An Abstract Interpretation Framework for Termination

Patrick Cousot
CNRS, École Normale Supérieure, and INRIA, France
Courant Institute *, NYU, USA
cousot@ens.fr, pcousot@cims.nyu.edu

Radhia Cousot
CNRS, École Normale Supérieure, and INRIA, France
rcousot@ens.fr

Abstract

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.

The abstract interpretation design principle is first illustrated for the design of new forward and backward proof, verification and analysis methods for safety. The safety collecting semantics defining the strongest safety property of programs is first expressed in a constructive fixpoint form. Safety proof and checking/verification methods then immediately follow by fixpoint induction. Static analysis of abstract safety properties such as invariance are constructively designed by fixpoint abstraction (or approximation) to (automatically) infer safety properties. So far, no such clear design principle did exist for termination so that the existing approaches are scattered and largely not comparable with each other.

For (1), we show that this 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 definition of the best variant function. By further abstraction of this best variant function, we derive the Floyd/Turing termination proof method as well as new static analysis methods to effectively compute approximations of this best variant function.

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 assertions 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 assertions 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 assertions 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 assertions 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 assertions 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 assertions and static analysis methods by induction on both program structure, control, and data.

Categories and Subject Descriptors D.2.4 [Software/Program Verification]: D.3.1 [Formal Definitions and Theory]: F.3.1 [Specifying and Verifying and Reasoning about Programs]

General Terms Languages, Reliability, Security, Theory, Verification

Keywords Abstract Interpretation, Induction, Proof, Safety, Static analysis, Variant function, Verification, Termination

1. Introduction

Floyd/Turing program proof methods for invariance and termination [24, 40, 59] have inspired most sound static analysis methods. For static invariance analysis by abstract interpretation [19, 21], a key step is to express the strongest invariant as a fixpoint and next to approximate this strongest invariant to automatically infer an abstract inductive invariant using the constructive fixpoint approximation methods.

For static termination analysis, the discovery of variant functions is either decidable in limited cases [54] or else is based on the Floyd/Turing idea of variant functions into well-founded sets obtained by observing quantities that strictly decrease within loops while remaining lower-bounded, or dually. So most termination analysis methods indirectly reduce to a relational invariance analysis hence can reuse classical static analysis methods.

The abstract interpretation design principle is instantiated with suitable abstractions for safety and termination analysis, proof, and checking/verification (either potential termination or definite termination for nondeterministic systems).

The first main idea for termination is that there exists a most precise variant function that can be expressed in fixpoint form by abstract interpretation of a termination collecting semantics itself abstracting the program operational trace semantics. This yields new static analysis methods automatically inferring abstractions of that variant function by the constructive fixpoint approximation methods of abstract interpretation.

The second main idea introduced in this paper both for safety and termination is that of semantic structural induction, including termination proofs, over trace segment covers and their abstractions. Trace segments are more powerful than binary relations between states which have been used traditionally in program termination proofs (for example, the transition invariants used in [53] are binary relation abstractions of the set of trace segments). Examples include structural induction on the program syntax (including loop invariants à la Floyd [40]), induction on data, à la Burstall [3], the covering of the transition relation closure by well-founded relations, à la Podelski-Rybalchenko [53], their combinations and generalizations.

2. Fixpoints, fixpoint induction, abstraction, and approximation

We express semantics as fixpoints of maps \( f \in A \mapsto A \) i.e. elements \( x \in X \) such that \( x = f(x) \). We let \( \text{lfp}^\varphi \) be the least fixpoint of \( f \in A \mapsto A \) on the poset \( \langle A, \sqsubseteq \rangle \) greater than or equal to \( a \in A \), if any. The dual notion is that of greatest fixpoint \( \text{gfp}^\varphi \). We write \( \text{lfp}^\varphi a = \text{gfp}^\varphi a \) if \( a \in A \) is the infimum of \( A \), and \( \text{lfp}^\varphi f \) if the partial order \( \sqsubseteq \) is clear from the context. By Tarski/Pataraia’s fixpoint theorem [50, 58], the least fixpoint \( \text{lfp}^\varphi \) of \( f \) is the infimum of \( \{ f n \mid n \in \mathbb{N} \} \) of \( f \) increasing\(^1\) on a complete lattice \( \langle A, \sqsubseteq, a, \top, \bot, \lor, \land \rangle \) or on a cpo \( \langle A, \sqsubseteq, a, \bot \rangle \). The fixpoint iterates are \( f^0 = a, f^{n+1} = f(f^n) \), \( f^n \sqsubseteq \bigcup_{i<n} f^i \) which is \( \text{lfp}^\varphi f \) when \( a \sqsubseteq f(a) \) is a pre-fixpoint and \( f \) is continuous\(^2, 5, 6\). If \( f \) is increasing but not continuous, transfinite iterations may have to be used [22].

\(^1\) \( f \in A \mapsto A \) is increasing (also monotone, isotone,…) on a poset \( \langle A, \sqsubseteq \rangle \) if and only if \( \forall x,y \in A : (x \sqsubseteq y) \Rightarrow (f(x) \sqsubseteq f(y)) \).
\(^2\) A complete lattice \( \langle A, \sqsubseteq, \bot, \top \rangle \) is a poset s. t. any subset has a least upper bound \( \text{lub} \in \mathbb{L} \), hence a greatest lower bound \( \text{glb} \in \mathbb{L} \), i.e. \( \bot = \text{glb} \). A complete partial order \( \langle A, \sqsubseteq, \bot \rangle \) is a poset \( \langle A, \sqsubseteq \rangle \) such that any increasing \( C \subseteq A \) such that \( \forall x,y \in C : x \sqsubseteq y \Rightarrow C \) has a least upper bound \( \text{lub} \subseteq \mathbb{L} \), whence has an infimum \( \bot = \text{lub} \) for the empty chain.
\(^5\) A complete partial order \( \langle A, \sqsubseteq, \bot \rangle \) is a poset \( \langle A, \sqsubseteq \rangle \) such that any increasing \( C \subseteq A \) such that \( \forall x,y \in C : x \sqsubseteq y \Rightarrow C \) has a least upper bound \( \text{lub} \subseteq \mathbb{L} \), hence has an infimum \( \bot = \text{lub} \) for the empty chain.
\(^6\) \( f \in A \mapsto A \) is continuous on a poset \( \langle A, \sqsubseteq \rangle \) if and only if for all increasing \( C \subseteq A \) such that \( \text{lub} \subseteq \mathbb{L} \) does exist then the lub \( \text{lub} \subseteq \mathbb{L} \) exists and is such that \( \text{lub} \subseteq \mathbb{L} \).

\( 1 \) \( \text{fp}(f) \) or \( Z^f \) is the powerset of \( X \) i.e. the set of all subsets of a set \( X \).
\( 2 \) The post-image (or image) of \( X \in \text{fp}(A) \) by a map \( f \in A \mapsto B \) is \( f[X] = \{ f(x) \mid x \in X \} \in \text{fp}(B) \).
Fixpoint induction follows as a sound (⇒) and complete (⇒⇒) proof method since for all S ∈ A, \( \text{lp}^s \subseteq S \iff \exists P \in \text{A} \ such \ that \ \text{lp}^s \subseteq P \wedge P \subseteq S \).

S is called a specification or invariant and P is an inductive invariant.

The idea is to prove that an invariant S, one has to check (in checking/verification methods), to guess (in proof methods) or to compute (in analysis methods) a stronger inductive invariant P.

Following [19, 21], abstraction is formalized by Galois connections\(^7\) between posets (A, ⊆) and (B, ⊆) meaning that \( a \in A \rightarrow B, \ y \in B \rightarrow A \) and \( x \in A : \forall y \in B : a(x) \leq y \iff x \subseteq y(\gamma) \).

We write \( \langle A, \subseteq \rangle \xrightarrow{\gamma} \langle B, \subseteq \rangle \) when the abstraction \( \gamma \) is surjective (hence the concretization \( \gamma \) is injective), \( \langle A, \subseteq \rangle \xrightarrow{\gamma} \langle B, \subseteq \rangle \) when \( \gamma \) is injective (hence \( \gamma \) is surjective), and \( \langle A, \subseteq \rangle \xrightarrow{\gamma} \langle B, \subseteq \rangle \) when \( \gamma \) is bijective.

Given a concrete fixpoint characterization \( \text{lp}^c \) of program properties on complete lattices or cpos \( \langle A, \subseteq \rangle \) with \( a \subseteq f(a) \) and an abstraction \( \langle B, \subseteq \rangle \xrightarrow{\gamma} \langle A, \subseteq \rangle \), the sufficient commutation condition \( a \circ f = f \gamma a \) (resp. \( \text{semi-commutation condition} \ a = f \gamma a \)) implies the fixpoint abstraction \( \text{lp}^c = \text{lp}^a \gamma f \) (resp. fixpoint approximation \( \text{lp}^c = \text{lp}^a \gamma f \leq \text{lp}^c \gamma f \))[21]. The [semi-]commutation condition can be restricted to the iterates of f from a to the elements of which are \( \subseteq \)-less that equal to \( \text{lp}^c \). The result also holds when \( a \) is continuous [13]. In absence of existence of a best abstraction, similar results can be obtained using only one of the abstraction or concretization functions [26].

3. Transition semantics

We consider a programming language with nondeterministic programs P. The set of all states of P is \( \Sigma(P) \). The transition relation \( \tau(P) \subseteq \rho(\Sigma(P) \times \Sigma(P)) \) describes the possible transitions between a state and its immediate successor states during program execution [11, 21]. The program small-step operational semantics is the transition system \( (\Sigma(P), \tau(P)) \). When restricting to initial states \( \langle P \rangle \subseteq \rho(\Sigma(P)) \), we write \( (\Sigma(P), \langle P \rangle, \tau(P)) \). The termination/branching states are \( \beta(P) \triangleq \{ s \in \Sigma(P) : \forall s' \in \Sigma(P) : (s, s') \notin \tau(P) \} \). For brevity we write X for \( \langle X \rangle \) etc. (\( \Sigma, \tau, \Sigma, I, \tau \), or \( \beta \)).

4. Trace semantics

4.1 Traces

We let \( \Sigma^0 = \emptyset, \Sigma^1 = \bigcup_{n \geq 0} \Sigma^n \), \( \Sigma^+ = \bigcup_{n \geq 0} \Sigma^n \) and \( \Sigma^* = \bigcup_{n \geq 0} \Sigma^n \) be the set of all finite traces of length \( n \in \mathbb{N} \), non-empty finite, infinite, finite, infinite or infinite, in finite or infinite traces over the states \( \Sigma \) where \( \varepsilon \) is the empty trace.

We define the following operations on traces, writing \( \varepsilon \) for the length of the trace \( \sigma \in \Sigma^* \), \( |\sigma| \), \( 0 \leq n \leq m \) for the subtrace \( \sigma_{[0,n]}, \sigma_{[n+1]}, \ldots, \sigma_{[\min(m,|\sigma|)-1]} \) of \( \sigma \) and \( \sigma^\omega \) for the concatenation of \( \sigma, \sigma' \in \Sigma^\omega \) (with \( \sigma^\omega = \sigma \sigma' = \sigma r \) when \( \sigma \in \Sigma^\omega \)).

We define the following operations on sets of traces writing S for the set of traces \( \{ \sigma \in \Sigma^* \} \) of \( S \) made of one state of \( \Sigma^\omega \).

The traditional safety/observability program properties are relative to the trace property abstraction of the collecting semantics \( \alpha_{\rho}(\theta(\Sigma^\omega)) = \theta(\Sigma^\omega) \subseteq \rho(\Sigma^\omega) \).

Some program properties are not trace properties [5]. An example is “all program executions are deterministic” which is \( |\sigma| \)
6. Safety trace semantics

We now illustrate the classical abstract interpretation framework by generalizing invariance verification and static analysis to arbitrary safety properties. Safety properties are abstractions of program trace properties (essentially forgetting about liveliness properties).

6.1 Safety abstraction

The prefix abstraction of a set \( T \) of traces is the topological closure \[ pf(T) = \{ \sigma \in \Sigma^\omega \mid \exists \sigma' \in \Sigma^\omega : \sigma = \sigma'\omega' \} \]

The prefix abstraction expresses the fact that program executions can only be observed for a finite period of time (by \( pf(T) \)).

The limit abstraction of a set of traces is the topological closure \[ \lim(T) = T \cup \{ \sigma \in \Sigma^\omega \mid \forall n \in \mathbb{N} : \sigma[n, n+1] \in T \} \]

The limit abstraction expresses the fact that when observing program executions for finite periods of time it is impossible to distinguish between terminating and unbounded finite executions.

The safety abstraction of a set of traces is the topological closure \[ sf(T) = \lim^{-1}(pf(T)) \]

The safety abstraction provides the strongest program property resulting from finite observations of program executions (excluding the observation of infinite executions).

6.2 Safety trace properties

The safety properties are \[ SF \triangleq sf(\Theta(\Sigma^\omega)) = \{ sf(T) \mid T \in SF \} \triangleq \{ \sigma \in \Sigma^\omega \mid \exists \sigma' \in \Sigma^\omega : sf(T) = \sigma \} \]

We have the Galois isomorphism \[ (SF, \subseteq) \cong (\rho(\Sigma \times \Sigma), \subseteq) \]

where \( pf(T) = pf(\lim(T)) \) and so safety trace properties can equivalently be represented by their finite prefixes in Sect. 6.4 and 6.5.

6.3 Safety semantics

The safety semantics of a program \( P \) is its strongest safety property of any \( \omega \in \Sigma^\omega \)

6.4 Fixpoint safety semantics

It follows, by fixpoint abstraction, that the safety semantics of a program \( P \) with operational semantics \((\Sigma, \tau)\) is

\[ \tau^f[P] \triangleq \tau^\omega(\lim^{-1}(pf(T))) \]

\[ \tau^f[P] \triangleq \lim^{-1}(pf(\lim(T))) \]

\[ \tau^f[P] = \text{lfp} \phi^f \]

where \( \phi^f[P] = \lim^{-1}(pf(T)) \)

6.5 Proofs in the safety trace domain

By fixpoint induction, one immediately gets new forward and backward sound and complete safety proof methods\(^{15}\) generalizing invariance [37, 40, 48, 49]. For all safety specifications \( S \in SF \),

\[ sf(T) \subseteq S \iff \exists \sigma \in SF : \{ T \cup \{ \sigma \} \} \subseteq S \]

Observe that forward and backward safety semantics and proof methods are respectively equivalent. This property is preserved by relational abstractions in next Sect. 7, but this is not the general case (e.g. with abstractions of Sect. 7.6). [42] is an example of static analysis in the safety trace domain.

7. Invariance/reachability semantics

Invariance/reachability is an abstraction of safety and so invariance proof methods are abstractions of safety proof methods.

7.1 Relational abstraction

The relational abstraction \[ \langle SF, \subseteq \rangle \cong (\rho(\Sigma \times \Sigma), \subseteq) \] such that

\[ a^R(T) \triangleq \{ \langle \sigma_0, \sigma_{\text{next}} \rangle \mid \exists n > 0 : \sigma_0 \in \Sigma^n \cap T \} \]

\[ \gamma^R(R) \triangleq \{ \sigma \in \Sigma^n \mid \exists n > 0 : \forall \sigma' \in R : \langle \sigma, \sigma' \rangle \in R \}

abstracts traces by a relation between their initial and final states (so that intermediate computations are lost in that abstraction).\(^{16}\)

7.2 Relational invariance/reachability abstraction

Applied to a safety semantics which is prefix-closed, the relational abstraction provides a relation between initial and current states (where, in particular, “initial” can be any state).

The abstraction \( a^R \circ sf \) is therefore equal to the relational reachability abstraction \( \langle \rho(\Sigma^\omega), \subseteq \rangle \cong (\rho(\Sigma \times \Sigma), \subseteq) \) such that

\[ a^R(T) \triangleq \{ \langle \sigma_0, \sigma_{\text{next}} \rangle \mid \exists n > 0 \land \forall \sigma \in \Sigma^n \cap T \]

\[ \gamma^R(R) \triangleq \{ \sigma \in \Sigma^n \mid \exists n > 0 \land \forall \sigma' \in R : \langle \sigma, \sigma' \rangle \in R \}

abstracts traces by a relation between their initial and current states.

7.3 Relational invariance/reachability semantics

The relational invariance/reachability semantics of a program \( P \) is its strongest relational reachability property

\[ \tau^R(P) \triangleq a^R(\tau^\omega(\lim^{-1}(pf(T)))) \]

\[ \tau^R(P) \triangleq a^R(\tau^\omega(\lim^{-1}(pf(T)))) = a^R(\tau^\omega(\lim^{-1}(pf(T)))) = a^R(\tau^\omega(\lim^{-1}(pf(T)))) \]

7.4 Fixpoint relational invariance/reachability semantics

The commutation condition applied to the transformer of the safety semantics \( \tau^f[P] \) yields the fixpoint characterization of the relational reachability semantics of a program \( P \) with operational semantics \((\Sigma, \tau)\)

\[ \tau^R(P) = \text{lfp} \phi^R \]

where \( \phi^R \)

\[ \phi^R \]

\[ \phi^R \]

\[ \phi^R \]

\[ \phi^R \]

\[ \phi^R \]

\[ \phi^R \]

\[ \phi^R \]
Applying fixpoint induction to the fixpoint relational reachability semantics, we get sound and complete forward and backward proof methods for a specification \( S \in \varphi(\Sigma \times \Sigma) \) [23], respectively generalizing [40, 49] and [37, 48].

\[ \tau^R[\varphi](p) = \{ s \in S : \varphi(s, \sigma) \} \]

\[ \tau^R[\varphi](p) = \{ s \in S : \varphi(s, \sigma) \} \]

8.2 Termination trace abstraction

The termination trace abstraction eliminates the program execution traces not starting by a state from which execution may/must terminate.

Example 1. Consider the example of the non-deterministic program \( b : \{ 1: \text{loop} [\epsilon : \text{skip} ] \} \) with states \( \{ b, l, e \} \), transitions \( \{ (b, l), (b, e), (l, l) \} \) and complete trace semantics \( \{ b, e, blll, \ldots, illl \ldots \} \).

8.2.1 Potential termination trace abstraction

The potential termination or may-terminate trace semantics eliminates infinite traces.

Example 2. The potential termination trace semantics of program \( b : \{ 1: \text{loop} [\epsilon : \text{skip} ] \} \) in Example 1 is \( \{ b, e \} \) since an execution starting in state \( b \) may terminate (by choosing a transition to state \( e \)).

The corresponding potential termination abstraction is \( \varphi(\Sigma^\omega), \varphi(\Sigma^\omega), \varphi(\Sigma^\omega) \mapsto \varphi(\Sigma^\omega), \varphi(\Sigma^\omega), \varphi(\Sigma^\omega) \) where \( \varphi(\Sigma^\omega) \mapsto \varphi(\Sigma^\omega), \varphi(\Sigma^\omega), \varphi(\Sigma^\omega) \) where \( \varphi(\Sigma^\omega) \mapsto \varphi(\Sigma^\omega), \varphi(\Sigma^\omega), \varphi(\Sigma^\omega) \).

If the trace corresponds to Dijkstra’s weakest liberal/angelic precondition [37] it is considered in [11] (together with backward reachability) to automatically compute necessary conditions for termination (in Example 1, this analysis would yield the potential termination states \( \{ b, e \} \) proving definite non-termination in state \( l \)).

8.2.2 Definite termination trace abstraction

The definite termination or must-terminate trace semantics eliminates all traces potentially branching, through local non-determinism, to non-termination.

Example 3. The definite termination trace semantics of program \( b : \{ 1: \text{loop} [\epsilon : \text{skip} ] \} \) in Example 1 is \( \{ e \} \) since in state \( b \) there is a possibility of non-termination (by choosing a transition to state \( l \)).

A trace is in the definite termination semantics if and only if it is finite, independently of the potential non-deterministic choices along that trace. The corresponding definite termination abstraction is

\[ a^M(T) = \{ \sigma \in T | \varphi(\sigma) \cap \varphi(\Sigma^\omega) = \emptyset \} \]

\[ a^M(\varphi) = \{ \varphi(\Sigma^\omega), \varphi(\Sigma^\omega), \varphi(\Sigma^\omega) \} \mapsto \{ \varphi(\Sigma^\omega), \varphi(\Sigma^\omega), \varphi(\Sigma^\omega) \} \]

A trace is finite in the definite termination semantics if and only if it is finite, independently of the potential non-deterministic choices along that trace. The corresponding definite termination abstraction is

\[ a^M(T) = \{ \varphi(\Sigma^\omega), \varphi(\Sigma^\omega), \varphi(\Sigma^\omega) \} \mapsto \{ \varphi(\Sigma^\omega), \varphi(\Sigma^\omega), \varphi(\Sigma^\omega) \} \]

\[ a^M(\varphi) = \{ \varphi(\Sigma^\omega), \varphi(\Sigma^\omega), \varphi(\Sigma^\omega) \} \mapsto \{ \varphi(\Sigma^\omega), \varphi(\Sigma^\omega), \varphi(\Sigma^\omega) \} \]

A trace is finite in the definite termination semantics if and only if it is finite, independently of the potential non-deterministic choices along that trace. The corresponding definite termination abstraction is

\[ a^M(T) = \{ \varphi(\Sigma^\omega), \varphi(\Sigma^\omega), \varphi(\Sigma^\omega) \} \mapsto \{ \varphi(\Sigma^\omega), \varphi(\Sigma^\omega), \varphi(\Sigma^\omega) \} \]

\[ a^M(\varphi) = \{ \varphi(\Sigma^\omega), \varphi(\Sigma^\omega), \varphi(\Sigma^\omega) \} \mapsto \{ \varphi(\Sigma^\omega), \varphi(\Sigma^\omega), \varphi(\Sigma^\omega) \} \]

8.3 Termination trace semantics

The potential termination collecting semantics of a program \( P \) is therefore defined as

\[ a^M(T) = \{ \varphi(\Sigma^\omega), \varphi(\Sigma^\omega), \varphi(\Sigma^\omega) \} \mapsto \{ \varphi(\Sigma^\omega), \varphi(\Sigma^\omega), \varphi(\Sigma^\omega) \} \]

\[ a^M(\varphi) = \{ \varphi(\Sigma^\omega), \varphi(\Sigma^\omega), \varphi(\Sigma^\omega) \} \mapsto \{ \varphi(\Sigma^\omega), \varphi(\Sigma^\omega), \varphi(\Sigma^\omega) \} \]

8.4 Termination trace collecting semantics

The termination property states either that all executions in the trace semantics \( \Theta^{\varphi}(P) \) of a program \( P \) must always be finite

\[ \Theta^{\varphi}(P) = \{ s \in S : \varphi(s, \sigma) \} \]

or that the trace semantics \( \Theta^{\varphi}(P) \) may be finite (hence must not always be infinite)

\[ \Theta^{\varphi}(P) = \{ s \in S : \varphi(s, \sigma) \} \]

The infinite extension abstraction

\[ a^\omega(T) = T \cup \{ \sigma_1, \sigma_2 \in \Sigma^\omega : \varphi(\sigma_1, \sigma_2) \in \Sigma^\omega \} \]

is a topological closure and so \( \varphi(\Sigma^\omega), \varphi(\Sigma^\omega), \varphi(\Sigma^\omega) \). 

\[ a^\omega(T) = T \cup \{ \sigma_1, \sigma_2 \in \Sigma^\omega : \varphi(\sigma_1, \sigma_2) \in \Sigma^\omega \} \]

and so, if necessary, we only need to consider semantics closed by \( a^\omega \).
definite termination collecting semantics while the definite termination semantics of a program P is defined as

$$\tau^M[P] \triangleq \alpha^M(\tau^\omega[P])$$

8.4 Fixpoint termination trace semantics

By abstraction of the fixpoint trace semantics of Sect. 4.3, the strongest termination property of a program P with operational semantics ($\Sigma$|P| , $\tau$|P|) and termination states $\beta_1$|P| is

$$\tau^m[P] = \text{lfp}_\phi \phi^m[P]$$

potential termination

$$\phi^m[P]T \triangleq \beta_1[P] \cup \tau[P] \setminus T$$

$$\tau^M[P] = \text{lfp}_\phi \phi^M[P]$$

definite termination

$$\phi^M[P]T \triangleq \beta_1[P] \cup (\tau[P] \setminus T \cup (\neg \tau[P] \setminus \neg T))$$

where the term $(\neg \tau[P] \setminus \neg T)$ eliminates potential transitions towards non-terminating executions.

8.5 Proofs in the termination trace domain

Fixpoint induction provides formal methods to check fixpoint over-approximations, either $\tau^m[P] \subseteq S$ or $\tau^M[P] \subseteq S$. Over-approximations yield necessary but not sufficient termination conditions which may introduce spurious infinite traces for which the proof cannot be done. The proof method is therefore useful to prove invariance under termination assumptions\(^{19}\) but not for may/must termination.

On the contrary, termination proofs require fixpoint under-approximations $S \subseteq \tau^m[P]$ or $S \subseteq \tau^M[P]$. Under-approximations yield sufficient but not necessary termination conditions and so may eliminate some termination cases for which the termination proof could have been done automatically. Fixpoint under-approximation proof methods have been proposed e.g. by [15, Sect. 11] and would yield the requested termination proof methods. More classically, we will favor over-approximations for static analysis.

9. Termination domain

Programs may not always potentially/definitely terminate in all states. So one problem is to determine for which states $l \in \phi(S)$ do executions starting from these states may/must terminate.

9.1 Termination domain abstraction

This potential/definite termination domain semantics is provided by the weakest precondition abstraction $\phi(\Sigma^\omega)$, $\subseteq \xrightarrow{\forall} \phi(\Sigma)$, $\subseteq$ of the termination trace semantics, such that

$$\alpha^\omega(T) = \{x \mid \sigma \in T\}$$

precondition abstraction.

9.2 Termination domain semantics

$$\tau^{\text{wp}}[P] \triangleq \alpha^\omega(\tau^{\text{wp}}[P])$$

potential termination

$$\tau^{\text{aff}}[P] \triangleq \alpha^\omega(\tau^{\text{aff}}[P])$$

definite termination.

Using Dijkstra’s notations [37], $\tau^{\text{wp}}[P] = \text{wp}[P]\text{true}$ and $\tau^{\text{aff}}[P] = \text{wp}[P]\text{true}$.

9.3 Fixpoint termination domain semantics

By fixpoint abstraction of the termination trace semantics in Sect. 8.4 using transformer commutation, we get Dijkstra’s fixpoint weakest (liberal) termination precondition semantics [38]\(^{20}\)

8

9

10. Termination proofs for the trace semantics generated by a transition system

In practice a termination proof is decomposed in two parts. First a necessary termination condition is found by over-approximating $\tau^{\text{aff}}[P]$ or $\tau^{\text{aff}}[P]$. Then this necessary termination condition is shown to be sufficient by Floyd/Turing variant function method (e.g. [17]) or reversed (e.g. [8]). This corresponds to different abstractions, specific to the trace semantics generated by a transition system, that we now elaborate.

10.1 Transition-based termination proofs

A program which trace semantics is generated by a transition system $(\Sigma, \tau)$ definitely terminates if and only if the program transition relation is well-founded\(^{21}\)

$$\tau^{\omega}(\Sigma) \subseteq \tau^{\omega}(P) \iff (\exists \tau^{\omega}(P))$$

termination.

In practice one considers traces starting from initial states $l \in \phi(S)$, e.g. $l$ is the termination domain of Sect. 9. In that case a program which trace semantics is generated by a transition system $(\Sigma, \tau)$ definitely terminates for traces starting from initial states $l \in \phi(S)$ if and only if the program transition relation restricted to reachable states is well-founded.

$$\alpha(l)(\tau^{\omega}(P)) \subseteq \tau^{\omega}(P) \iff \langle \alpha(l)(\tau^{\omega}(P)) \rangle, \text{ is well-founded}$$

where the initialization abstraction $\phi(\Sigma^\omega), \subseteq \xrightarrow{\forall} \phi(\Sigma^\omega)$, $\subseteq$ is

$$\alpha(l)T \triangleq \{x \in T \mid \sigma_0 \in T\}$$

and the reachable states abstraction $\phi(\Sigma^\omega), \subseteq \xrightarrow{\forall} \phi(\Sigma^\omega)$ is

$$\alpha'(l) \triangleq \{s \mid \exists \sigma \in \Sigma', \sigma' \in \Sigma : \sigma s \sigma' \in T\}$$

reachability abstraction.

The transition-based termination proof method is sound and complete. As noticed in Sect. 9, the precondition $l$ can be inferred automatically by static analysis. Moreover, an over-approximation $R \supseteq \alpha(l)(\tau^{\omega}(P))$ is $\text{wp}[P][1]\text{true}$ of the reachable states can be computed by classical abstract interpretation algorithms [19].

\(^{19}\) A relation $x \in \phi(y \times z)$ on a set $\phi$ is well-founded if and only if there is no strictly decreasing infinite chain $x_0 > x_1 > \ldots > x_n > x_{n+1} > \ldots$ of elements $x_0, x_1, \ldots, x_n, x_{n+1}, \ldots$ of $\phi$. ($\phi$, $< \phi$ is called a well-founded set. A (total) well-order is well-founded (total) strict order relation $<$. The set of all well-founded relations in $\phi(y \times z)$ is written $\text{wp}[\phi(y \times z)]$.\(^{22}\)

\(^{22}\) $\text{wp}[\phi(y \times z)]$ is the reflexive transitive closure of a binary relation $\phi(y \times z)$.\(^{22}\)
10.2 Transition abstraction
If the program semantics $\Theta_0^\omega[P]$ is not generated by a transition system we might consider the transition abstraction $(\Sigma, \tau(\Theta_0^\omega[P]))$ where the transition abstraction $\langle \rho(\Sigma^\omega), \subseteq \rangle \xrightarrow{\gamma} \langle \rho(\Sigma \times \Sigma), \subseteq \rangle$ is
\[ \alpha(T) \triangleq \{ (s, s') \mid \exists \sigma, \sigma' : s \sigma s' \sigma' \in T \} \]
transition abstraction but the following counter-example shows that the condition is sufficient but not necessary.

Counter-example 5. Let $T \triangleq \{ab, ba\}$ be a trace semantics. The corresponding transition relation $\tau \triangleq \alpha(T) = \{(a, b), (b, a)\}$ generates the infinite trace $abababa$ and so the transition relation $\tau$ restricted to the reachable states $\{a, b\}$ is not well-founded. □

Another counter-example is fairness [35]. In the following, we consider complete/maximal trace semantics $T$ that are transition closed (also generated by a transition system) that is $\overline{\alpha(T)} = T$ or equivalently $T$ is closed by elimination of strict prefixes, closed by extension by fusion, and closed by limits [35, Th. 2.6.8].

11. Variant semantics
It remains to design verification and static analysis methods to show that $(R, \tau)$ is well-founded where
\[ R \supseteq \alpha'(\alpha'(I)(\tau_0^\omega[P])) = \tau[\gamma[P]] \]
over-approximates the reachable states. There are two important remarks.
1. If $\tau \subseteq R$ and $(R, \tau)$ is well-founded then $(R, \tau)$ is well-founded.
2. $(R, \tau)$ is well-founded if and only if there exists a variant function $v : \Sigma \rightarrow \nu$ into a well-founded set $(\nu, <)$ which domain is $R$.

So for the traces generated by a transition system, termination can be proved by mapping invariant states to a well-founded relation which is the principle of Floyd/Turing variant function method.

11.1 Variant function
A variant function $v : \Sigma \rightarrow \nu$ is a partial function from the set of states into a well-founded set $(\nu, <)$ where $<$ is a well-founded relation on the set $\nu$ (and $\nu$ is its non-strict version). With appropriate hypotheses on states and the transition relation, the co-domain of the variant function can be fixed a priori and the variant function can be found by constraint solving e.g. [17, 54]. However, these methods are not as general as Floyd/Turing’s method.

In mathematics, the ordinals provide a standard well-founded set thanks to ranking functions mapping each element of a well-founded set to its ordinal rank. So, up to a ranking function, the well-founded set $(\nu, <)$ can always be chosen as the class $(0, <)$ of ordinals. The intuition is that any execution $\sigma$ starting in a state $\sigma_0 \in \text{dom}(v)$ must terminate in “at most” $\nu(\sigma_0)$ execution steps while an execution $\sigma$ starting in a state $\sigma_0 \notin \text{dom}(v)$ might not terminate. We have $\tau \subseteq \{(s, s') \in \Sigma^2 \mid s \in \text{dom}(v) \land v(s) > v(s')\}$ and this relation is well-founded on states, proving termination.

11.2 Variant abstraction
A variant function is an abstraction of a set of finite traces. It is a partial function which domain is the set of terminating states. Its value is an upper bound of the remaining number of “steps” to termination. It may be transfinite for unbounded non-determination with unbounded execution trace lengths. Let us define
\[ a^\tau_x \in \rho(\Sigma \times \Sigma) \mapsto (\Sigma \not\ni b) \]
ranking abstraction
\[ a^\tau_x(s) \triangleq 0 \quad \text{when} \forall s' \in \Sigma \land (s, s') \notin \tau \]
\[ a^\tau_x(s) \triangleq \sup \{ a^\tau_x(s') + 1 \mid s' \in \text{dom}(a^\tau_x(r)) \land (s, s') \in \tau \} \]
\[ a^\tau_x(s) \] extracts the well-founded part of relation $\tau$ and provides the rank of the elements $s$ of its domain. $a^\tau_x(T)$ does the same for the transition relation by abstracting the set $T$ of finite traces
\[ a^\tau_x(s) \in \rho(\Sigma^\omega) \mapsto (\Sigma \not\ni b) \]
variant abstraction
\[ a^\tau_x(T) \triangleq A x \cdot a^\tau_x(\alpha(T)) s \]
This yields new termination proof methods and static analysis methods by abstraction of this fixpoint definition.

11.3 Variant semantics
A variant function can always be found by abstraction of the termination semantics into a variant semantics
\[ \tau_0^\forall[P] \triangleq \alpha'(r_0^\forall[P]) \]
potential termination variant
\[ \tau^\forall[P] \triangleq \alpha'(r^\forall[P]) \]
definite termination variant.

11.4 Fixpoint variant semantics
Fixpoint abstraction of the fixpoint termination trace semantics of Sect. 8.4, we get the fixpoint characterization of the variant semantics26,27
\[ \tau^\forall[P] = \text{fp}_P^\tau \text{fp}_P^\forall[P] \]
potential termination
\[ \text{fp}_P^\forall[P](\nu)(\nu) \triangleq \{ s \in \beta[P] \mid \exists z \sup \{ v(s') + 1 \mid s' \in \text{dom}(v) \land (s, s') \in \tau \} \}
\]
definite termination
\[ \text{fp}_P^\forall[P](\nu)(\nu) \triangleq \{ s \in \beta[P] \mid \exists z \sup \{ v(s') + 1 \mid s' \in \text{dom}(v) \land (s, s') \in \tau \} \}
\]
Example 6. Consider the trace semantics as represented on the right. We have represented below the fixpoint iterates for the corresponding potential and definite variant functions. Unlabelled states are outside the variant function domain.

25 This can be generalized from $(0, <)$ to well-orders $(\nu, <)$ using $\text{succ}(x) \triangleq \nu \in \nu \mid \forall x < y \land \exists z \in \nu \mid x < z < y$ and sup is an upper-bound. For ordinals $\text{succ}(x) = \{ x + 1 \}$ is the successor ordinal and sup is the lub.
26 The partial map $0 \in \Sigma \rightarrow \emptyset$ is totally undefined and has $\text{dom}(0) = \emptyset$.  
27 The conditional is \{ true $a \rightarrow b$ $\} \triangleq a$ and \{ false $a \rightarrow b$ $\} \triangleq b$.  

Example 7. The definite termination variant semantics $\text{lp}^{\text{ct}}_{\text{fr}}[P]$ of the following program $P$

\begin{verbatim}
int main () { int x; while (x > 0) { x = x - 2; }}
\end{verbatim}

is the limit $v^\infty$ of the iterates $v^n$, $n \in \mathbb{N}$ of the $\text{fr}^P[M][P]$ from 0. Considering only one loop head control point so that the state can be reduced to the value $x$ of $x$,

$$\text{fr}^P[M][P](v)(x) \triangleq \left\{ x \leq 0 \not\supset 0 \supset \supset v \geq v(x - 1) + 1 \not\supset x - 2 \in \text{dom}(v) \right\}.$$ 

The program being deterministic, the potential termination equation $v = \text{fr}^P[M][P](v)$ is similar. The fixpoint iterates are
\begin{align*}
  v^0 & = \emptyset \\
  v^1 & = \lambda x \in [-\infty, 0] \cdot 0 \\
  v^2 & = \lambda x \in [-\infty, 0] \cdot 0 \cup \lambda x \in [1, 2] \cdot 1 \\
  v^3 & = \lambda x \in [-\infty, 0] \cdot 0 \cup \lambda x \in [1, 2] \cup \lambda x \in [3, 4] \cdot 2 \\
  \vdots & = \vdots \\
  v^\infty & = \lambda x \in [-\infty, 0] \cdot 0 \cup \lambda x \in [1, +\infty) \cdot (x + 1) / 2.
\end{align*}

11.5 Termination proof method

The variant semantics is sound and complete to prove termination of a program $P$ for initial states $I$ since
\begin{align*}
  \alpha(1)(\text{fr}^P[M][P]) & \subseteq \Sigma'[P] \iff 1 \subseteq \text{dom}(\text{fr}^M[P]) \\
  \alpha(1)(\text{fr}^P[M][P]) & \cap \Sigma'[P] \not= \emptyset \iff 1 \subseteq \text{dom}(\text{fr}^P[M][P])
\end{align*}

Applying fixpoint induction to check for the least fixpoint overapproximation, we get a termination proof method. We have
\begin{align*}
  3\forall v \in \Sigma \not\supset 0: \text{fr}^P[M][P] & \subseteq v \land 1 \subseteq \text{dom}(v) \\
  2\exists v: \exists x, v \subseteq v', v \subseteq v, v \subseteq v' \land \text{fixpoint ind.} \\
  3\forall v: \text{fr}^P[M][P] & \subseteq v \land 1 \subseteq \text{dom}(v)
\end{align*}

For the program of Ex. 7, the definite termination proof
\begin{align*}
  \exists v: \forall x, v(x) & \not\supset 0 \supset x \supset x + 2 \\
  \exists x, v(x') & \supset x \supset x + 1 \\
  \forall x \not\supset x + 2 \supset x + 1 \iff (x + 1) / 2.
\end{align*}

A similar calculational design, yields the potential termination proof
\begin{align*}
  \exists v: \forall x, v(x) & \not\supset 0 \supset x \supset x + 2 \\
  \forall x \not\supset x + 2 \supset x + 1 \iff (x + 1) / 2.
\end{align*}

Observe that the fixpoint variant semantics of Sect. 11.4 is calculated backwards (the variant function increases on previous steps) but that the termination induction principles proceed forward (the variant function decreases on next steps).

Example 8. A similar induction principle is proposed in [35, Ch. 5.2.3] for relational inevitability proofs (a state must be reached that relates to the initial state as given by a specification relation $\Psi$). The following example is used in [35, Ch. 5.2.5] to show that the invariant and variant function must also be relational, that is relate the current and initial state; $\Sigma \triangleq \{1, 2, 3\}, 1 \triangleq \{1, 2\}, \tau \triangleq (x, x + 1) \mid x, x + 1 \in \Sigma, \Psi \triangleq \tau$. We can prove termination with assertions, no relational invariants being needed. For the above example, choose $I = \Sigma, (\langle t, \langle \rangle \rangle = (\langle t, \langle \rangle \rangle, \not\supset t(1) = 2, t(2) = 1), \not\supset t(2) = 0$. This example shows that termination proofs are simpler than inevitability proofs.

Example 9. For the program of Ex. 7, the definite termination proof for the simplified transition system
\begin{align*}
  \tau[P] \triangleq (x, x') \mid 0 \not\supset x > 0 \land x' = x + 1
\end{align*}

requires guessing $I = \Sigma, (\not\supset \langle \rangle \rangle = (\not\supset \langle \rangle \rangle \supset x \supset x + 1 \supset x + 1 / 2)$ and proving $\forall x, x' \in \Sigma: (x > 0 \land x' = x + 1) \iff (x' - x) = x + 1 \iff x'(x) < x(x)$. Because Turing/Floyd method uses the reachability abstraction $\alpha'$ of (4), it is not possible to directly relate states occurring at different times during computations. This is why the program is transformed by using auxiliary variables to relate the current values of the variables to their past values. This induces a transformed transition system, which under the reachability abstraction $\alpha'$ is equivalent to the relational abstraction of the original transition system by the relational abstraction (1).

Example 10. Continuing Ex. 9, the program is transformed into
\begin{verbatim}
int main () { int x, x; \\
  while (x > 0) { x0 = x; x = x - 2; }}
\end{verbatim}

which consists in reasoning on the transformed transition system.
which consists in reasoning on the transformed transition system, which under the reachability abstraction diverges at different times during computations. This is why the program is\footnote{More rigorously, we should write the dot product \( d^0(\tau) \cdot (x', 1) \).} equivalent to the relational abstraction of the original transition principle at that program point, which is a limitation when compared to the more flexible reasoning by induction on traces (see Sect. 15.3).

12. Variant abstraction analysis

We get a termination static analysis by abstracting the variant semantics. We need an abstraction \((\Sigma \mapsto \emptyset, \emptyset) \mapsto (\Lambda, \emptyset)\) of functions. Many abstractions of functions have been proposed e.g.\footnote{\cite{20, 30}} that can be reused for termination static analysis. As a simple example, we consider a piecewise linear variant abstraction. The purpose of this new abstract domain is to illustrate the abstraction of fixpoint definitions of variant functions with widening, many more abstractions being necessary to cover all practical cases.

12.1 Piecewise linear variant abstraction

Let us consider a program with integer variables \( X = x_1, \ldots, x_n \), \( n > 0 \). We first apply an abstraction of states extracting the numerical variables in the form of an environment \( \tilde{X} = \Sigma \mapsto (\Lambda \mapsto Z) \) so that, by composition, we are left with an abstraction \((\tilde{X} \mapsto Z) \mapsto (\emptyset \mapsto \emptyset) \mapsto (\Lambda, \emptyset)\). Encoding the partial map by a total map (using “\( \perp \)” for “undefined/not in the domain” and abstracting higher-order ordinals by “\( \perp \)” (“unknown/infinite”, e.g. in case of non-termination or unbounded nondeterminism), we can choose \((\tilde{X} \mapsto Z) \mapsto N \cup \{\perp\}\). There is no loss of information for bounded deterministic and unbounded executions are still allowed but disregarded by the abstraction. We can now further abstract by piecewise linear functions.

The values \( x_i \) of each variable \( x_i \in X \), \( i \in [1, n] \) are segmented into \( \ell_i^0 = -\infty \prec \ell_i^1 \prec \cdots \prec \ell_i^n = +\infty \). This provides a partition of the space \( Z^k \) of values \( x_1, \ldots, x_n \) of the variables \( x_1, \ldots, x_n \). The blocks of the partition are therefore \( [\ell_i^0, \ell_i^1), \ell_i^1, \ell_i^2), \cdots, \ell_i^n, \ell_i^0 + \infty \). In practice machine integers are bounded, in which case \( -\infty \) and \( +\infty \) are the smallest and largest machine integers. The number of blocks in the partitions can also be limited by widening thus favoring efficiency of the abstract domain to the detriment of precision.\footnote{The abstract domain is therefore (omitting the case of blocks with \( \perp \) for “not in the domain” and \( \top \) for “unknown”)\footnote{\cite{18, 19}}.

\[
A \triangleq \{ A \in Z^n \mid \prod_{i=1}^n \left( \ell_i^0 \leq x_i < \ell_i^{m_i + 1} \right) \ 
\text{dom} \sup_{\ell_i^k \in \Q} \left( \ell_i^k \leq x_i < \ell_i^{k+1} \right) \}
\]

\[
\forall i \in [1, n] : \ell_i^0 = -\infty \prec \cdots \prec \ell_i^k < \cdots < \ell_i^n = +\infty \wedge \ell_i^k = \nu_i \Rightarrow \nu_i \geq \nu_j \}
\]

When the \( \ell_i^k \in \Q \), \( i \in [1, n] \), \( j \in [1, m] \) is a rational, this abstraction essentially reuses the classical abstractions of intervals [18, 19], linear inequalities [31] and segmentation [33]. An immediate generalization consists in using consecutive segments with symbolic bounds as done in [33] for array content analysis. A further generalization consists in using decision trees [32] instead of a segmentation of the domain of the abstract variant function.

12.1.2 Piecewise linear abstract transformers

The abstract transformer \( \mathcal{B}_{\mathcal{F}^i}^m \) abstracting the concrete transformer \( \mathcal{F}_{\mathcal{F}^i}^m \) of Sect. 11.4 is applied blockwise by computing the abstract pre-image of each block by assignments or tests. The condition in tests may split the block into sub-blocks for which the condition is true or false.

Example 11. Here is an example of first iteration of the backward termination analysis of an exit preceded by a test. The initialization of the fixpoint iterates by \( A = (-\infty, +\infty) \) which indicates potential non-termination. The exit forces termination in 0 steps. The test splits the block \((-\infty, +\infty)\) into \([-\infty, 0]\) and \([1, +\infty]\).}
12.1.4 Piecewise linear variant abstract join

Similarly, the join $v_1 \Join v_2$ first unifies blocks of the partitioned domains of $v_1$ and $v_2$ into a common refined partition. Then the linear expressions are joined blockwise. This blockwise join $\Join_p$ is $d.x$ defined for each block $e_1^{i_1} \ldots e_1^{i_n}, i \in [1, n], j \in [1, m]$ of the partition such that $\forall i \in [1, n]$, $\forall x \in [e_i^{j_1}, e_i^{j_{m+1}}]$, $\forall a' \in Q^{m+1},$

\[
\begin{align*}
&d^{i_1} - e_1^{i_1} \ldots e_1^{i_n} \cdot x \leq d.x \\
&d^{i_1} - e_1^{i_1} \ldots e_1^{i_n} \cdot x \leq d'.x \implies d.x \leq d'.x.
\end{align*}
\]

Example 14.

A coarser partition can also be used in the join (as in [33, Sect. 11.4: Segmentation unification]) which is less precise but enforces faster convergence.

12.1.5 Piecewise linear variant abstract widening

Finally, the widening $v_1 \triangledown v_2$ follows the idea introduced in [20] of widening functions by widening the domain of their parameters with a domain widening $\triangledown_v$ and then their results with a range widening $\triangledown_v$. So the blocks of the partitioned domains of $v_1$ and $v_2$ are first widened using e.g. interval widening $\triangledown_v$ (possibly with thresholds) of the blocks with respect to their neighbors in all directions.

Example 15. An interval widening for a two-dimensional domain $(x, y) \in Z^2$ yields

Then the range-widening $\triangledown_p$ increases the gradient (i.e. slope in two dimensions) of the variant function of each block in the directions of its domain-widened neighbors to over-approximate their respective variants functions (extended to the widened domains).

Example 16.

To enforce convergence, the widening may have to skip to finitely many given thresholds of gradients before abandoning the constraint to $T$.

Example 17. We use two loop unrolls to stabilize iterations before widening [56].

\[
\begin{align*}
v_1^0 &= \lambda x \in [-\infty, +\infty] \cdot \perp \\
v_1^1 &= \lambda x \cdot \{ x \in [-\infty, 0] \cap 0 \leq x \leq 1 \} \\
v_1^2 &= \lambda x \cdot \{ x \in [-\infty, 0] \cap 0 \leq x \leq 1 \cup x \in [1, 2] \cup 1 \leq x \leq 3, +\infty] \cdot \perp \\
v_1^3 &= \lambda x \cdot \{ x \in [-\infty, 0] \cap 0 \leq x \leq 1 \cap 1 \leq x \leq 3, +\infty] \cap 2 \leq x \leq 5, +\infty] \cdot \perp \\
v_1^4 &= \triangledown v_1^0 \triangledown v_1^1 \\
v_1^5 &= \lambda x \cdot \{ x \in [-\infty, 0] \cap 0 \leq x \leq 1 \cap 1 \leq x \leq 3, +\infty] \cap 2 \leq x \leq 5, +\infty] \cdot \perp \\
v_1^6 &= \lambda x \cdot \{ x \in [-\infty, 0] \cap 0 \leq x \leq 1 \cap 1 \leq x \leq 3, +\infty] \cap 2 \leq x \leq 5, +\infty] \cdot \perp \\
v_1^7 &= \lambda x \cdot \{ x \in [-\infty, 0] \cap 0 \leq x \leq 1 \cap 1 \leq x \leq 3, +\infty] \cap 2 \leq x \leq 5, +\infty] \cdot \perp \\
v_2^0 &= \lambda x \cdot \{ x \in [-\infty, 0] \cap 0 \leq x \leq 1 \cap 1 \leq x \leq 3, +\infty] \cap 2 \leq x \leq 5, +\infty] \cdot \perp \\
v_2^1 &= \lambda x \cdot \{ x \in [-\infty, 0] \cap 0 \leq x \leq 1 \cap 1 \leq x \leq 3, +\infty] \cap 2 \leq x \leq 5, +\infty] \cdot \perp \\
v_2^2 &= \lambda x \cdot \{ x \in [-\infty, 0] \cap 0 \leq x \leq 1 \cap 1 \leq x \leq 3, +\infty] \cap 2 \leq x \leq 5, +\infty] \cdot \perp \\
v_2^3 &= \lambda x \cdot \{ x \in [-\infty, 0] \cap 0 \leq x \leq 1 \cap 1 \leq x \leq 3, +\infty] \cap 2 \leq x \leq 5, +\infty] \cdot \perp \\
v_2^4 &= \lambda x \cdot \{ x \in [-\infty, 0] \cap 0 \leq x \leq 1 \cap 1 \leq x \leq 3, +\infty] \cap 2 \leq x \leq 5, +\infty] \cdot \perp \\
v_2^5 &= \lambda x \cdot \{ x \in [-\infty, 0] \cap 0 \leq x \leq 1 \cap 1 \leq x \leq 3, +\infty] \cap 2 \leq x \leq 5, +\infty] \cdot \perp \\
v_2^6 &= \lambda x \cdot \{ x \in [-\infty, 0] \cap 0 \leq x \leq 1 \cap 1 \leq x \leq 3, +\infty] \cap 2 \leq x \leq 5, +\infty] \cdot \perp \\
v_2^7 &= \lambda x \cdot \{ x \in [-\infty, 0] \cap 0 \leq x \leq 1 \cap 1 \leq x \leq 3, +\infty] \cap 2 \leq x \leq 5, +\infty] \cdot \perp \\
\end{align*}
\]

The over-approximation of $v$ in Ex. 7, by $v_A$ is as follows

\[
\begin{align*}
\forall (x, y) \in \mathbb{R}^2, v_A(x) = \min\{v(x), v(y)\}
\end{align*}
\]

Notice that the domain of termination is widened which is an over-approximation which might include non-termination cases. However, the iterates with widening stop at a post-fixpoint $v_A$

\[
\text{which, by definition of the abstract partial order } \preceq_v \text{ ensures that } v_A \text{ is decreasing on blocks for which it is defined. Termination is therefore proven for blocks with either 0 or a strictly decreasing variant. By undecidability, there might be blocks which variant value is } T \text{ indicating insufficient precision to conclude.}
\]

12.2 Non-linear variant abstraction

Besides classical linear relational abstractions (e.g. octagons [46], polyhedra [31], etc.) which can be used pointwise as in Sect. 12.1, the variant function in each block of the partition can also be non-linear (e.g. polynomials [47], exponentials [39], etc.).

13. Relational variant semantics

To use relational abstractions for static termination analysis, we can further abstract variant functions into relations.

13.1 Relation variant abstraction

A variant function $v$ can be abstracted as the pair of an abstraction of its domain $\text{dom}(v)$ by a set abstraction (such as e.g. intervals) and an abstraction of its value by (a relational abstraction of) the down-closed relation $r$ which over-approximates the variant function on its domain that is $\forall s \in \text{dom}(v), r = \{ (s, w) \mid w \in \mathbb{R} \}$

The abstraction is therefore (the first component is redundant but useful for static analysis)

\[
\alpha^v(v) \triangleq (\text{dom}(v), \alpha^v(\{ (s, v(s)) \mid s \in \text{dom}(v) \}))
\]

where the down-closure of a relation $r \in \phi(\Sigma \times \mathbb{R})$ is

\[
\alpha^v(r) \triangleq \{ (s, w') \mid \exists w : w' \leq w \wedge (s, w) \in r \}.
\]

Observe that the effect of the down-closure is to replace equalities by inequalities for which numerous abstract domains are available. Moreover, an over-approximation of the first component is known by Sect. 9 but for correctness we either need an under-approximation or prove termination for this over-approximation, which is the usual option. For the second component, an over-approximation is correct (this over-estimates the termination time). We have\(^{31}\)

\[
\begin{align*}
(\Sigma \triangledown \mathcal{V}, \mathcal{V}^v) \xrightarrow{\text{def}} (\phi(\Sigma) \times \mathbb{R}) \triangledown \alpha^v(\phi(\Sigma \times \mathcal{V}^v)) \subseteq \mathcal{V}^v.
\end{align*}
\]

13.2 Relation variant semantics

The relational variant semantics of a program $P$ is

\[
\begin{align*}
\mathcal{R}^{\text{mv}}(P) \triangleq \alpha^v(\mathcal{R}^{\text{mv}}(P)) \quad \text{potential termination relational variant} \\
\mathcal{R}^{\text{mv}}(P) \triangleq \alpha^v(\mathcal{R}^{\text{mv}}(P)) \quad \text{definite termination relational variant.}
\end{align*}
\]

\(^{31}\underline{x} \subseteq \mathcal{V}^v\) is the componentwise partial order $(x, y) \subseteq (x', y') \iff x \subseteq x' \wedge y \subseteq y'$. 253
13.3 Fixpoint relational variant semantics

By fixpoint abstraction of the fixpoint variant semantics of Sect. 11.4, we get, by calculational design, the fixpoint definite and potential relational variant semantics\(^3\):

\[ \phi_{\text{ravv}}[P] = \mu_{\phi_0 \cap \Sigma} \phi_{\text{ravv}}[P] \]

potential termination

\[ \phi_{\text{ravv}}[P](D, r) \triangleq \text{let } D' = D \cup \beta_1[P] \cup r \tau[P]^{-1}[D] \text{ in } \langle D', \{ (s, 0) \mid s \in \beta_1[P] \} \cup \{ (s, \rho + 1) \mid \wedge \exists s': (x', s') \in r \tau[P] \wedge s' \in D' \wedge (x', \rho) \in r \rangle \}

\[ \phi_{\text{ravv}}[P] = \mu_{\phi_0 \cap \Sigma} \phi_{\text{ravv}}[P] \]

definite termination

The over-approximation of \( D \) is classical in static analysis\(^{19,21} \) so we concentrate on the over-approximation of the relational variant \( r \).

14. Transition-based termination analysis

We consider the case when states \( s \in \Sigma \) consist of a pair \((\xi, \mu)\) of a control state \( \xi \) (used for state or trace partitioning) and a memory state \( \mu \). The memory state maps variables \( x \in \Xi \) to numerical values \( \mu(x) \in Z \) (for simplicity all other types are ignored in the examples).

We consider a relational abstraction \((\alpha^r[\phi(\Sigma \times \nu)]) \subseteq \frac{1}{\delta}s \subseteq A \subseteq \phi)\) of the fixpoint relational variant semantics of Sect. 13.2. In practice, we choose \( \delta s = s \) and adjoint an extra variable \# to contain the value of \( \mu \).

We can use octagons\(^{46} \), polyhedra\(^{31} \), polynomials\(^{47} \), exponentials\(^{39} \), their numerous variants, possibly partitioned on states\(^{12} \), traces\(^{56} \), or conditions of decision trees\(^{32} \).

Example 18. Consider the program of Ex. 7, where a forward interval analysis has determined the invariants given as comments.

```plaintext
int main () { int x;  
  /* x:[-2147483648, 2147483648] */ 
  while (x > 0) {  
     if (x + 1 < 2147483647) {  
       /* x:[1, 2147483647] */  
       x = x - 2;  
     }  
     else {  
       /* x:[-1, 2147483645] */  
     }  
   }  
  /* x:[-2147483648, 0] */ 
}
```

The abstraction of the fixpoint equations of Sect. 13.3 is given below in logical form (representing a set by its characteristic predicate) with restriction to the reachable states over-approximated by the interval analysis.

\[ r(x, \#) \triangleq (-2147483648 \leq x \leq \#) \lor (\exists x', \#': x \in [1, 2147483647] \wedge x' = x - 2 \land \# \leq \#' + 1 \land r(x', \#')) \]

Inverting the assignment yields the classical simplification

\[ r(x, \#) \triangleq (-2147483648 \leq x \leq \#) \lor (\exists x': x \in [1, 2147483647] \land \# \leq \#' + 1 \land r(x-2, \#')) \]

Partitioning into \( r_1(x, \#) = r(x, \#) \land x \leq 0 \) and \( r_2(x, \#) = r(x, \#) \land x > 0 \), the iterates for \( r_1(x, \#) \) immediately converge while the iterates for \( r_2(x, \#) \) abstracted with octagons\(^{46} \) are

\[ r^1_2(x, \#) = \text{false} \]
\[ r^2_2(x, \#) = \exists x : [1, 2147483647] \land \# \leq \#' + 1 \land r_1(x - 2, \#') \]
\[ r_{1}^{(2)}(x, \#) = (x = 1 \land \# \leq 1) \lor (x = 3 \land \# \leq 2) \]
\[ r^2_2(x, \#) = \exists x : [1, 2147483647] \land \# \leq \#' + 1 \land (r_1(x - 2, \#') \lor r_2(x - 2, \#')) \]
\[ r^3_2(x, \#) = (x = 1 \land \# \leq 1) \lor (2 \leq x \leq \# + 1 \land \# \leq 3) \]
\[ r^0_2(x, \#) = (x = 1 \land \# \leq 1) \land \# \leq 3 \]

proving termination since \# strictly decreases around the loop and remains positive. Of course direct resolution methods\(^{17, 54} \) would find the same result. However tests are excluded within loops in\(^{54} \) while the presence of tests is not impairing the above octagon abstraction or the piecewise linear variant abstraction of Sect. 12.1.

For example, the loop body if \( x \) (odd \( x \)) \( \{ x = x - 1; \} \) else \( \{ x = x + 1; \} \) with state partitioning on the conditional branches yields the same results.

15. Semantic structural induction

Semantic structural induction is by induction on the structure of computations as opposed to transitional verification based on an induction on the program steps as in Floyd/Turing method\(^{40, 59} \). This point of view generalizes syntactic structural induction on program syntax as in Hoare logic\(^{43} \), replacing the syntactic by a semantic point of view using the concept of structural inductive cover. We start by the simple case of structuring states in next Sect. 15.1 before generalizing to the more concrete trace computations in Sect. 15.3 and their abstractions in Sect. 16.

15.1 Inductive state cover

Many inductive formal definitions and verification methods can be formalized in a language-independent way by an inductive cover of the set \( \Sigma \) of states (examples are given in next Sect. 15.2).

Definition 1. An inductive state cover of a non-empty set \( \chi \subseteq \phi(\Sigma) \) of states is tree encoded as a set \( C \subseteq \chi \) (of finite) sequences \( S \) of non-empty members \( B \subseteq \phi(\chi) \setminus \{ 0 \} \) such that

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

By the prefix-closure condition Def. 1.1, the inductive cover is a tree (so that proofs based on the cover \( C \) are by case analysis on the tree width and induction on the tree depth). By the root condition Def. 1.2, the tree is rooted at \( \chi \) (which ensures that inductive proofs based on the cover \( C \) are valid for \( \chi \)). By the strictly-decreasing condition Def. 1.3, the sequences \( S \) are necessarily finite so the immediate component relation between a node of the tree and its sons is well-founded. It follows that proofs on states can be done by induction on this well-founded relation. And, by the covering condition Def. 1.4, the states in a node are covered by the join of the states in its sons (which ensures that proofs based on the cover \( C \) do not forget any possible case). Inductive state covers are abstractions of inductive trace covers introduced in forthcoming Sect. 15.3 but are introduced first for simplicity. An example is\(^{45} \).

\(^3\)The dual pre-image of \( Y \in \phi(\Sigma) \) by a relation \( r \in \phi(\mathcal{A} \times \mathcal{B}) \) is \( r^{-1}[Y] \triangleq -r^{-1}[-Y] \) also written \( \text{pre}[r][Y] \).
15.2 Examples of semantic structural induction

15.2.1 Loop invariants and variants

In Floyd’s total correctness proof method, one typically provides a loop invariant and a loop variant function for termination. It is not necessary for the variant function to strictly decrease at each program step but only once around each loop iterate. This corresponds to a cover of the states of the loop according to their control component which induces a decomposition of executions into trace segments for the loop containing trace segments for the loop body considered as one step in the inductive reasoning on loop iterations.

Moreover a different variant function is used for each loop so that this decomposition is applied recursively for nested loops.

15.2.2 Hoare logic

Inductive definition/verification in the form of structural induction on the program syntax originates from axiomatic semantics [43], denotational semantics [57], and operational semantics [51].

Hoare logic for a structured imperative language [43], and its extension to total correctness [44], can be understood as the inductive state cover based on the control states of a command (ignoring its memory states). For example, a while loop can be covered by the states which control is in the condition and the states which control is in the loop body. The states of the loop body can themselves be covered recursively, by structural induction on the program syntax. This structural induction on the program syntax can be understood as induction on a state cover which itself induces a cover of the execution traces by segments which states are in a block of the state cover. A termination proof by structural induction on the program syntax [44] has the advantage, a.o., to be able to handle unbounded non-determinism without requiring transfinite ordinals (equivalent to a lexicographic ordering on nested loops).

The inductive cover contains the program P, the hand-simulation/symbolic execution blocks P HS1, P HS2, P HS3, and two lemmas with respective blocks P L1, P L1 L2−1, P L1 L2−1 P L2, P L2, P L2 L3−1, ..., P L2 L3−1 L4−1 L3−1, corresponding to proofs by recurrence on the data with respective ranks i and j.

Observe that the termination analysis method of [9] can be seen as implicitly relying on Burstall’s proof method.

15.3 Trace-based semantic structural induction

The previous examples of Sect. 15.2 show the need to go beyond purely syntactic, language-dependent induction and that induction on states can be generalized to induction on trace segments. Consequently, we introduce a general form of inductive reasoning on the semantic structure of computations, first starting by induction on blocks of trace segments and then their abstractions in Sect. 16.

15.3.1 Trace segment abstraction

We first observe that considering segments of traces is an abstraction. The segment abstraction \( \phi(\Sigma^\omega) \bowtie \{ \alpha \mid \Sigma^\infty \bowtie \alpha \} \)

is the set of segments of traces of T. If T, T’ \( \bowtie \phi(\Sigma^\omega) \), we define \( T \subseteq T’ \) \( \bowtie \forall \alpha \bowtie \phi(\Sigma^\omega) \bowtie \alpha \bowtie T’ \) to mean that all traces of T are segments of traces of T’. We define the join \( \bigcup_{i \in \Delta} T_i \bowtie \bigcup_{i \in \Delta} \gamma(T_i) = \{ \sigma_i \ldots \sigma_n \mid \forall k \in [1, n] : \sigma_k \in T_{i,k} \} \)

to be the set of all the traces made out of segments in the \( T_i \), \( i \in \Delta \).

15.3.2 Inductive trace segment cover

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

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

Example 19. An example of inductive trace segment cover is trace partitioning [56].

Example 20. A variant function \( \nu \in \Sigma \bowtie \varphi \bowtie \mathbb{N} \) defines a trivial inductive trace cover. Each value \( \nu \in \text{codom}(\nu) \) defines segments starting with states \( \sigma \) such that \( \nu(\sigma) = \nu \) of length at most \( \nu \).

The following definitions are classical for trees \( C \in \mathcal{E}(\chi) \).

- \( \text{root}(C) \bowtie \chi \)
- \( \text{leaves}(C) \bowtie \{ B \in \phi(\chi) \mid \exists S : SB \in C \land \forall S’ : SBB’ \not\in C \} \)
- \( \text{inner}(C) \bowtie \{ B \in \phi(\chi) \mid \exists S, B, S’ : SBB’S’ \in C \} \)
- \( \text{nodes}(C) \bowtie \text{leaves}(C) \cup \text{inner}(C) \)
- \( \text{sons}_C(B) \bowtie \{ B’ \in \text{nodes}(C) \mid \exists S, S’ : SBB’S’ \in C \} \).
The immediate component relation \( B' \succcurlyeq_c B \triangleq B' \in \text{sons}_c(B) = \exists B' \in C \text{ is well-founded, so that proofs on segments can be done by induction on this well-founded relation. The component relation } \succcurlyeq_c \text{ is its reflexive transitive closure. The blocks of a cover } C \text{ are nodes}(C) \triangleq \{ B \in \varphi(\Sigma) \mid B \succcurlyeq_c \Sigma \}.

15.4 State cover induced by an inductive trace cover

Given an inductive trace cover \( C \in \mathcal{C}(\chi), \chi \in \varphi(\Sigma^{*}) \) of Def. 2, define the abstractions

\[
\begin{align*}
\alpha^b(\chi) & \triangleq \{ \alpha^b(S) \mid S \in C \} \quad C \in \varphi((\varphi(\alpha^c(S))^{*}) \\
\alpha^b(SS') & \triangleq \alpha^b(S) \alpha^b(S') \quad S, S' \in (\varphi(\alpha^c(S))^{*} \\
\alpha^b(B) & \triangleq \{ \alpha^b(\sigma) \mid \sigma \in B \} \quad B \in \varphi(\alpha^c(S)) \\
\alpha^b(\sigma) & \triangleq \{ \sigma_{i} \mid i \in [0, |\sigma| - 1] \} \quad \sigma \in \alpha^c(S). 
\end{align*}
\]

Then \( \alpha^b(\chi) \) is an inductive state cover in the sense of Def. 1.

15.5 Trace cover induced by an inductive trace cover

Inversely, given an inductive state cover \( C \in \mathcal{C}(\chi), \chi \in \varphi(\Sigma) \) of Def. 1, define

\[
\begin{align*}
\gamma^b(\chi) & \triangleq \{ \gamma^b(S) \mid S \in C \} \quad C \in \varphi((\varphi(\gamma(\chi))^{*}) \\
\gamma^b(SS') & \triangleq \gamma^b(S) \gamma^b(S') \quad S, S' \in (\varphi(\gamma(\chi))^{*} \\
\gamma^b(B) & \triangleq B^* \quad B \in \varphi(\gamma(\chi)) \\
\gamma^b(\sigma) & \triangleq \{ \sigma_{i} \mid i \in [0, |\sigma| - 1] \} \quad \sigma \in \gamma^c(S). 
\end{align*}
\]

We have \( (\varphi((\varphi(\chi))^{*})^{*}), \subseteq \xrightarrow{\alpha^b} (\varphi((\varphi(\chi))^{*}), \subseteq) \) and \( \gamma^b(\chi) \) is an inductive trace cover of \( \chi^* \).

15.6 Syntactic trace cover

Similarly one can define the inductive state cover induced by the syntax of commands of a programming language by considering the states which control is in a given command. This in turns induces a trace cover which is the basis for e.g. Hoare logic or structural static analysis by induction on program commands, as opposed to induction on program transitions as in dataflow analysis.

15.6.1 Inductive proof method

We have a sound and complete inductive proof method of a semantic property \( \Theta^{\omega}[P] \subseteq \chi \in P \) for an inductive trace cover \( C \in \mathcal{C}(\chi) \)

\[
\Theta^{\omega}[P] \cap B \in P, \quad B \in \text{inner}(C)
\]

\[
\forall B' \in \text{sons}_c(B) : \Theta^{\omega}[P] \cap B' \in P, \quad B \in \text{inner}(C)
\]

In particular, for termination \( \tau^{\omega}[P] \subseteq \Sigma^* \cup P \) with a trace cover \( C \in \mathcal{C}(\Sigma^* \cup P) \), we get

\[
\Theta^{\omega}[P] \subseteq B \subseteq \Sigma^*, \quad B \in \text{inner}(C)
\]

\[
\forall B' \in \text{sons}_c(B) : \Theta^{\omega}[P] \subseteq B' \subseteq \Sigma^*, \quad B \in \text{inner}(C)
\]

Example 21. Another form of decomposition of reasonings on termination is proposed by the transition invariant proof method of Podelski-Rybalchenko [53] based on a relational semantics [15].

The transition invariants proof method of [53] can be seen as the \( \alpha^b \) abstraction of the above inductive proof method based on an inductive trace cover of height 1 with root \( \Sigma^* \cup P \) and sons \( \alpha^c(T_i), \ldots, \alpha^c(T_n) \) where \( T_1, \ldots, T_n \in \varphi(\Sigma^* \cup P) \) such that

\[
\Theta^{\omega}[P] \subseteq \Sigma^* \cup P \iff \forall i \in [1, n] : \Theta^{\omega}[P] \cap T_i \subseteq \Sigma^*.
\]

The generalization by inductive trace covers is both on the use of trace segments (instead of their relational abstraction of Sect. 7.1), and the possibility of recursive application of the method by induction, including on data, à la Burstell [3].

16. Abstract semantic structural induction

Assume that we can prove a program trace property in the concrete using an inductive trace cover. Can we prove an abstract program property using the abstraction of the inductive trace cover? We have seen an example in Sect. 15.5. The question is whether this observation is general.

16.1 Abstract inductive cover

Definition 3. An inductive abstract cover of a trace semantics \( \chi \in \varphi(\Sigma^{*}) \) is an element \( C \in A_c \) of an abstract domain \( A_c \) such that

\[
(\varphi((\varphi(\chi))^{*}), \subseteq) \xrightarrow{\alpha^b} (A_c, \subseteq)
\]

and \( \gamma^b(\chi) \) is an inductive trace cover of \( \chi \).

A standard way to define such inductive abstract covers is to follow the example of Sect. 15.5 generalized to a block abstraction \( \langle \varphi((\varphi(\chi))^{*}), \subseteq \rangle \xrightarrow{\alpha^b} \langle A_B, \subseteq \rangle \). We get the cover abstraction

\[
(\varphi((\varphi(\chi))^{*}), \subseteq) \xrightarrow{\alpha^b} (A_B(\chi), \subseteq) \quad \text{by generalizing } \alpha^b \text{ to sequences of abstract blocks and sets of such abstract sequences as follows}
\]

\[
\gamma^b(SS') \triangleq \gamma^b(S) \gamma^b(S') \quad S, S' \in (A_B) \\
\gamma^b(B) \triangleq (\gamma^b(S) \mid S \in C) \quad C \in \varphi(A_B(\chi)).
\]

Then \( A_c \) is chosen to be the set of elements \( C \in \varphi(A_B(\chi)) \) of sequences \( S \) of members of \( A_B \) such that

1. if \( SS' \in C \) then \( S \in C \) (prefix-closure) 
2. if \( S \in C \) then \( 3S' : S = a^b(\chi)S' \) (root) 
3. if \( SBB'C \in C \) then \( \gamma^b(B) \supseteq \gamma^b(B') \) (well-foundedness) 
4. if \( SBB'C \in C \) then \( \gamma^b(B) \subseteq \gamma^b(\chi) \subseteq \gamma^b(B') \) (cover).

It follows that any \( C \in A_c \) is an inductive abstract cover of the trace semantics \( \chi \in \varphi(\Sigma^{*}) \) in the sense of Def. 3.

Example 22. The transition invariant proof method of [53] follows from the relational abstraction \( (\varphi(\Sigma^*) \times \Sigma^*) \subseteq \varphi(\Sigma^*) \) [15] where \( a^b(B) \triangleq \{(\sigma_{n}, \sigma_{n-1}) \mid n > 0 \land \sigma \in B \cap \Sigma^* \} \) is limited to the trace covers of the form given in Ex. 21.

16.2 Abstract inductive proof

The inductive proof method of Sect. 15.6.1 can be abstracted as follows.

\[
\forall B' \in \text{sons}_c(B) : a^b(\Theta^{\omega}[P]) \subseteq B', \quad B \in \text{inner}(C)
\]

\[
\forall B' \in \text{sons}_c(B) : a^b(\Theta^{\omega}[P]) \subseteq B', \quad B \in \text{inner}(C)
\]

The proofs \( a^b(\Theta^{\omega}[P]) \subseteq B \) can be done by the abstract by fixpoint induction using a fixpoint abstraction of the fixpoint definition of the trace semantics \( \Theta^{\omega}[P] \).

17. Related work

Most directly relevant work has been cited in the text. For programs with unbounded executions, any finite homomorphic abstraction must introduce a loop so that finite model-checking [4] or bounded model-checking [2] are unapplicable (or unsound) to prove termination (or non-termination). Nevertheless, predicate abstraction [41] remains applicable since it is a finite encoding of an infinite abstract interpretation [16]. With predicate abstraction the end-user is left with the hard problem of providing candidate variant functions [14], as in [1]. Moreover [27] shows that infinitary abstractions
Abstract interpretation has established constructive principles for reasoning about semantics. A semantics is a fixpoint so proving a semantic property at some level of abstraction consists in verifying properties of abstract fixpoints which have to be checked (in checking/verification methods), guessed (in proof methods), or automatically inferred or approximated (in static analysis methods).

This principle was mainly applied in the past to invariance and indirectly to termination. We have shown that the abstract interpretation principle directly applies to both safety (generalizing invariance) and termination.

Moreover we have generalized the classical syntactic structural induction into the language-independent semantic concept of semantic induction based on (abstractions of) inductive trace covers which includes induction on syntax, control states, memory states, and execution trace segments and thus generalizes all verification and static analysis methods.

This methodology allowed us to establish new principles for proving termination by abstract interpretation of a termination semantics. It remains to design a suitable collection of abstract domains beyond the examples proposed in this paper and the corresponding implementations.

The present abstract interpretation termination framework has to be extended to liveliness [6, 53] and more generally to inevitability under fairness hypotheses [35, 52, 55].

18. Conclusion

Abstract interpretation has established constructive principles for reasoning about semantics. A semantics is a fixpoint so proving a semantic property at some level of abstraction consists in verifying properties of abstract fixpoints which have to be checked (in checking/verification methods), guessed (in proof methods), or automatically inferred or approximated (in static analysis methods).

This principle was mainly applied in the past to invariance and indirectly to termination by reduction to invariance. We have shown that the abstract interpretation principle directly applies to both safety (generalizing invariance) and termination.

Moreover we have generalized the classical syntactic structural induction into the language-independent semantic concept of semantic induction based on (abstractions of) inductive trace covers which includes induction on syntax, control states, memory states, and execution trace segments and thus generalizes all verification and static analysis methods.

This methodology allowed us to establish new principles for proving termination by abstract interpretation of a termination semantics. It remains to design a suitable collection of abstract domains beyond the examples proposed in this paper and the corresponding implementations.

The present abstract interpretation termination framework has to be extended to liveliness [6, 53] and more generally to inevitability under fairness hypotheses [35, 52, 55].

**References**