Coinductive Program Verification

Thesis Proposal

Brandon Moore

University of Illinois

December 12, 2013
Outline

1. Introduction
   - Goals and Motivation
   - Operational Semantics
   - Specifications as Reachability

2. Approach
   - Reachability by Coinduction
   - Coinduction with Derived Rules
   - Higher-Order Specifications

3. Proposed Work
   - Coinduction Principles
   - Operational Semantics
   - Automating Verification
   - Validation
Goal
Program Verification for every language
Goal

Program Verification from (multi-step) Operational Semantics

- Always need executable semantics, to test formalization
- Can we avoid axiomatic semantics?
- Why operational?
  - Denotational is a whole different story
  - Don’t know how to handle big step
Goal

Program Verification from (multi-step) Operational Semantics

- Always need executable semantics, to test formalization
- Can we avoid axiomatic semantics?
- Why operational?
  - Denotational is a whole different story
  - Don’t know how to handle big step
Goal

Program Verification from (multi-step) Operational Semantics

- Always need executable semantics, to test formalization
- Can we avoid axiomatic semantics?
- Why operational?
  - Denotational is a whole different story
  - Don’t know how to handle big step
Program verifiers should be at least *certifying*

Certifying

A *certifying* verifier produces a proof certificate along with claims

Certified

A *certified* verifier has a proof that it returns only true claims.

- Semantics in certificate language
- Translate specifications to claims about semantics
- Certificates are proofs claims are true
- Coq for certificate language, proof checker
General Perspective

- Language independence by passing from syntax to extension
  - Semantics, specifications, proof principles, etc.
- Truth/Proof as inclusion
- Coinduction
Project Structure

$\varphi \rightarrow \varphi'$

Predicate Automation

Program

Coinduction Principles

Operational Semantics

Claims
Operational semantics

Definition

An *Operational Semantics* is a set $\textit{cfg}$ of configurations and a one-step transition relation $S \subseteq \textit{cfg} \times \textit{cfg}$.
A simple imperative language

- $cfg = Stmt \times (Var \rightarrow \mathbb{Z})$, written $\langle code, store \rangle$
- $S$ contains steps like
  
  $\langle \text{while} (n \neq 0) \{ s = s + n; n = n - 1 \},$
  
  $\{ s \mapsto 1, n \mapsto 10 \}\rangle$

  to

  $\langle s = s + n; n = n - 1; \text{while} (n \neq 0) \{ s = s + n; n = n - 1 \},$

  $\{ s \mapsto 1, n \mapsto 10 \}\rangle$
Transitive closure $S^*$ takes multiple steps, e.g.
\[
\langle \text{while } (n != 0) \{ s = s + n; \ n = n - 1 \} \}; \ x = s, \\
\{ s \mapsto 1, n \mapsto 10, x \mapsto 0 \} \rangle
\]
to
\[
\langle x = s, \\
\{ s \mapsto 56, n \mapsto 10, x \mapsto 1 \} \rangle
\]
Reachability

Definition

A configuration \( x \in \text{cfg} \) “reaches” a set \( P \subseteq \text{cfg} \) of configurations, written \( x \Rightarrow P \), when \( x S^* y \) for some \( y \in P \), or \( x \) diverges in \( S \)
Reachability from Hoare Triples

Start with a standard Hoare Triple

\[
\{ s = s_0, n = n_0 \} \\
\text{while } (n \neq 0) \{ s = s + n; \ n = n - 1 \} \\
\{ s = s_0 + \sum_{i=0}^{n_0} i \} 
\]
Drop special syntax for variables

Ordinary predicates on store of configuration

\{store(s) = s_0 \land store(n) = n_0\}

\textbf{while} (n \neq 0) \{s=s+n; \ n=n-1\}

\{store(s) = s_0 + \sum_{i=0}^{n_0} i\}
Ordinary predicates on configuration $\gamma$

\[
\begin{align*}
\{ \gamma.store(s) &= s_0 \land \gamma.store(n) = n_0 \\
&\land \gamma.code = \text{while} (n \neq 0) \{s = s + n; \ n = n - 1\}; R \} \\
\{ \gamma.store(s) &= s_0 + \sum_{i=0}^{n_0} i \\
&\land \gamma.code = R \}
\end{align*}
\]
Spec became a matching logic reachability property:

$$\varphi \Rightarrow_{RL} \varphi'$$

($$\varphi, \varphi'$$ predicates on $$cfg$$)
Expands to reachability claims

\[ \varphi \Rightarrow_{RL} \varphi' \]

iff

\[ \forall \gamma, \forall \text{fv}(\varphi), \varphi(\gamma) \rightarrow (\gamma \Rightarrow \{ \gamma' \mid \exists \text{fv}(\varphi') \setminus \text{fv}(\varphi), \varphi'(\gamma') \}) \]
Approach

- Specifications become sets of claims
- Proving sets of claims
- Coinduction
- Derived rules
**Definition**

Let $\text{claims} = \text{cfg} \times \mathcal{P}(\text{cfg})$, naming the set of all reachability claims.

**Definition**

Let $\text{reaches} \subseteq \text{claims}$ be the set of true reachability claims,

$$\text{reaches} = \{(x, P) \mid x \text{ reaches } P\}$$
Proof by Inclusion

\[ x \Rightarrow P \text{ for all } (x, P) \in R \]

\[ \text{iff} \]

\[ R \subseteq \text{reaches} \]
Inclusion by Coinduction

Definition
Given set $A$ and function $F : \mathcal{P}(A) \rightarrow \mathcal{P}(A)$, $X \subseteq A$ is $F$-stable if $X \subseteq F(X)$.

Coinduction
If $G$ is the greatest fixpoint of a monotone $F$, $X \subseteq F(X)$ implies $X \subseteq G$. 
Reachability as a Fixpoint

*reach* is the greatest fixpoint of $\text{step} : \mathcal{P}(\text{claims}) \rightarrow \mathcal{P}(\text{claims})$

$$\text{step}[R] = \text{done} \cup \text{next}[R]$$

where $\text{done} : \mathcal{P}(\text{claims})$, $\text{next} : \mathcal{P}(\text{claims}) \rightarrow \mathcal{P}(\text{claims})$

$$\text{done} = \{(x, P) \mid x \in P\}$$

$$\text{next}[R] = \{(x, P) \mid \exists y. x \mathcal{S} y \land (y, P) \in R\}$$

(Proof)
done in pictures

\[
\begin{align*}
(a, P) &\in \text{done} \\
(b, P) &\in \text{done} \\
(c, P) &\in \text{done} \\
(d, P) &\not\in \text{done}
\end{align*}
\]

\[
\begin{align*}
(a, Q) &\not\in \text{done} \\
(b, Q) &\not\in \text{done} \\
(c, Q) &\in \text{done} \\
(d, Q) &\in \text{done}
\end{align*}
\]
next in pictures

$R$

$P$
next in pictures

\[ \text{next}[R] \]
next in pictures

\[\text{next}[\text{next}[R]]\]
reachability as a fixpoint

reach is the greatest fixpoint of \( \text{step} : \mathcal{P}(\text{claims}) \rightarrow \mathcal{P}(\text{claims}) \)

\[
\text{step}[R] = \text{done} \cup \text{next}[R]
\]

where \( \text{done} : \mathcal{P}(\text{claims}) \), \( \text{next} : \mathcal{P}(\text{claims}) \rightarrow \mathcal{P}(\text{claims}) \)

\[
\text{done} = \{(x, P) \mid x \in P\}
\]

\[
\text{next}[R] = \{(x, P) \mid \exists y. x \text{ S } y \land (y, P) \in R\}
\]
Direct coinduction 1

Goal: \( d \Rightarrow P \)
Direct coinduction 1

Expand to stable set $R$
Direct coinduction 1

\[ \text{step}[R] \]
Direct coinduction 1

\[(e, P) \in done \subseteq step[R]\]
\[(e, P) \in done \subseteq \text{step}[R]\]
Direct coinduction 1

\[(e, P) \in R \rightarrow (d, P) \in \text{next}[R] \subseteq \text{step}[R]\]
Direct coinduction 1

\((e, P) \in R \rightarrow (d, P) \in \text{next}[R] \subseteq \text{step}[R]\)
Goal: $a \Rightarrow P$
Direct coinduction 2

Expand to stable set $R$
Direct coinduction 2

step[$R$]
Direct coinduction 2

\[(b, P) \in R \rightarrow (a, P) \in \text{next}[R] \subseteq \text{step}[R]\]
Direct coinduction 2

$$(b, P) \in R \rightarrow (a, P) \in \text{next}[R] \subseteq \text{step}[R]$$
$(c, P) \in R \rightarrow (b, P) \in next[R] \subseteq step[R]$
\[(c, P) \in R \rightarrow (b, P) \in \text{next}[R] \subseteq \text{step}[R]\]
Direct coinduction 2

\[(a, P) \in R \to (c, P) \in next[R] \subseteq step[R]\]
\[(a, P) \in R \rightarrow (c, P) \in \text{next}[R] \subseteq \text{step}[R]\]
Direction coinduction on a program

To prove
\[
\langle \text{while } (n \neq 0) \{ n=n-1 \}, \{ n \mapsto n_0 \} \rangle \Rightarrow \{ \langle \text{skip}, \{ n \mapsto 0 \} \rangle \}\]

must also claim
\[
\langle n=n-1; \text{ while } (n \neq 0) \{ n=n-1 \}, \{ n \mapsto n_0 \} \rangle \\
\Rightarrow \{ \langle \text{skip}, \{ n \mapsto 0 \} \rangle \}\]

and
\[
\langle \text{skip}, \{ n \mapsto 0 \} \rangle \Rightarrow \{ \langle \text{skip}, \{ n \mapsto 0 \} \rangle \}\]
Recovering Rules

- Direct coinduction tedious
- Replacements for proof rules
  - Transitivity, Weakening, Assertion, etc.
- Combine freely
Multi-step coinduction

Given \( R : \mathcal{P}(claims) \), close under \( step \) by fixpoint construction

\[ \mu C \cdot R \cup step[C] \]

Definition

\( R : \mathcal{P}(claims) \) is step-stable using multiple steps if

\[ R \subseteq step[\mu C \cdot R \cup step[C]] \]

Multi-step coinduction

If \( R : \mathcal{P}(claims) \) has \( R \subseteq step[\mu C \cdot R \cup step[C]] \) then

\[ (\mu C \cdot R \cup step[C]) \subseteq step[\mu C \cdot R \cup step[C]] \]

and thus

\[ R \subseteq (\mu C \cdot R \cup step[C]) \subseteq reaches \]
Proving with multiple steps

Goal: $R = \{(a, P)\}$
Proving with multiple steps

Closing: $\text{step}[\emptyset]$
Proving with multiple steps

Closing: $R \cup \text{step}[\emptyset]$
Proving with multiple steps

Closing: $\text{step}[R \cup \text{step}[$\emptyset$]]$
Proving with multiple steps

Closing: $R \cup \text{step}[R \cup \text{step}[\emptyset]]$
Proving with multiple steps

Closing: \( \text{step}[R \cup \text{step}[R \cup \text{step}[\emptyset]]] \)
Proving with multiple steps

Closing: $R \cup step[R \cup step[R \cup step[\emptyset]]]$
Proving with multiple steps

Closed: $\mu C.R \cup step[C]$
Proving with multiple steps

Final step: \( \text{step}[\mu C . R \cup \text{step}[C]] \)

![Diagram with nodes and arrows representing graph structure](image)

- Node a
- Node b
- Node c
- Node d
- Node e
- Node P

Branches and connections illustrate relationships between the nodes.
Proving with multiple steps

Supported: $R \subseteq step[\mu C.R \cup step[C]]$
Reasoning forward

Goal: any $R$ which includes $(a, P)$
Reasoning forward

\[(a, P) \in \text{step}[\mu C.R \cup \text{step}[C]]\]
\((a, P) \in \text{done} \cup \text{next}[\mu C.R \cup \text{step}[C]]\)
Reasoning forward

\[(a, P) \in next[\mu C.R \cup step[C]]\]
Reasoning forward

\[(b, P) \in \mu C \cdot R \cup \text{step}[C] \land \forall a \exists b \rightarrow (a, P) \in \text{next}[\mu C \cdot R \cup \text{step}[C]]\]
Reasoning forward

\[(b, P) \in \mu C.R \cup step[C]\]
Reasoning forward

\[(b, P) \in R \cup \text{step}[\mu C. R \cup \text{step}[C]]\]
Reasoning forward

\[(b, P) \in \text{step}[\mu C.R \cup \text{step}[C]]\]
Reasoning forward

\((b, P) \in next[\mu C.R \cup step[C]]\)
Reasoning forward

\((c, P) \in \mu C.R \cup \text{step}[C]\)
Reasoning forward

\[(c, P) \in R \cup \text{step}[\mu C.R \cup \text{step}[C]]\]
Reasoning forward

\[(c, P) \in \text{next}[\mu C.R \cup \text{step}[C]]\]
Reasoning forward

\[(a, P) \in \mu C.R \cup \text{step}[C]\]
Reasoning forward

\[(a, P) \in R \cup \text{step}[\mu C.R \cup \text{step}[C]]\]
Reasoning forward

\[(a, P) \in R\]
Multi-step coinduction on a program

The set of claims

\[ \langle \text{while} \ (n \neq 0) \ {n=n-1}, \{n \mapsto n_0\} \rangle \Rightarrow \{\langle \text{skip}, \{n \mapsto 0\} \rangle\} \]

is \textit{step}-stable with multiple sets.
Transitivity

Want $x \Rightarrow Q$ if $x \Rightarrow P$ and $y \Rightarrow Q$ for all $y \in P$.
As a function $\text{trans} : \mathcal{P}(\text{claims}) \rightarrow \mathcal{P}(\text{claims})$

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

As before, given $R : \mathcal{P}(\text{claims})$, close under $\text{step}$ and $\text{trans}$ by fixpoint

$$\mu C.R \cup \text{step}[C] \cup \text{trans}[C]$$

Definition

$R : \mathcal{P}(\text{claims})$ is step-stable using multiple steps and transitivity if

$$R \subseteq \text{step}[\mu C.R \cup \text{step}[C] \cup \text{trans}[C]]$$

Is this sound?
In general, any monotone \( F : \mathcal{P}(\text{claims}) \rightarrow \mathcal{P}(\text{claims}) \) may be rules

**Definition**

Given monotone \( F : \mathcal{P}(\text{claims}) \rightarrow \mathcal{P}(\text{claims}) \) and claims \( R : \mathcal{P}(\text{claims}) \), define the closure

\[
\text{derived} : (\mathcal{P}(\text{claims}) \rightarrow \mathcal{P}(\text{claims})) \times \mathcal{P}(\text{claims}) \rightarrow \mathcal{P}(\text{claims})
\]

\[
\text{derived}[F, R] = \mu D. R \cup \text{step}[D] \cup F[D]
\]
Proving with rules

**Definition**

For monotone $F: \mathcal{P}(\text{claims}) \rightarrow \mathcal{P}(\text{claims})$, a set $R: \mathcal{P}(\text{claims})$ is **step-stable using** $F$ if

$$R \subseteq \text{step}[\text{derived}[F, R]]$$
Rule Validity

**Definition**

Monotone $F : \mathcal{P}(\text{claims}) \rightarrow \mathcal{P}(\text{claims})$ is *simply valid* if for any $X : \mathcal{P}(\text{claims})$,

$$X \subseteq \text{step}[\text{derived}[F, X]]$$

implies

$$\text{derived}[F, X] \subseteq \text{step}[\text{derived}[F, X]]$$
Compositional Validity

**Composition**

Given functions $F, G : \mathcal{P}(\text{claims}) \rightarrow \mathcal{P}(\text{claims})$, define *composition* by

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

Composition preserves monotonicity, but not simple validity: strengthen

**Definition**

$F : \mathcal{P}(\text{claims}) \rightarrow \mathcal{P}(\text{claims})$ is *compositionally valid* if it is monotone and

$$\forall X. F[\text{step}[X]] \subseteq \text{step}[\text{derived}[F, X]]$$
Validity and Compositionality of Compositional Validity

Validity Lemma

If $F : \mathcal{P}(\text{claims}) \to \mathcal{P}(\text{claims})$ is compositionally valid and $R \subseteq \text{claims}$ satisfies

$$R \subseteq \text{step}[\text{derived}[F, R]]$$

then

$$R \subseteq \text{derived}[F, R] \subseteq \text{step}[\text{derived}[F, R]] \subseteq \text{reaches}$$

Compositionality Lemma

If $F, G : \mathcal{P}(\text{claims}) \to \mathcal{P}(\text{claims})$ are both compositionally, then $(F + G)$ is also compositionally valid.
Validity Example: \textit{trans}

To show \textit{trans} is compositionally valid, fix $R \subseteq \text{claims}$, assume

$$(x, P) \in \text{step}[R] \quad \text{forally} \in P.(y, Q) \in \text{step}[R]$$

- If $x \in P$ then $(x, Q) \in \text{step}[R]$ by second hypothesis
- Else $x S z$ with $(z, P) \in R$. Then $(z, Q) \in \text{trans}[R \cup \text{step}[R]]$, $(x, Q) \in \text{step}[\text{trans}[R \cup \text{step}[R]]]$.  
- In either case,

  $$(x, Q) \in \text{step}[\text{derived}[\text{trans}, X]]$$
Interlude: Combining Proofs

To show

\[(\text{prog} \cup \text{library} \cup \text{internals}) \subseteq \text{step[derived[(trans + weaken + F), (\text{prog} \cup \text{library} \cup \text{internals})]]}\]

it suffices to show

\[\text{prog} \subseteq \text{step[derived[trans + F, prog \cup library]]}\]

and

\[\text{library} \cup \text{internals} \subseteq \text{step[derived[trans + weaken, library \cup internals]]}\]
“Second-Order” Program

What about higher order programs? Consider

\[ \text{add3}(\text{plus}, x, y, z) = \text{plus}(\text{plus}(x, y), z) \]

Specified by

\[
\forall p. \ (\forall a \ b \ E \ \sigma. \langle E[p(a, b)], \sigma \rangle \Rightarrow \{\langle E[a + b], \sigma \rangle\}) \rightarrow \\
\forall x \ y \ z \ E \ \sigma. \langle E[\text{add3}(p, x, y, z)], \sigma \rangle \Rightarrow \{\langle E[x + y + z], \sigma \rangle\}
\]
“Second-Order” Specification

First-order specifications became sets of claims. Now abstract over set of claims to consider true in premises. Specification becomes function $Spec : \mathcal{P}(\text{claims}) \rightarrow \mathcal{P}(\text{claims})$

$$Spec[H] = \{(\langle E[\text{add3}(p, x, y, z)], \sigma \rangle, \{\langle E[x + y + z], \sigma \rangle\}) \mid \forall a \ b \ E'[\sigma].(\langle E'[p(a, b)], \sigma' \rangle, \{\langle E'[a + b], \sigma' \rangle\})) \in H\}$$
“Second-Order” Proof

Functions from claims to claims used as derived rules, re-use validity condition.
If $F$ is compositionally valid,

$$\forall X. \ Spec[\text{step}[X]] \subseteq \text{step}[\text{derived}[F + Spec, X]]$$

implies $F + Spec$ is compositionally valid

Lemma

If $F + Spec$ and $F$ are compositionally valid,

$$Spec[\text{reaches}] \subseteq \text{reaches}$$
Higher-Order properties

In general, may take and return higher-order functions.

Simple types

\[ \tau := \mathbb{N} \mid \tau \rightarrow \tau \]

give a reachability property

\[
\text{type}(\mathbb{N}, v) = v \text{ is a number} \\
\text{type}(A \rightarrow B, v) = \forall a. \text{type}(A, a) \rightarrow \\
\langle E[v(a)] \rangle \Rightarrow \{ \langle E[r] \rangle \mid \text{type}(B, r) \}
\]

Can’t translate to spec of form \( \mathcal{P}(\text{claims}) \rightarrow \mathcal{P}(\text{claims}) \), monotoniticy ruined by positive and negative occurances of \( \Rightarrow \)
Mixed-Variance Predicates

Splitting argument suffices:

\[
\text{type} : \mathcal{P}(\text{claims}) \rightarrow \mathcal{P}(\text{claims}) \rightarrow \text{bool}
\]

\[
\text{type}(\mathbb{N}, v, \text{Neg}, \text{Pos}) = v \text{ is a scalar}
\]

\[
\text{type}(A \rightarrow B, v, \text{Neg}, \text{Pos}) = \forall a. \text{type}(A, a, \text{Pos}, \text{Neg}) \rightarrow \\
\left(\langle E[v(a)] \rangle, \{\langle E[r] \rangle \mid \text{type}(B, r, \text{Neg}, \text{Pos})\}\right) \in \text{Pos}
\]

Is monotone in Pos, anti-monotone in Neg
Higher-Order proof

A higher-order specification is function

\[ \text{Spec} : \mathcal{P}(\text{claims}) \rightarrow \mathcal{P}(\text{claims}) \rightarrow \mathcal{P}(\text{claims}) \]

which is anti-monotone in the first argument and monotone in the second argument.

Proved under rules \( F \) when

\[ \forall (\text{Neg} \; \text{Pos} : \mathcal{P}(\text{claims})) . \]
\[ \text{step}[\text{derived}[F, \text{Spec}[\text{Neg}, \text{Pos}]]] \subseteq \text{Neg} \rightarrow \text{Pos} \subseteq \text{step}[\text{derived}[F, \text{Spec}[\text{Neg}, \text{Pos}]]] \rightarrow \text{Spec}[\text{Neg}, \text{Pos}] \subseteq \text{step}[\text{Spec}[\text{Neg}, \text{Pos}]] \]

When \( \text{Spec} \) proved under \( F \) and rules \( F \) valid, there exist some \( N, P \subseteq \text{claims} \) such that

\[ \text{Spec}[N, P] \subseteq \text{reaches} \]
Project Structure

\[ \varphi \rightarrow \varphi' \]

Predicate
Automation

Program

Coinduction
Principles

Operational
Semantics

Claims

Brandon Moore  (University of Illinois)  Coinductive Program Verification  December 12, 2013  48 / 58
Coinduction Principles

- Formalize in Coq
- Publish Basic Theory
  - Submit to LICS 2014, mid January, with MatchC examples.
- Analyze derived rule conditions
- Develop higher-order case
Operational Semantics

- Handwritten semantics to start
- Generate from K definitions
  - Reuse predicates, semantic domains from handwritten semantics.
- Try existing Coq semantics, e.g. OTT
Operational Semantics - Claims

Also need claims

- Begin writing by hand
- Extend K translator for annotations
- Abbreviations as needed
  - Filling code component by embedding annotations in program
  - Hoare style variable lookup abbreviation?
  - Automatic threading of mixed-variance predicates?
Automating Verification

Two major sorts of automation

- **Order for applying proof rules**
  - MatchC successful with simple tactic: try in order to
    1. Finish
    2. Apply claim from specification
    3. Take a step in semantics
    4. Make a case split to enable above
       (generate tactic by analyzing semantics?)
  - May replay trace from K reachability prover.

- **Folding, unfolding, simplifying predicates of specification**
  - Pure single-state domain reasoning, should be able to borrow
Validating Power

Prove properties handled by
- MatchC (trees, Schorr-Waite)
- Bedrock (allocator, cooperative threads)
- FLINT (self-modifying code, interrupts)
Validating Automation

After power, look at proof effort

- Bedrock
- Boogie
- Why
Validating Adequacy

Proofs sound with respect to Coq semantics, may not match K interpreter

- Match execution of K tests
- Generate executable semantics, extract interpreter
- Formalize K, translate by deep embedding
End
Reachability is Greatest Fixpoint

fixed

If $x$ diverges, it has an immediate successor $y$ which also diverges. If $x S^* z$ for some $z \in P$, then either $x \in P$ already or $x$ has an immediate successor $y$ with $y S^* z$ as well, so $\text{reaches} = \text{step}[\text{reaches}]$.

greatest

Suppose $X \subseteq \text{step}[X]$, and $(x, P)$ is a claim in $X$. Consider whether $x$ diverges. If so, $(x, P) \in \text{reaches}$. If $x$ terminates, $S$ is well-founded under $x$, so by induction assume any claim $(y, P)$ in $X$ with $x S^+ y$ is also in $\text{reaches}$. As $X$ is $\text{step}$-stable, either $x \in P$ already, or $x$ has some successor $y$ with $(y, P) \in X$. In either case, $(x, P) \in \text{reaches}$.
ω not always enough

The least fixpoint is not necessarily reached by iterating a countable number of times from ∅. The set of states which terminate along all paths is the least fixpoint of

\[ F[S] = \{ x \mid \forall y. x S y \rightarrow y \in S \} \]

but

\[ \bigcup_{i=0}^{\infty} F^i[\emptyset] \]

misses programs such as the one that picks any natural number, and then runs for that many more steps. It’s found in the next step:

\[ F[\bigcup_{i=0}^{\infty} F^i[\emptyset]] \]

i.e, transfinite induction up to \( \omega + 1 \). Higher ordinals can be required.