Termination Proof Inference by Abstract Interpretation

Patrick Cousot, Radhia Cousot

cims.nyu.edu/~pcousot/  www.di.ens.fr/~rcousot/

Abstract

The existing approaches to termination proof are scattered and largely not comparable with each other.

We introduce a unifying design principle for termination based on an abstract interpretation of a complete infinitary trace semantics. We show that proof, verification and analysis methods for termination all rely on two induction principles: (1) a variant function or induction on data ensuring progress towards the end and (2) some form of induction on the program structure.

For (1), we show that the abstract interpretation-based design principle applies equally well to potential and definite termination. The trace-based termination collecting semantics is given a fixpoint definition. Its abstraction yields a fixpoint equally well to potential and definite termination. The trace-based termination checking.

For (2), we introduce a generalization of the syntactic notion of structural induction (as found in Hoare logic) into a "semantic structural induction" based on the new semantic concept of inductive trace cover covering execution traces by "segments", a new basis for formulating program properties. Its abstractions allow for generalized recursive proof, verification and static analysis methods by induction on both program structure, control, and data. Examples of particular instances include Floyd's handling of loop cut-points as well as nested loops. Burstall's intermittent assertion total correctness proof method, and Podelski-Rybalchenko transition invariants.

Three principles

Program verification methods (formal proof or static analysis methods) are abstract interpretations of a semantics of the programming language (\(^{(*)}\))


Refinement to principle II

Safety as well as termination verification methods are abstract interpretations of a maximal trace semantics of the programming language.

Comments on principle II

- This is well-known for instances of safety (like invariance) using prefix trace semantics
- This is true for full safety
- New for termination

New principle III

More expressive and powerful verification methods are derived by structuring the trace semantics (into a hierarchy of segments)

Comments on principle III

- Syntactic instances have been known for long (different variant functions for nested loops, Hoare logic for total correctness,...)
- Semantic instances have been ignored for long (Burstall's total correctness proof method using intermittent assertions) and very successful recently (Podelski-Rybalchenko)

---


Maximal trace semantics

Fixpoint maximal trace semantics

• Complete lattice
  \( \langle \wp(\Sigma^{\infty}), \subseteq, \Sigma^{\infty}, \cup, \cap \rangle \)

• Computational ordering
  
  \[
  (T_1 \subseteq T_2) \triangleq (T_1^+ \subseteq T_2^+) \land (T_1^{\infty} \supseteq T_2^{\infty})
  
  (T_1 \cup T_2) \triangleq (T_1^+ \cup T_2^+) \cup (T_1^{\infty} \cap T_2^{\infty})
  \]

• Fixpoint semantics
  
  \[
  \tau^{+\infty}[P] = \operatorname{lfp}_{\Sigma^{\infty}} \phi^{+\infty}_T[P]
  
  = \operatorname{lfp}_{\Sigma^+} \phi^+_T[P] \cup \operatorname{gfp}_{\Sigma^{\infty}} \phi^{\infty}_T[P]
  \]

(Trace) properties
Program properties

- A program property $P$ is the set of semantics which have this property:
  $$P \in \emptyset(\forall(\Sigma^+))$$

- Example:
  $$P = \left\{ \right\}$$

- Strongest property of program $P$:
  $$\{ \tau^+ [P] \}$$

The Termination Problem

Trace property abstraction

- Trace property abstraction:
  $$\alpha_\Theta(P) \triangleq \bigcup P <_{\forall(\Sigma^+), \subseteq} \frac{\gamma_\Theta}{\alpha_\Theta} <_{\forall(\Sigma^+), \subseteq}$$

- Example:
  $$P = \left\{ \right\}$$
  $$\alpha_\Theta(P) = \left\{ \right\}$$

- The strongest trace property of a trace semantics is this trace semantics $\alpha_\Theta((\tau^+ [P])) = \tau^+ [P]$

- Safety/liveness (termination) are trace properties, not general program properties

The termination proof problem

- Termination abstraction:
  $$\alpha^I(T) \triangleq T \cap \Sigma^+$$

- Termination proof:
  $$\alpha^I(\tau^+ [P]) = \tau^+ [P]$$

- Termination proofs are not very useful since programs do not always terminate
Example

- Arithmetic mean of integers x and y
  ```
  while (x <> y) {
    x := x - 1;
    y := y + 1
  }
  ```
- Does not always terminate e.g.
  `<x,y> = <1,0> → <0,1> → <-1,2> → <-2,3> → ...

The Termination Inference Problem

- Determine a necessary condition for program termination and prove it sufficient
- Example:
  - (1) Under which necessary conditions
    ```
    while (x <> y) {
      x := x - 1;
      y := y + 1
    }
    ```
    does terminate?
  - (2) Prove these conditions to be sufficient

Potential termination

- For non-deterministic programs, we may be interested in potential termination
**Definite termination abstraction**

- or in **definite termination**

  must terminate

- Potential and definite termination coincide for deterministic programs. Only **definite termination** in this presentation.

**Definite termination trace abstraction**

- Prefix Abstraction

  \[ \text{pf}(\sigma) \triangleq \{ \sigma' \in \Sigma^{+\infty} \mid \exists \sigma'' \in \Sigma^{+\infty} : \sigma = \sigma' \sigma'' \} \]

  \[ \text{pf}(T) \triangleq \bigcup \{ \text{pf}(\sigma) \mid \sigma \in T \} . \]

- Definite termination abstraction

  \[ \alpha^{Mt}(T) \triangleq \{ \sigma \in T^+ \mid \text{pf}(\sigma) \cap \text{pf}(T^{\infty}) = \emptyset \} \]

**Finite abstractions do not work**

- « Abstract and model-check » is **impossible**\(^{(1)}\) for termination and **unsound** for non-termination of **unbounded programs**

  - Unbounded executions:

    ![Unbounded executions diagram]

  - Finite homomorphic abstraction:

    ![Finite homomorphic abstraction diagram]

  - **Termination:** impossible (lasso)

  - **Non-termination:** (lasso): unsound

---

\(^{(1)}\) Excluding trivial solutions, see: Patrick Cousot, Partial Computation of Abstract Fixpoint Checking. SARA 2000, 1-25.
Definite termination domain

Reachability analysis

- A forward invariance analysis infers states potentially reachable from initial states (by over-approximating an abstract fixpoint \( \text{lfp} \, F \)) ✁

Combined reachability/accessibility analyses

- An iterated forward/backward invariance analysis infers reachable states potentially/definitely accessing final states (by over-approximating \( \text{lfp} \, F \cap \text{lfp} \, B \)) ✁

Accessibility analysis

- A backward invariance analysis infers states potentially/definitely accessing final states (by over-approximating an abstract fixpoint \( \text{lfp} \, B \)) ✁

\[ X^0 = \top \]
\[ \ldots \]
\[ X^{2n+1} = \text{lfp} \, \lambda Y. X^{2n} \cap F(Y) \]
\[ X^{2n+2} = \text{lfp} \, \lambda Y. X^{2n+1} \cap B(Y) \]
\[ \ldots \]


Example

• Arithmetic mean of two integers \( x \) and \( y \)

\[
\{x \geq y\}
\]

while (x <> y) {
  \{x = y + 2\}
  x := x - 1;
  \{x > y + 1\}
  y := y + 1
  \{x = y\}
} \{x = y\}

• Necessarily \( x \geq y \) for proper termination

Example (cont'd)

• Arithmetic mean of two integers \( x \) and \( y \) (cont'd)

while (x <> y) {
  \{x = y + 2k, x > y\}
  \{x = y + 2k, x = y + 2\}
  k := k - 1;
  \{x = y + 2k + 2, x > y + 2\}
  x := x - 1;
  \{x = y + 2k + 1, x > y + 1\}
  y := y + 1
  \{x = y + 2k, x > y\}
} \{x = y, k = 0\}
\begin{align*}
  &\text{assume (k = 0)} \\
  &\{x = y, k = 0\}
\end{align*}

• The difference \( x - y \) must initially be even for proper termination

Example (cont'd)

• Arithmetic mean of two integers \( x \) and \( y \) (cont'd)

while (x <> y) {
  k := k - 1;
  x := x - 1;
  y := y + 1
} \begin{align*}
  &\text{assume (k = 0)} \\
  &\{x = y, k = 0\}
\end{align*}

Hint: imagine \( k \) is the number of remaining steps to be done in the loop

Observations

• \( k \) provides the value of the variant function in the sense of Turing/Floyd

• The constraints on \( k \) (hence the variant function) are computed backwards

\[ \implies \text{a backward analysis should be able to infer the variant function} \]


The Turing-Floyd termination proof method

The ranking abstraction

\[ \alpha^{rk} \in \mathcal{P}(\Sigma \times \Sigma) \implies (\Sigma \not\ni \emptyset) \]
\[ \alpha^{rk}(r)s \triangleq 0 \text{ when } \forall s' \in \Sigma : \langle s', s' \rangle \notin r \]
\[ \alpha^{rk}(r)s \triangleq \sup \{ \alpha^{rk}(r)s' + 1 \mid \exists s' \in \Sigma : \langle s, s' \rangle \in r \land \forall s' \in \Sigma : \langle s, s' \rangle \in r \implies s' \in \text{dom}(\alpha^{rk}(r)) \} \]

- \( \alpha^{rk}(r) \) extracts the well-founded part of relation \( r \)
- provides the rank of the elements \( s \) in its domain
- strictly decreasing with transitions of relation \( r \)

\[ \implies \text{the most precise variant function} \]

Fixpoint definition of the variant function

We now apply the abstract interpretation methodology:

- The maximal trace semantics has a fixpoint definition
- The variant function is an abstraction of the maximal trace semantics
- With this abstraction, we construct a fixpoint definition of the abstract variant semantics

\[ \implies \text{Fixpoint induction provides a termination proof method} \]
\[ \implies \text{Further abstractions and widenings provide a static analysis method} \]
Where the transition abstraction terminates, we can say that the traces generated by a transition system terminate. So for the traces generated by a transition system, termination can be defined as follows:

$$\mathcal{R} \rightarrow (\mathcal{R} \rightarrow \mathcal{R})$$

(resp. $$\mathcal{W} \rightarrow (\mathcal{W} \rightarrow \mathcal{W})$$) is well-founded then the terminal semantics is well-founded.

Let $$\mathcal{R}$$ be the set of finite execution traces or $$\mathcal{W}$$ be the set of infinite execution traces. Then, the terminal semantics into a potential formula, $$\nu$$, of the form:

$$\nu(x) = \nu_0(x)$$

or

$$\nu(x) = \nu_0(x) \cup \nu(x + 1)$$

is definite. If the domain of $$\nu$$ is not the set of all execution traces, then it over-approximates the reachable states. There are two important conditions for the terminal semantics:

1. The set of atomic formulas must include all the potential formulas that can be derived from the program.
2. The set of logical operators must include all the logical operators that can be derived from the program.

Example I

- Maximal trace semantics:

- Ranking fixpoint iterates:

Example II

- Program:

  ```
  int x; while (x > 0) { x = x - 2; }
  ```

- Fixpoint:

  $$\nu = \text{lfp}^\nu \frac{\nu}{P}$$

- Iterates:

  $$\nu_0 = \emptyset$$

  $$\nu_1 = \lambda x \in [-\infty, 0] \cdot 0$$

  $$\nu_2 = \lambda x \in [-\infty, 0] \cdot 0 \cup \lambda x \in [1, 2] \cdot 1$$

  $$\nu_3 = \lambda x \in [-\infty, 0] \cdot 0 \cup \lambda x \in [1, 2] \cdot 1 \cup \lambda x \in [3, 4] \cdot 2$$

  $$\ldots$$

  $$\nu_\infty = \lambda x \in [-\infty, 0] \cdot 0 \cup \lambda x \in [1, 2 \times (n - 1)] \cdot (x + 1) \cdot \frac{1}{2}$$

  $$\ldots$$

  $$\nu^\alpha = \lambda x \in [-\infty, 0] \cdot 0 \cup \lambda x \in [1, +\infty] \cdot (x + 1) \cdot \frac{1}{2}$$

Example III

- Program:

  ```
  { even(x-y), x >= y }
  while (x <> y) {
    x := x - 1;
    y := y + 1
  } { x = y }
  ```

- Iterates (linear abstraction):

  $$\exists k : \nu(x, y) = k, x - y - 2k = 0, k \geq 0$$
Example IV

- In general, a widening is needed to enforce convergence.
- Program: int x; while (x > 0) { x = x - 2; }
- Iterates with widening:
  \[ v_i = x \in [-\infty, +\infty); +1 \]
  \[ v_i = x \in [-\infty, 0] \cup x \in (1, +\infty); \]
  \[ v_i = x \in [-\infty, 0] \cup x \in (1, 2] \cup x \in (3, +\infty); +1 \]
  \[ v_i = x \in [-\infty, 0] \cup x \in (1, 2] \cup x \in (3, 4]; \]
  \[ v_i = x \in \frac{1}{2} + 1 \]

Objection I: Turing/Floyd’s method goes forward not backward!

- An analysis can be inverted using auxiliary variables

  ```
  int x; while (c(x)) {
  x := f(x)
  }
  ```

  ```
  int x, x0; while (c(x)) {
  x0 := x;
  x := f(x)
  }
  ```

  Backward variant \( v \):
  \[ v(x_{\text{before}}) = v(x_{\text{after}}) + 1 \]
  \[ \iff v(x_{\text{before}}) = v(f(x_{\text{before}})) + 1 \]

  Forward variant \( v \):
  \[ v(x_0) = v(x) + 1 \]
  \[ \iff v(x_0) = v(f(x_0)) + 1 \]

Objection II: you need ordinals!

- Example: \( x := ?; \) while (\( x \geq 0 \)) do \( x := x - 1 \) od
- Ranking:

To avoid transfinite ordinals/well-founded orders for unbounded non-determinism, the computations need to be structured!

---

Floyd/Turing termination proof method

- Trivial postfix structuring of traces into segments

- Also used for termination of straight-line code (no need for variant functions)

Floyd with nested loops

- The trace semantics is recursively structured in segments according to loop nesting

Prove termination of outer loop assuming termination of body/nested inner loops

( equivalent to lexicographic orderings)

Hoare logic

- The trace semantics is recursively structured in segments according to the program syntax

- while (c) { b; a } ...

Burstall’s proof method by hand-simulation and a little induction

- Program 

  do odd(x) and x ≥ 3 → x := x+1
  □ even (x) and x ≥ 2 → x := x/2
  od

- Proof chart

  x' = x

  [odd(x) ∧ x ≥ 3] 
  [even(x) ∧ x ≥ 2] 
  Handsimulation

  [even(x) ∧ x ≥ 2 ∧ x' = x/2] 
  Handsimulation

  [odd(x) ∧ x ≥ 3 ∧ x' = x + 1] 
  Handsimulation

  [odd(x) ∧ x ≥ 3 ∧ x' = (x + 1)/2] 
  Handsimulation

  Theorem (since x/2 < x)

  Theorem (since x/2 < x)

  Theorem (since x/2 < x)
Well-founded tree structure of the trace segmentation

Podelski-Rybalchenko

- Transition invariants are abstractions of trace segments covering the trace semantics by their extremities

- Termination based on Ramsey theorem on colored edges of a complete graph, no recursive structure

Rely-guarantee

- Example of abstraction of segments into rely-guarantee/contracts state properties:

Burstall’s proof method by hand-simulation and a little induction

- Iterative program but recursive proof structure
- Inductive trace cover by segments
- Example:


Kedar Namjoshi and Amir Pnueli for their remarks on rank-abstraction.
Trace semantics segmentation

- **Recursive trace segmentation**

  **Definition 2.** An inductive trace segment cover of a non-empty set \( \chi \in \wp(\Sigma^{\infty}) \) of traces is a set \( C \in \wp(\chi) \) of sequences \( S \) of members \( B \) of \( \wp(\alpha^+(\chi)) \) such that

  1. if \( SS' \in C \) then \( S \in C \) (prefix-closure)
  2. if \( S \in C \) then \( \exists S' : S = \chi S' \) (root)
  3. if \( SBB' \in C \) then \( B \supseteq B' \) (well-foundedness)
  4. if \( SBB' \in C \) then \( B \subseteq \bigcup_{SBB' \in C} B' \) (cover).

- **Proof by induction** on the possibly infinite but well-founded trace segmentation tree

- **Orthogonal** to proofs on segment sets (using variant functions, Ramsey theorem, etc.)

---

Presentation based on our POPL‘2012 paper

- Patrick Cousot, Radhia Cousot: An abstract interpretation framework for termination. POPL 2012: 245-258

---

Conclusion

- The paper provides
  - More topics (e.g. general safety by abstract interpretation, abstract trace covers/proofs)
  - More technical details (e.g. fixpoint definitions of the various abstract termination semantics)
  - More examples (e.g. a more detailed piecewise linear termination abstraction)
Contributions

- Formalization of existing termination proof methods as abstract interpretations
- Pave the way for new backward termination static analysis methods (going beyond reduction of termination to safety analyzes)
- The new concept of trace semantics segmentation is not specific to termination and applies to all specification/verification/analysis methods

Future work

- Abstract domains for termination
- Semantic techniques for segmentation inference
- Eventuality verification/static analysis
- (General) liveness verification/static analysis