A Coinduction Proof Rule for Hoare Doubles

Christian J. Bell
Massachusetts Institute of Technology
cj@csail.mit.edu

Adam Chlipala
Massachusetts Institute of Technology
adamc@csail.mit.edu

Abstract

We show how to implement a Hoare logic for corecursive programs encoded by a mixed embedding in Coq in the continuation-passing style. One of the challenges that we encountered while working with corecursion is that Coq's support for coinductive proofs is (notoriously) difficult to work with directly. Hence our primary focus is on our development of a Hoare double proof rule for coinduction that enables compositional and incremental proofs. Our results have been formalized and proved in Coq.[1]

1. Introduction

In our proof developments, we strive to take maximal advantage of the concision of typed functional programming and higher-order logic provided by Coq. One economical choice is to work with programs in *continuation-passing style* (CPS), eliminating the concept of postconditions from our verification vocabulary and allowing us to reason gracefully about unstructured control flow. *Hoare doubles*, consisting of just a precondition and program statement, provide a more natural way to reason about such programs than the canonical *Hoare triple*, which additionally specifies a postcondition.

In this demonstration, we consider imperative programs that are defined coinductively in CPS:

```
CoInductive stmt : Set :=
  | ret: ℤ → stmt
  | abort: stmt
  | act: operation → stmt
  | bind: stmt → (ℤ → stmt) → stmt.
```

Statement `ret` v represents a program that has terminated with return value v, `abort` allows a program to issue an error (unsafe), `act` α performs imperative operation α (e.g. writing or reading from the heap) and steps to state `ret` v for return value v, and `bind` s k either steps s, or else if $s =$ `ret` v, runs the continuation by stepping to k v. We write $x \leftarrow s_1; s_2$ to denote `bind` s_1 (`fun` $x \Rightarrow s_2$), $s_1; s_2$ to denote `bind` s_1 (`fun` _ $\Rightarrow s_2$), and $x \leftarrow \alpha; s$ to denote `bind` (`act` α) (`fun` `x` $\Rightarrow s$). Variables, conditional branching, and looping are not directly encoded; we instead shallowly embed them as Gallina functions that return `stmt`, hence programs are ultimately written in a "mixed" embedding style.

1.1 An example

Program `isTitleCase` checks whether a 0-terminated string is in title case, returning 1 if true and 0 otherwise:

```
CoFixpoint parseWS a : stmt:=
  x ← read a;
  if x = 0 then ret 1 (* Title Case *)
  else if x = " " then parseWS (1+a)
  else if ¬ is_letter x ∨ is_upper x then parseLetters (1+a)
  else ret 0 (* first letter of a word is lowercase *)
with parseLetters a :=
  x ← read a;
  if x = 0 then ret 1 (* Title Case *)
  else if x = " " then parseWS (1+a)
  else if ¬ is_letter x ∨ is_lower x then parseLetters (1+a)
  else ret 0 (* cAps in woRd *).

Definition isTitleCase := parseWS.
```

The specification should look something like:

$$\frac{\texttt{is\_title\_case}\ s \Longrightarrow \vdash \{a \mapsto s * F\}\ k\ 1 \qquad \neg\texttt{is\_title\_case}\ s \Longrightarrow \vdash \{a \mapsto s * F\}\ k\ 0}{\vdash \{a \mapsto s * F\}\ c \leftarrow \texttt{isTitleCase}\ a;\ k\ c}$$

Where `is_title_case` is a pure Coq function that determines if the abstract string s is in title case. The specification states that if its continuation is safe, then `isTitleCase` is safe. Our definition of safety is a coinductive property on programs and machine states, which holds when either the program is terminated (`ret`) or else is able to take a step and, for any step taken, it continues to be safe.

There are two ways to prove safety: by induction on the abstract string or by coinduction on the execution of the program. But proof by induction is generally not applicable because many programs do not terminate or else their termination may be inconvenient to check or even nondeterministic. Coinduction applies in all cases, so we seek a general technique to apply it in a way that is (ideally) no harder to use than induction.

If we directly invoke the `cofix` tactic, however, Coq's guardedness checker will not allow us to call any helper lemmas that make use of the coinductive hypothesis, such as a separate proof for `parseLetters`. This breaks modularity in two ways. First, we cannot factor the proof of `isTitleCase` into two smaller lemmas because they cannot use each other's coinductive hypotheses. Second, we must perform coinduction on the execution of the entire program all at once. In other words, we must write a coinductive hypothesis by hand that accounts for all future branching behaviors. A more natural proof would be *incremental*, such that we perform coinduction on `parseWS`, and then perform coinduction again when we eventually step to `parseLetters`.

1.2 Hoare doubles with program invariants

Invariants are one of the central ideas in program proof, describing properties of states of some transition system, where the property holds initially and is preserved by all transitions. We define two extended versions of our Hoare double to facilitate invariant-based reasoning that will allow us to write modular and incremental coinductive proofs.

The first is $\mathcal{I} \Vdash \{\mathcal{A}\}\ s$, saying that either precondition $\mathcal{A}$ guarantees the safety of running statement s, where we also consider it "safe" to run at least one step and reach a state in the set $\mathcal{I}$, or else $\mathcal{A}$ and s satisfy $\mathcal{I}$. These sets $\mathcal{I}$ consist of pairs of predicates and statements. This judgment is defined in terms of $\mathcal{I} \Vdash\!\!\vdash \{\mathcal{A}\}\ s$, which covers the case where at least one step *must* be taken.

[1] Coq source is available at `https://github.com/siegebell/hdcoind`

$$\frac{\mathcal{I} \Vdash \{\mathcal{A}\}\ s}{\mathcal{I} \Vdash\!\models \{\mathcal{A}\}\ s}\ \text{H-1} \qquad \frac{(\mathcal{A}, s) \in \mathcal{I}}{\mathcal{I} \Vdash\!\models \{\mathcal{A}\}\ s}\ \text{H-2} \qquad \frac{\vdash \{\mathcal{A}\}\ s}{\mathcal{I} \Vdash \{\mathcal{A}\}\ s}\ \text{H-3}$$

$$\frac{\emptyset \Vdash \{\mathcal{A}\}\ s}{\vdash \{\mathcal{A}\}\ s}\ \text{H-4} \qquad \frac{\forall x.\, \mathcal{I} \Vdash \{\mathcal{A}(x)\}\ s}{\mathcal{I} \Vdash \{\exists x.\, \mathcal{A}(x)\}\ s}\ \text{H-5}$$

$$\frac{\mathcal{I}_1 \Vdash \{\mathcal{A}_1\}\ s \qquad \mathcal{I}_1 \subseteq \mathcal{I}_2 \qquad \mathcal{A}_2 \vdash \mathcal{A}_1}{\mathcal{I}_2 \Vdash \{\mathcal{A}_2\}\ s}\ \text{H-6}$$

$$\frac{\mathcal{I}_1 \Vdash\!\models \{\mathcal{A}_1\}\ s \qquad \mathcal{I}_1 \subseteq \mathcal{I}_2 \qquad \mathcal{A}_2 \vdash \mathcal{A}_1}{\mathcal{I}_2 \Vdash\!\models \{\mathcal{A}_2\}\ s}\ \text{H-7}$$

$$\frac{\begin{array}{c}(\mathcal{A}, s) \in \mathcal{I}' \\ \forall \mathcal{A}', s'.\, (\mathcal{A}', s') \in \mathcal{I}' \Rightarrow (\mathcal{I}' \cup \mathcal{I}) \Vdash \{\mathcal{A}'\}\ s'\end{array}}{\mathcal{I} \Vdash \{\mathcal{A}\}\ s}\ \text{H-Inv}$$

Figure 1: Selection of proof rules for Hoare doubles.

Figure 1 shows several lemmas that we have proved about Hoare doubles with invariants. The crucial final lemma in the figure justifies the usefulness of the new judgments. It corresponds to *strengthening the coinduction hypothesis*. When we are trying to prove some $\mathcal{I} \Vdash \{\mathcal{A}\}\ s$ (or $\mathcal{I} \Vdash\!\models \{\mathcal{A}\}\ s$), we may decide that the invariant $\mathcal{I}$ is not general enough. We pick some $\mathcal{I}'$ to extend it with, where $\mathcal{I}'$ contains the current proof obligation $(\mathcal{A}, s)$ but may also contain others. We are then given a proof obligation that is *more work*, in the sense that we need to prove a Hoare double for every state compatible with $\mathcal{I}'$; but also *less work*, in that we may now assume $\mathcal{I}' \cup \mathcal{I}$ as the more general invariant. The choice of $\mathcal{I}'$ is analogous to picking a good loop invariant in conventional Hoare logic.

Judgments $\Vdash\!\models$ and $\Vdash$ work together in a coinductive proof of safety. The first, $\mathcal{I} \Vdash\!\models \{\mathcal{A}\}\ s$, corresponds to a proof state where Coq's cofixpoint guardedness checker has been satisfied: we can use the coinductive hypothesis by appealing to $(\mathcal{A}, s) \in \mathcal{I}$ directly. The second corresponds to a proof state where the guardedness checker has not yet been satisfied, so we must take at least one step of execution before appealing to $\mathcal{I}$. Each time we strengthen the induction hypothesis, we are left with a $\Vdash$ judgment, so it becomes necessary for the program to take at least one step before we can appeal to its induction hypothesis.

2. Application

To make the experience of performing coinduction more natural, we have defined a tactic, `hd_coind`, to 1) apply rule H-Inv, 2) automatically infer $\mathcal{I}'$ based on the hypotheses that the user has generalized (i.e. reverted to the goal), and 3) pose new hypotheses to help the user apply their generalized invariant. In our experience, this has made the proofs of programs like `isTitleCase` feel much like performing plain induction or proving a while loop (as it should be!). Using rule H-Inv, we can now break the proof into separate lemmas, stated in terms of $\Vdash$ and $\Vdash\!\models$, and parameterized over $\mathcal{I}$.

```
Definition tc_ret s := if is_title_case s then 1 else 0.

Lemma parseLetters_ok: ∀ I₀ I a s F k,
  I₀ ⊩= {a ↦ s ∧ F} k (tc_ret ("A"::s)) →
  I₀ ⊆ I →
  (∀ a s F,  I₀ ⊩= {a ↦ s ∧ F} k (tc_ret s) →
    I ⊩={a ↦ s ∧ F} x ← parseWS a; k x) →
  I ⊩ {a ↦ s ∧ F} x ← parseLetters a; k x.
Proof. ... Qed.
```

To prove `parseLetters_ok`, we first generalize the premise:

```
HI0 : I₀ ⊆ I
Hsafe_sws : ∀ a s F,
            I₀ ⊩={a ↦ s ∧ F} k (tc_ret s) →
            I ⊩={a ↦ s ∧ F} x ← parseWS a; k x
______________________________________(1/1)
∀ a s F,
I₀ ⊩={a ↦ s ∧ F} k (tc_ret ("A"::s)) →
I ⊩{a ↦ s ∧ F} x ← parseLetters a; k x
```

and then run `hd_coind` to strengthen the invariants to:

$$\texttt{I'} = \texttt{I} \cup \bigcup_{a,s,F} \{\, (a \mapsto s \wedge F, \texttt{c} \leftarrow \texttt{parseLetters}\ a; \texttt{k c}) \mid \texttt{I}_0 \Vdash\!\models \{a \mapsto s \wedge F\}\ \texttt{k (tc\_ret ("A"::}s\texttt{))} \,\},$$

which results in proof state:

```
HI0 : I₀ ⊆ I
Hsafe_sws : ∀ a s F,
            I₀ ⊩={a ↦ s ∧ F} k (tc_ret s) →
            I ⊩={a ↦ s ∧ F} x ← parseWS a; k x
H :  ∀ a s F,
     I₀ ⊩={a ↦ s ∧ F} k (tc_ret ("A"::s)) →
     I' ⊩={a ↦ s ∧ F} x ← parseLetters x; k x
H0 :  I ⊆ I'
Hsafe_k : I₀ ⊩={a ↦ s ∧ F} k (tc_ret ("A"::s))
______________________________________(1/1)
I' ⊩  {a ↦ s ∧ F}x ← parseLetters a; k x
```

By automatically generating hypotheses like `H`, the user may use their usual automation tactics (e.g. `eauto`) to access their coinductive hypotheses instead of manually applying rules H-1 and H-2. We continuing to step through the program, and when we reach a corecursive call to `parseLetters`, it will have occurred after at least one step, so we may appeal to `I'` by applying `H`. Reaching `parseWS`, we assume safety by `Hsafe_sws` even though we have not yet proved `parseWS`. In the next proof, we will have to show that `parseWS`'s coinductive hypothesis lines up with premise `Hsafe_sws`.

```
Lemma parseWS_ok: ∀ I a s F k,
  I ⊩= {a ↦ s ∧ F} k (tc_ret s) →
  I ⊩ {a ↦ s ∧ F} x ← parseWS a; k x.
Proof. ... Qed.
```

To prove `parseWS_ok`, we first generalize its premise and apply `hd_coind`, resulting in proof state:

```
H :  ∀ a s F,
     I ⊩={a ↦ s ∧ F} k (tc_ret s) →
     I' ⊩={a ↦ s ∧ F} x ← parseWS x; k x
H0 :  I ⊆ I'
Hsafe_k : I ⊩={a ↦ s ∧ F} k (tc_ret s)
______________________________________(1/1)
I' ⊩{a ↦ s ∧ F} x ← parseWS a; k x
```

When we reach `parseLetters`, we apply `parseLetters_ok` with parameter $\texttt{I}_0$ set to `I` and parameter `I` set to `I'`, and we are done.

```
Lemma isTitleCase_ok: ∀ I a s F k,
  I ⊩= {a ↦ s ∧ F} k (tc_ret s) →
  I ⊩ {a ↦ s ∧ F} x ← isTitleCase a; k.
Proof. intros; apply parseWS_ok; assumption. Qed.
```

Note that the specifications above weakened the goal for the continuation k from $\Vdash$ to $\Vdash\!\models$, which implies that the functions will take at least one step. This is a guarantee to any caller that they may immediately use their coinductive hypotheses once the continuation is called.

3. Conclusion

We have given a very brief introduction to using Hoare doubles for programs in CPS encoded as a mixed embedding in Coq. Hoare

doubles work well for reasoning about CPS and arbitrary control flow, while CPS and the mixed embedding allow us to use reuse language features provided by Gallina while also having operations that mutate the machine state [1].

Our primary contributions are the invariant-based Hoare double judgements $\Vdash$ and $\models$ and the corresponding proof rule, H-INV. Combined with a tactic to automatically apply coinduction and generate user-friendly coinductive hypotheses, our proofs were of nearly the same size and complexity as our *inductive* proofs of `parseLetters_ok` and `parseWS_ok`, with the bonus of not having a base case for the empty string. We are also using these techniques in a separate project to verify concurrent programs.

To implement and prove the soundness of these judgments and rules, we use Tarski's greatest fixed point thereoms [3]. The approach for obtaining modular and incremental coinductive proofs is also documented and available as a general purpose Coq library named Paco [2], which takes care of some of the boiler-plate definitions and provides a generic tactic similar to `hd_coind`. Although Paco could have simplified some of our definitions and tactics, it does not provide a program-logic style interface for reasoning about imperative corecursive programs, which is where most of our proof effort went.

References

[1] A. 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. URL `http://doi.acm.org/10.1145/2500365.2500592`.

[2] C.-K. Hur, G. Neis, D. Dreyer, and V. Vafeiadis. The power of parameterization in coinductive proof. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '13, pages 193–206, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1832-7. URL `http://doi.acm.org/10.1145/2429069.2429093`.

[3] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific J. Math.*, 5(2):285–309, 1955. URL `http://projecteuclid.org/euclid.pjm/1103044538`.