]$. Between these cases, $X \subseteq \mathit{step}[\mu S.\, \mathit{step}(S) \cup X]$.

To generalize further, think of a proof rule as a function which takes a set of known claims, and returns the set of claims that can be produced by applying the rule using the assumed claims as hypotheses. This action is just a monotone function on sets of claims, so we can consider any monotone function on sets of claims as a (potentially invalid) rule, or collection of rules.

Then the desired claims X can be closed under the action of this rule by taking a least fixpoint of the mapping $D \mapsto X \cup \mathit{step}[D] \cup F[D]$ (*step* is included explicitly because it's always sound). Call this operator $\mathit{derived} : (\mathcal{P}(\mathit{cfg} \times \mathcal{P}(\mathit{cfg})) \to \mathcal{P}(\mathit{cfg} \times \mathcal{P}(\mathit{cfg}))) \times \mathcal{P}(\mathit{cfg} \times \mathcal{P}(\mathit{cfg})) \to \mathcal{P}(\mathit{cfg} \times \mathcal{P}(\mathit{cfg}))$, defined by

$$\mathit{derived}[F, X] = \mu D.X \cup \mathit{step}[D] \cup F[D]$$

Following the earlier lemma, say X is stable under rules F if $X \subseteq \mathit{derived}[F, X]$. The most general notion of a rule F being sound is that it showing some set X is stable under F suffices to prove that X is valid,

$$\forall X.X \subseteq \mathit{step}[\mathit{derived}[F, X]] \implies X \subseteq \mathit{reaches}$$

This condition provides little guidance on how to prove a set of rules is actually valid, so it's useful to consider a stronger condition. The obvious way to show $X \subseteq \mathit{reaches}$ is to exhibit a *step*-stable superset of X, and one might hope that $\mathit{derived}[F, X]$ would itself be such a set. This gives a notion of validity

$$\forall X.X \subseteq \mathit{step}[\mathit{derived}[F, X]] \implies \mathit{derived}[F, X] \subseteq \mathit{step}[\mathit{derived}[F, X]$$

which is equivalent to

$$\forall X.X \subseteq \mathit{step}[\mathit{derived}[F, X]] \implies F[\mathit{step}[X]] \subseteq \mathit{step}[\mathit{derived}[F, X]$$

The next question is whether combining two sets of valid rules gives a set of valid rules, under a pointwise union

$$(F + G)[X] = F[X] \cup G[X]$$

Dropping the hypothesis of the previous validity condition gives a stronger condition which is compositional in this sense:

$$\forall X.F[\mathit{step}[X]] \subseteq \mathit{step}[\mathit{derived}[F, X]]$$

This validity condition is satisfied in particular by transitivity and weakening rules. These are sufficient the proof search tactic which was used with great success in MatchC: For each claim $(x, P) \in X$, try to demonstrate the inclusion by first checking whether $x \in P$, if that fails by looking for another specification $(x, Q) \in X$ and applying it by transitivity leaving subgoals of showing that (q, P) is supported for any $q \in Q$ (to avoid stepping into code abstracted over by a specification), and if that isn't possible finding the successor state xRy and reducing by *step* to the goal (y, P).

Transitivity is useful for example in verifying the following code for computing tetrahedral numbers.

```
while (k > 0) {n = k;TRI;k = k - 1}
```

Building from the sets of claims about `TRI` above and letting `TET` abbreviate the larger loop, it can be specified as

$$\begin{aligned} X_{\texttt{TET}} = X_{\texttt{TRI}} \\ \cup \{(\langle \texttt{TET}; R, \mathtt{s} \mapsto s, \mathtt{k} \mapsto k, \mathtt{n} \mapsto n, S\rangle, \\ \{\langle R, \mathtt{s} \mapsto s + \sum_{i=0}^{k} \sum_{j=0}^{i} j, \mathtt{k} \mapsto k', \mathtt{n} \mapsto n', S\rangle \mid k', n' \in \mathbb{Z}\}) \\ \mid s, k, n \in \mathbb{Z}, k \geq 0, R \in \text{Stmt}, S \in (\text{Var} \setminus \{\mathtt{s}, \mathtt{k}, \mathtt{n}\}) \hookrightarrow \mathbb{Z}\} \end{aligned}$$

The initial states of `TET` where $k = 0$ reach the goal in a fixed number of steps as for `TRI`. The ones with $k > 0$ reach in two steps code that starts with `TRI`. As `TRI;k = k - 1;TET`; R in an instance of the claim about `TRI`, transitivity can be used to shows s will reach a state where the remaining code is `k = k - 1;TET`; R

and s has been increased by $\sum_{i=0}^{k} i$, from which another step matches an original claim. Together this says

$$X_{\texttt{TET}} \setminus X_{\texttt{TRI}} \subseteq \mathit{step}[\mathit{step}[\mathit{trans}[X_{\texttt{TRI}} \cup \mathit{step}[X_{\texttt{TET}}]]]]$$

so $X_{\texttt{TET}} \subseteq \mathit{step}[\mathit{derived}[\mathit{trans}, X_{\texttt{TET}}]]$.

2.4 Derived Rule Examples

All derived rules are functions $\mathcal{P}(\mathit{cfg} \times \mathcal{P}(\mathit{cfg})) \to \mathcal{P}(\mathit{cfg} \times \mathcal{P}(\mathit{cfg}))$.

Transitivity:

$$\mathit{trans}[X] = \{(x, P) \mid (x, Q) \in X \wedge \forall y \in Q.(y, P) \in X\}$$

This satisfies $\mathit{trans}[\mathit{step}[X]] \subseteq \mathit{step}[\mathit{derived}[\mathit{trans}, X]]$ because $(x, Q) \in \mathit{step}[X]$ means either $x \in Q$ or $(z, Q) \in X$ for some $x\ R\ z$. In the first case by construction of trans, $x \in Q$ implies that (x, P) is in $\mathit{step}[X]$. In the second case we have $(z, Q) \in X \subseteq \mathit{derived}[\mathit{trans}, X]$ and $(w, P) \in \mathit{step}[X] \subseteq \mathit{derived}[\mathit{trans}, X]$ for each $w \in Q$, so $(z, P) \in \mathit{trans}[\mathit{derived}[\mathit{trans}, X]] \subseteq \mathit{derived}[\mathit{trans}, X]$, and thus $(x, P) \in \mathit{step}[\mathit{derived}[\mathit{trans}, X]]$.

Weakening:

$$\mathit{weaken}[X] = \{(x, P) \mid (x, Q) \in X \wedge Q \subseteq P\}$$

For $(x, P) \in \mathit{weaken}[\mathit{step}[X]]$, take a $(x, Q) \in \mathit{step}[X]$ with $Q \subseteq P$. If $x \in Q$ then also $x \in P$ so (x, P) is in $\mathit{step}[\mathit{derived}[\mathit{trans}, X]]$, and in fact in any image of step. Otherwise, there is some y with $x\ R\ y$ and $(y, Q) \in X$. By assumption on Q, $(y, P) \in \mathit{weaken}[X]$, so $(x, P) \in \mathit{step}[\mathit{derived}[\mathit{trans}, X]]$.

Allowing the proof to use any rules already known to be true (perhaps from other reachability proofs):

$$\mathit{assumed}[X] = \mathit{reaches}$$

Here the fixpoint equation $\mathit{reaches} = \mathit{step}[\mathit{reaches}]$ implies

$$\mathit{assumed}[\mathit{step}[X]] = \mathit{reaches} = \mathit{step}[\mathit{reaches}] \subseteq \mathit{step}[\mathit{derived}[\mathit{assumed}, X]]$$

2.5 Higher Order Specifications

For many language features such as objects, higher order function, or simple machine-level jump instructions, the preconditions or postconditions of a specification include reachability claims about arguments or results. This is sometimes known as the "Embedded Code Pointer" problem, for which the XCAP logic [12] was introduced, and adopted in Bedrock.

A higher order specification is naturally written as a predicate mentioning *reaches*. To fit in the coinductive proof system, such a specification is abstracted over *reaches*, taking a relation argument to be used instead. Instantiating the specification with *reaches* gives the original form. In this style, a specification

is a function $Spec : \mathcal{P}(cfg \times \mathcal{P}(cfg)) \to \mathcal{P}(cfg \times \mathcal{P}(cfg))$ rather than a simple set $X : \mathcal{P}(cfg \times \mathcal{P}(cfg))$.

Now that the about a specification is a function from a set of claims to a set of claims we can, if it is monotone, analyze it as another set of derived rules. By the standard validity condition, showing that

$$Spec[step[X]] \subseteq step\ derived[F + Spec, X]$$

for any valid rules F implies that all claims in $Spec[reaches]$ are true. In case the specification doesn't actually mention the argument relation, this condition reduces to showing that

$$Spec[\emptyset] \subseteq step\ derived[F, Spec[\emptyset]]$$

A simple set of claims could be considered a constant function always returning that set of claims, so this notion of a higher-order specification is a pure generalization of the first-order notion.

However, unless the program just takes some first-order functions as arguments the specification is probably not monotone, and uses the parameter abstracting *reaches* in both positive and negative conditions. To allow arbitrarily higher order specifications, instead of a single parameter abstracting *reaches*, a specification takes two, one for negative occurrences and one for positive occurrences. Now a specification is a function $Spec : \mathcal{P}(cfg \times \mathcal{P}(cfg)) \times \mathcal{P}(cfg \times \mathcal{P}(cfg)) \to \mathcal{P}(cfg \times \mathcal{P}(cfg))$ which should be anti-monotone in the negative (first) argument, and monotone in the positive (second) argument.

A suitable notion of proving such a specification while using derived rules F is showing

$$\begin{aligned} \forall NP.\ & P \subseteq derived[F, Spec[N, P]] \to \\ & derived[F, Spec[N, P]] \subseteq N \to \\ & Spec[N, P] \subseteq step[derived[F, Spec[N, P]]] \end{aligned}$$

While actually given such a proof, it may be that a claim is included in $Spec[N, P]$ only if certain claims being in N implies other claims are in P. The hypothesis on P allows the claims in P to be used while showing an element of $Spec[N, P]$ is in the conclusion, the hypothesis on N allows the needed claims in N to be supplied by constructing a proof in $derived[F, Spec[N, P]]$.

It can be shown that for any specification which is suitably monotone, provable in the above sense, and meets an additional weakening condition, there are choices of N and P such that $Spec[N, P] \subseteq reaches$. This is not as ideal as showing $Spec[reaches, reaches] \subseteq reaches$, but will still be useful. A main function or first-order entry point will have a first order specification, whose claims are included in $Spec[N, P]$ for any choice of N and P, even if those portions of the inclusion

$$Spec[N, P] \subseteq step[derived[F, Spec[N, P]]]$$

used higher-order specifications of other functions.

3 Research Goals

The central goal of this work is to introduce coinductive program verification, and validate it by producing a language-independent certifying program verifier.

To support this goal, we introduce an approach to proving partial correctness properties based on elementary reasoning about the transition relation, which nevertheless is sufficiently powerful to take the place of an axiomatic semantics.

We plan to build a Coq backend for the K framework to gain access to formal definitions of many small languages, and extensively-tested definitions of large languages. Handwritten operational semantics of toy languages will be used initially to develop the approach, and operational semantics in Coq from other sources such as OTT[16] will validate the generality of the approach.

4 Proposed Work

The proposed work falls into three categories. The first is refining and formalizing the central coinductive proof and specification techniques. This is almost entirely mathematical.

The second is obtaining definitions of languages, programs, and specifications in a compatible style, so the coinductive proof techniques can be applied to leave concrete proof goals. This splits into defining the transition system of a language, and defining the predicates on individual states used in specifications.

The final portion is proof automation, to actually deliver proofs, and a functional system.

4.1 Coinductive Proof Techniques

We have described the basic foundations of the coinductive verification approach above. The basic techniques seem to already be a sufficient foundation for interesting program verification tasks, as shown by MatchC. For these techniques, the main work to be done is describing them nicely for publication, and translate them into Coq. Both are in progress, with the paper draft aiming for LICS, and the formalization extensive and already applied in proofs of some simple simple examples, such as verifying a program for reversing a linked list.

There is some room for further refinement in the basic theory. One area is analyzing conditions for validity of rules. Fixing the basic notion of giving a proof of X using rules F as showing $X \subseteq \mathit{step}[\mathit{derived}[F, X]]$, two notions of soundness are that this implies in any way that $X \subseteq \mathit{reaches}$, or that it implies in particular that $\mathit{derived}[F, X]$ is itself step-stable.

Ideally a validity condition would allow freely combining sets of rules - and then mixing in a set X of claims. The condition for rule validity given in section 2 is apparently stronger (by dropping a hypothesis on X) than a condition that exactly ensures $\mathit{derived}[F, X]$ is step-stable for X with a proof $X \subseteq \mathit{step}[\mathit{derived}[F, X]]$, but all natural rules such as transitivity, weakening, adding semantic facts, etc. meet the stronger condition. The only counterexamples I've found depend on making claims about states with no predecessors, so there may be some hope of

constructing from any rule that meets the necessary condition a related rule that meets the stronger, composable condition.

There are some avenues for generalization. Higher-order specifications in particular are just being developed. One sound definition of proof was given above, an example program shown to satisfy a second-order specification, and an appropriately monotonic definition given for an arbitrarily higher-order specifications (implementing a function of a higher-order type), but it remains to be seen how easy it is to assemble definitions of such properties, and to prove that programs satisfy higher-order specifications.

Another area to explore concerns translations between languages. Given a translation between languages, what is required to give a useful translation of reachability properties? Comparing more coarse and more detailed transition systems for the same language is one application, or validating a sufficiently simple compiler from a higher to a lower level language.

4.2 Formalization in Proof Assistants

The key reasoning principles will be formalized in proof assistants, beginning with a proof that reachability is a greatest fixpoint, proving the lemmas regarding coinduction with derived rules, the sufficiently of various derived rule conditions, and proving that various derived rules are sound. This has been done in Coq for all results. To demonstrate that the approach does not depend on details of Coq's logic, the results should also be formalized in some other system, and used to prove at least a few toy examples

4.3 Proof Automation

The linchpin of this proposal is giving a semantic description of partial correctness as a coinductive property within a transition system, and showing how coinduction justifies various reasoning principles.

Extending this to a certifying program verification framework requires two further elements. The first is translations of language definitions into definitions of transition systems in the proof assistant (or logic) in which we will produce proof certificates. The second is a collection of useful proof rules and tactics for applying them to attempt to automatically verify programs.

For a proof strategy we will build on Reachability Logic and the MatchC prover. The MatchC verifier uses reachability logic only in a way which can be seen as an instance of this approach, and it's tactics seem to be effective in practice. Directly using a semantic foundation grounded on coinduction actually makes it simpler to justify the principles such as "set circularity" that MatchC actually employs, which can only be shown to be valid derived rules in reachability logic with translations exponential in the size of the proof tree. This is unobjectionable when arguing for the soundness of a tool, but undesirable in proof certificates.

Uncertified Verifier An uncertified verifier based on Reachability Logic is being developed as part of the K framework, as a successor to MatchC. The design is based on Reachability Logic, but the proof search actually used produces proofs of a form that are valid coinductive evidence for the specifications. Translating a trace of steps taken in a successful proof into proof steps in Coq will give a proof certificate, up to conditions the matching logic prover delegates to an SMT solver.

This provides an alternative to fully automating proof search in Coq, in case some essential forms of reasoning are difficult or too inefficient to implement as tactics.

4.4 Formal semantics of languages

To apply coinductive reasoning based on an operational semantics, we need to have language semantics formalized in the same system as the coinduction principles.

Relational Having a definition of a translation relation is essential to applying coinductive proof techniques, and is the most natural translation of a K-style rewriting-based operational semantics.

Given a formalization of a translation relation, the soundness of the coinduction lemmas demonstrate that any rules proved are true about that transition relation, but there's still the question of whether the formalized definition corresponds to the original language.

Executable One reason to work with operational semantics is that they are executable, and thus we can test whether the language we have defined is the language we meant to define by running examples.

We have that confidence for the original specification. Translating the original definition into a transition function rather than a relation lets us similarly test that the formal translation is faithful to the behavior of the original definition, by running examples as function evaluation within Coq, or by program extraction. Proving equivalence with a relational definition will show the relational definition is equally sound.

This is somewhat more involved translation, needing to provide an implementation rather than just a specification of pattern matching. It might also have some use in proving, if simplification by evaluation can usefully replace some proof search.

Deep A second approach to improve confidence of translations is to define and validate a formal semantics of the language defining framework once and for all, and then translate individual definitions simply by transliterating the AST of the definition.

This representation would not be suitable for direct use, because it would almost certainly be based on generic representations of configurations and rules of

any language, adding an extra layer of indirection even beyond a relational semantics. For these reasons, a deep embedding of the source notation would mainly be used by proving it equivalent to the more convenient relational translation.

4.5 Evaluation

That coinductive proof techniques can be applied to reduce showing reachability properties in any transition relation to showing certain stability properties is simply a matter of mathematics, already formalized and verified in Coq.

The questions are how difficult it is to give specifications of interest as reachability properties, and how difficult it is to actually find coinductive proofs of a particular specification.

To evaluate this I will prove specifications covered by other systems, considering proof and specification effort.

Examples should cover both easy properties that should be proved automatically, and difficult properties that should be possible.

For comparative evaluation, I will start with examples from MatchC[14], as MatchC is closely related to my approach.

For more intricate and automated examples I consider those proved with Bedrock in [4]. As Bedrock handles all proofs within Coq I should be able to match the given level of automation (hopefully I can directly incorporate some of the lemmas and tactics for heap reasoning).

For exceptionally difficult problems I look to the work of Zhong Shao's FLINT group[1] at Yale, who have approaches to verification problems such as self-modifying code[3], garbage collection[10], even preemptive threading implementations[7].

For presumably simpler properties I will look at examples from verification-condition generation-based tools such as Why3[2] and Boogie[1], which tend to rely on SMT solvers and aim to be usable without much user assistance in the proving process.

5 Current Results

All the mathematical results concerning reachability, coinduction, the use and validity of collections of derived rules, and higher-order specifications have been formalized and verified in Coq as they are developed.

Some simple language definitions (IMP extended with a heap, and a stack-based language with code pointers) have been defined by hand support this work and earlier work on formalizing and proving the soundness of Reachability Logic in Coq. These languages have been used as a basis for writing, specifying, and verifying programs using the coinductive approach.

Preliminary work on a Coq backend can translate a small subset of K into relational specifications.

[1] http://flint.cs.yale.edu

6 Schedule

From now through mid-January I will complete my paper introducing the basic approach. This will consist of refining text describing the approach, and working with the existing hand-translated language definitions to verify the sort of properties already covered by MatchC.

After this I will resume work on a Coq backend for the K system, starting with the current labeled translation. As features of K are added, increasingly complicated languages will be available. I will verify properties of code in a variety of languages, to see what sort of predicates on states are necessary, and work on proof automation for these predicates. After I have some idea what predicates are needed, I'll look into borrowing Bedrock's definition of heaps and automation for predicates on heaps. This phase will provide transitions systems and experience stating and proving reachability properties for a variety of types of languages, including basic imperative languages, object oriented and functional languages, maybe logic programming, and perhaps even some of the more unusual K definitions like type systems. I estimate this will be done around May.

After getting a basic handle on a range of languages, I'll start working seriously on evaluation against other systems. The first step here is selecting examples from other systems, and defining in K the languages they can verify, unless suitable operational semantics are already in Coq, and working on verifying examples. Higher-order specifications will become essential at this point, to handle code pointers, and some time will be spent working out details. The other parts of evaluation are checking that the coinductive proof principles can be defined and applied in other proof assistants (on toy languages), and proofs can be made about Coq operational semantics from other systems such as OTT. I expect to have completed proofs of the examples or determined the limits of the technique by September, and have a much better idea how much language-specific proof automation is necessary.

In the fall I work on elements of completing the project that haven't been addressed above. By this point the uncertified matching logic prover for K will probably be available, translating proof traces into Coq proofs will be fit into the schedule as it develops. For the translation from K to Coq one remaining point is to apply the translation to the full semantics of large languages, adding support for any features of K not covered in phase 1, and testing how proof automation scales with more and more complicated rules. Another aspect of the translation is completing executable and or deeply-embedded translations to follow the principled approaches to validating the translation. This is the time to try to automatically generate any repetitive sort of language-specific proof tactics identified in phase two.

The spring will be reserved for writing the thesis.

7 Related Work

The most closely related work is Reachability Logic[14], this proposal originates from working to understand and simplify the soundness arguments for Reach-

ability Logic. I use exactly the same idea of a reachability property, and rely on the evidence from the development of reachability logic that reachability properties are sufficiently expressive to replace Hoare-style partial correctness claims, and may smoothly accommodate extensions like separation algebra in the single-state pattern language. My approach differs most notably in doing without a fixed syntactic proof system, instead building up soundness and a machinery for derived rules in semantic terms. This allows the approach to be directly implemented in the meta-logic of a theorem prover. Validating new proof rules and linking partial proofs together is more straightforward when these notions are simply inclusions between sets rather than proof trees.

7.1 Coinduction in verification

A few systems allow working with coinductive specifications. For example, the Dafny[8] language and verification tool allows allows coinductively defined predicates and data types, and writing corecursive predicates[9]. Here the specifications are still predicates on the return value or resulting state after a method call. However, the coinduction here is in the specifications, rather than the techniques for proving that the programs satisfy specifications.

Nakata and Uustalu[11] give Hoare-style proof system for a coinductive trace-based semantics of a simple imperative language. The big-step semantics evaluates code to a possibly infinite sequence of states, and the postconditions are predicates over the sequence of states, with various coinductively-defined predicates. Soundness and completeness is proven in the usual way, soundness by induction over proof trees, and completeness by showing a strongest postcondition predicate transformer can be defined, and that the strongest postcondition is derivable.

There has been some work using fixpoints to analyze the soundness of axiomatic semantics, such as [5]. Quoting the abstract,

> We argue that relative soundness and completeness theorems for Floyd-Hoare Axiom Systems ([6], [5], [18]) are really fixed point theorems. We give a characterization of program invariants as fixed points of functionals which may be obtained in a natural manner from the text of a program. We show that within the framework of this fixed point theory, relative soundness and completeness results have a particularly simple interpretation. Completeness of a Floyd-Hoare Axiom system is equivalent to the existence of a fixed point for an appropriate functional, and soundness follows from the maximality of this fixed point. The functionals associated with regular procedure declarations are similar to predicate transformers of Dijkstra; for non-regular recursions it is necessary to use a generalization of the predicate transformer concept which we call a relational transformer.

This was developed for simple imperative programs, but later extended into cover various properties of parallel programs[6]. It's crucially different in that it considers predicate transformers constructed as fixpoints of functionals derived

from the source code, to analyze the soundness and completeness of a preexisting proof system, while I analyze soundness of a specifications in terms of fixpoints of specification transformers.

8 Conclusion

I propose a new approach to verifying reachability properties of programs by coinductive reasoning directly on the transition relation of an operational semantics. It seems to be amenable to formalization and automation in proof assistants, giving a certifying program verification approach tool which requires minimal creativity or proof effort to retarget at different languages. I've outlined a plan to implement and evaluate this approach.

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.
2. 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.
3. Hongxu Cai, Zhong Shao, and Alexander Vaynberg. Certified self-modifying code. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 66–77, New York, NY, USA, 2007. ACM.
4. 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.
5. Edmund Melson Clarke. Program invariants as fixed points. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, SFCS '77, pages 18–29, Washington, DC, USA, 1977. IEEE Computer Society.
6. Edmund Melson Clarke, Jr. Synthesis of resource invariants for concurrent programs. In *Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '79, pages 211–221, New York, NY, USA, 1979. ACM.
7. Xinyu Feng, Zhong Shao, Yuan Dong, and Yu Guo. Certifying low-level programs with hardware interrupts and preemptive threads. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 170–182, New York, NY, USA, 2008. ACM.
8. K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, LPAR'10, pages 348–370, Berlin, Heidelberg, 2010. Springer-Verlag.
9. K. Rustan M. Leino and Micha Moskal. Co-induction simply: Automatic coinductive proofs in a program verifier. Technical Report MSR-TR-2013-49, Microsoft Research, Redmond, Washington, July 2013.

10. Andrew McCreight, Zhong Shao, Chunxiao Lin, and Long Li. A general framework for certifying garbage collectors and their mutators. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 468–479, New York, NY, USA, 2007. ACM.
11. Keiko Nakata and Tarmo Uustalu. A hoare logic for the coinductive trace-based big-step semantics of while. In *Proceedings of the 19th European Conference on Programming Languages and Systems*, ESOP'10, pages 488–506, Berlin, Heidelberg, 2010. Springer-Verlag.
12. Zhaozhong Ni and Zhong Shao. Certified assembly programming with embedded code pointers. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '06, pages 320–333, New York, NY, USA, 2006. ACM.
13. 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.
14. Grigore Rosu and Andrei Stefanescu. 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.
15. Grigore Rosu and Andrei Stefanescu. From hoare logic to matching logic reachability. In *Proceedings of the 18th International Symposium on Formal Methods (FM'12)*, volume 7436 of *Lecture Notes in Computer Science*, pages 387–402. Springer, 2012.
16. Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar, and Rok Strniša. Ott: Effective tool support for the working semanticist. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*, ICFP '07, pages 1–12, New York, NY, USA, 2007. ACM.