Notes

Corneliu Popeea

May 24, 2013

1 Propositional logic

Syntax We rely on a set of atomic propositions, AP, containing atoms like p, q. A propositional logic formula $\phi \in \mathit{Formula}$ is then defined by the following grammar given in BNF form:

$$\phi ::= p \mid \phi \wedge \phi \mid \neg\phi$$

We will use additional symbols in logical formulas, but they can be defined in terms of the conjunction and negation operators above.

$$\begin{aligned}
p \vee q &= \neg(\neg p \wedge \neg q) \\
p \rightarrow q &= \neg p \vee q = \neg(p \wedge \neg q) \\
p \leftrightarrow q &= (p \rightarrow q) \wedge (q \rightarrow p) \\
\bigwedge_{i \in \{1 \ldots n\}} p_i &= p_1 \wedge \cdots \wedge p_n \\
\bigvee_{i \in \{1 \ldots n\}} p_i &= p_1 \vee \cdots \vee p_n \\
\mathit{false} &= p \wedge \neg p \\
\mathit{true} &= \neg \mathit{false}
\end{aligned}$$

Semantics We assign a truth value to each propositional logic formula. The set of truth values contains two elements, T and F. A model M is a function that takes as argument an atomic proposition from AP and it returns a value from the set $\{T, F\}$. Then we define a satisfaction relation $M \models \phi$ (read "M satisfies ϕ") as follows.

- $M \models p$ iff $M(p) = T$
- $M \models \phi_1 \wedge \phi_2$ iff $M \models \phi_1$ and $M \models \phi_2$
- $M \models \neg\phi$ iff not $M \models \phi$

Note that logic, e.g., the satisfaction relation, is defined in terms of meta-logic, e.g., natural language words like "and", "not".

Decision problem Let us refer to a function *models* that given a propositional logic formula returns all its satisfying assignments. Then, we say a formula ϕ is *valid* if it is valid under every assignment. A formula ϕ is *satisfiable* if $\mathit{models}(\phi) \neq \emptyset$. Finally, a formula ϕ is *unsatisfiable* if $\mathit{models}(\phi) = \emptyset$.

The *decision problem for propositional logic*: given a propositional formula ϕ, is ϕ satisfiable? This is also called the boolean satisfiability problem or simply a SAT problem. A *decision procedure for propositional logic* is an algorithm that always terminates with a correct answer to the decision problem for propositional logic. Many SAT solvers are based on the Davis-Putnam-Loveland-Logemann (DPLL) framework. (Details of this family of decision procedures are beyond the scope of this report.)

2 Linear arithmetic

Syntax In the following, we will use V to denote a set of variables occurring in a formula, with $v \in V$. We use q to denote a rational number, $q \in \mathbb{Q}$. Then the grammar for linear arithmetic formulas consists of linear arithmetic terms and formulas:

$$\begin{aligned} t \quad &::= q * v \mid t + t \\ \phi_{LI} &::= t \leq q \mid t < q \mid \phi_{LI} \wedge \phi_{LI} \mid \neg \phi_{LI} \mid \exists v : \phi_{LI} \end{aligned}$$

Similar to the previous grammar for propositional logic formulas, the grammar for linear arithmetic formulas is kept concise without loss of expressivity. For example, the formula $3 \leq 5$ can be expressed using the grammar above as $0 * v \leq 2$, while $x = y$ can be represented as a conjunction of two inequalities, i.e., $x \leq y \wedge y \leq x$.

Semantics We again assign a truth value to each linear arithmetic formula. For this, we use a model M that is a function that takes as argument a variable from V and returns a value from $\mathbb{Q}$. We define a function to evaluate an arithmetic term, $eval : Term_{LI} \to \mathbb{Q}$.

- $eval(q * v, M) = q * M(v)$
- $eval(t_1 + t_2, M) = eval(t_1, M) + eval(t_2, M)$

We rely on a projection function $project(\phi_{LI}, v)$ to handle quantifier elimination, one possible realization being the Fourier-Motzkin elimination algorithm. As an example of the use of projection, consider $project(2 * v_1 - 3 * v2 \leq 5 \wedge v_2 - 6 * v3 \leq 0, v_2) = (2 * v_1 - 18 * v_3 \leq 5)$. Based on the evaluation and projection functions, we define a satisfaction relation for linear arithmetic formulas, $M \models_{LI} \phi_{LI}$:

- $M \models_{LI} t \leq q$ iff $eval(t, M) \leq q$
- $M \models_{LI} t < q$ iff $eval(t, M) < q$
- $M \models_{LI} \phi_1 \wedge \phi_2$ iff $M \models_{LI} \phi_1$ and $M \models_{LI} \phi_2$
- $M \models_{LI} \neg \phi$ iff not $M \models_{LI} \phi$
- $M \models_{LI} \exists v : \phi$ iff $M \models_{LI} project(\phi, v)$

Note that the symbol $\leq$ that occurs in the satisfaction relation is what we define ("the logic") in terms of the symbol $\leq$ from $eval(t, M) \leq q$ that comes from math ("the meta-logic").

Example 1. Let us consider the formula $\phi = (2 * v_1 - 3 * v_2 \leq 5)$ and the assignment $M(v_1) = 2$, $M(v_2) = 3$. We obtain that M satisfies ϕ, $M \models_{LI} \phi$, since $eval(2 * v_1 - 3 * v_2, M) = -5$ and $-5 \leq 5$.

3 Clauses

Syntax So far, we have used a number of functions and predicates with a definite meaning corresponding to linear arithmetic. We call the symbols "*" and "+" functions, and "$\leq$" and "$<$" predicates. Apart from these interpreted predicates, now we consider a set of uninterpreted predicate symbols (or query symbols) $\mathcal{Q}$. Predicates may be used either in infix of prefix form, e.g., $v_1 \leq v_2$ or $\leq (v_1, v_2)$.

Based on the predicate symbols $p \in \mathcal{Q}$, we define a language of clauses:

$$\begin{aligned} d \ &::= p(v, \ldots, v) \mid \phi_{LI} \mid d \vee d \mid \neg p(v, \ldots, v) \\ cl &::= \forall v, \ldots, v : d \end{aligned} \tag{1}$$

Note that the negation is used here in a more restrictive way than in the definition of linear arithmetic formulas ϕ_{LI}. In other words, a clause is a fancy name for a disjunction. One example of a clause follows, $\neg p(v_1) \vee v_1 \leq v_2 \vee q(v_2) \vee r(v_1)$, where p, q and r are uninterpreted predicate symbols.

We introduce an equivalent grammar that avoids the use of the negation operator.

$$\begin{aligned} \mathit{body} &::= p(v, \ldots, v) \mid \phi_{LI} \mid \mathit{body} \wedge \mathit{body} \\ \mathit{head} &::= p(v, \ldots, v) \mid \phi_{LI} \mid \mathit{head} \vee \mathit{head} \\ \mathit{cl} &::= \forall v, \ldots, v : \mathit{body} \rightarrow \mathit{head} \end{aligned} \tag{2}$$

For sake of brevity, we will abuse the notation and write $p(v)$ as a short-hand for $p(v_1, \ldots, v_n)$ even when the predicate p does not have arity 1. In such a case, v is used to denote the tuple of variables $(v_1, \ldots, v_n)$.

We define a variable substitution function on formulas from the background theory, e.g., LI,

$$(q_1 * v_1 + \cdots + q_n * v_n \leq q)[z/w] = q_1 * v'_1 + \cdots + q_n * v'_n \ ,$$

where $v'_i = z$ if $v_i = w$ and $v'_i = v_i$ if $v_i \neq w$.

Semantics We use a model M that is a function that takes uninterpreted predicates from P and returns a linear arithmetic formula ϕ_{LI}. We define a satisfaction relation for clauses, $M \models_{cl} cl$. The satisfaction relation makes use of the substitution function and the quantifier elimination procedure from the background theory.

- $M \models_{cl} \forall v, v_1, \ldots, v_m, w : \phi(v) \wedge p_1(v_1) \wedge \cdots \wedge p_n(v_n) \rightarrow \phi(w) \vee p_{n+1}(v_{n+1}) \vee \cdots \vee p_m(v_m)$ iff $\forall v, v_1, \ldots, v_m, w : \phi(v) \wedge \bigwedge_{i \in \{1,\ldots,n\}} M(p_i(a_i))[v_i/a_i] \rightarrow \phi(w) \vee \bigvee_{i \in \{n+1,\ldots,m\}} M(p_i(a_i))[v_i/a_i]$

Example 2. Let us consider the clause $cl = \forall i_1, i_2 : (i_1 = 0 \wedge h(i_1) \wedge i_2 = i_1 + 1 \rightarrow h(i_2))$ with one uninterpreted predicate symbol h of arity 1. The assignment $M(h(a)) = (a \geq 0)$ interprets this predicate symbol. We obtain that M satisfies cl since the following formula is true.

$$\forall i_1, i_2 : i_1 = 0 \wedge i_2 = i_1 + 1 \wedge i_1 \geq 0 \rightarrow i_2 \geq 0$$

4 Horn clauses

Syntax As an alternative to the previous two grammars for defining clauses, (1) and (2), we introduce a third definition that is more amenable to automated verification. The definition is based on a restricted form of clauses named Horn clauses. A Horn clause is a clause with a head that is either a formula from the background theory or a single uninterpreted predicate. The body and the clause definition are identical to those from the grammar (2). We obtain the following definition.

$$\begin{aligned} \mathit{body} &::= p(v, \ldots, v) \mid \phi_{LI} \mid \mathit{body} \wedge \mathit{body} \\ \mathit{head} &::= p(v, \ldots, v) \mid \phi_{LI} \\ \mathit{cl} &::= \forall v, \ldots, v : \mathit{body} \rightarrow \mathit{head} \\ \mathit{cls} &::= \mathit{cl} \wedge \mathit{cls} \mid \mathit{cl} \end{aligned} \tag{3}$$

Since we will deal with clauses with many terms, a monolithic formula *cls* is not ideal. We will use a set representation for clauses instead of presenting the conjunction of Horn clauses as a monolithic formula. This is illustrated by the next example.

Example 3. Consider the following monolithic formula represented in the language defined by the grammar (3).

$$\phi = \forall i_1, i_2, i_3, i_4 : (i_1 = 0 \rightarrow p(i_1)) \wedge (p(i_2) \wedge i_3 = i_2 + 1 \rightarrow p(i_3)) \wedge (p(i_4) \rightarrow q(i_4))$$

The formula ϕ refers to two uninterpreted predicate symbols of arity one p and q. An equivalent representation of ϕ as a set of Horn clauses follows.

$$\begin{aligned} HC_1 = \{\ & i = 0 \rightarrow p(i), \\ & p(i) \wedge i' = i + 1 \rightarrow p(i'), \\ & p(i) \rightarrow q(i)\} \end{aligned}$$

Furthermore, each element of the set HC_1 is implicitly universally quantified over all the free variables from the respective Horn clause.

Semantics The satisfaction relation for a Horn clause is the same as the relation $\models_{cl}$ for a general clause. A conjunction of Horn clauses is called a Horn formula and we use the same symbol $\models_{cl}$ for its satisfaction relation.

- $M \models_{cl} \mathit{cls}$ iff $M \models_{cl} \mathit{cl}$ for all clauses cl from the set cls

Observation Checking if the relation $\models_{cl}$ holds between a given M and a given HC is similar to making a proof by induction when the induction hypothesis is given. Consider the same clauses HC_1 where the uninterpreted predicate is renamed to hyp.

$$\{n = 0 \rightarrow \mathit{hyp}(n), \mathit{hyp}(n) \wedge n' = n + 1 \rightarrow \mathit{hyp}(n'), \mathit{hyp}(n) \rightarrow \phi_{\mathit{theorem}}(n)\}$$

Example 4. Consider an interpreted predicate $\mathit{edge}(x, x') = (x \geq 0 \wedge x' = x - 1)$ and an uninterpreted predicate $tc(x, x')$ that satisfy the following Horn clauses.

$$\begin{aligned} HC_2 = \{\ & \mathit{edge}(v, w) \rightarrow tc(v, w), \\ & tc(v, w) \wedge \mathit{edge}(w, z) \rightarrow tc(v, z)\} \end{aligned}$$

The assignment $M(tc(v, w)) = (v \geq 0 \wedge w < v)$ satisfies the clauses given above: $M \models_{cl} HC_2$. In fact, the interpretation of the tc is the transitive closure of the edge relation.

5 Inference algorithms for Horn clauses

The goal of this section is to present algorithms that, given a set of Horn clauses HC, infer an assignment M such that $M \models_{cl} HC$.

We recall the definition of a set of Horn clauses:

$$\begin{aligned} \mathit{body} &::= p(v, \ldots, v) \mid \phi_{LI} \mid \mathit{body} \wedge \mathit{body} \\ \mathit{head} &::= p(v, \ldots, v) \mid \phi_{LI} \\ \mathit{cl} &::= \mathit{body} \rightarrow \mathit{head} \\ HC &::= \{\mathit{cl}, \ldots, \mathit{cl}\} \end{aligned}$$

We distinguish three classes of Horn clauses:

- Inference clauses are clauses with a head that is a single uninterpreted predicate.

$$\phi_0(v_0) \wedge \bigwedge_{i \in \{0 \ldots n\}} p_i(v_i) \rightarrow p(v)$$

- Property clauses are clauses with a head that is a formula from the background theory.

$$\phi_0(v_0) \wedge \bigwedge_{i \in \{1 \ldots n\}} p_i(v_i) \rightarrow \phi(v)$$

- Initial clauses are inference clauses without uninterpreted predicates in their body.

$$\phi_0(v_0) \rightarrow p(v)$$

For clarity of presentation, the following algorithms assume that HC contains only inference clauses (no property clauses).

Two views of a Horn clause Note the following logical equivalence between the "universal view of a Horn clause",

$$\forall a, b : p(a, b) \rightarrow q(a)$$

and the "existential view of a Horn clause",

$$\forall a : (\exists b : p(a, b)) \rightarrow q(a)$$

The above equivalence suggests that given some interpretation for the predicate $p(a, b)$, the interpretation for $q(a)$ should contain at least $\exists b : p(a, b)$.

Bounded inference, symbolic reasoning The first algorithm that we present unfolds the Horn clauses from HC only up to a constant number of times k. The algorithm consists of the following three steps.

s1) Start with the *false* formula for each uninterpreted predicate. For a predicate $p_i(v_i)$, we denote its zero-th candidate as $p_i^0(v_i)$.

s2) Based on the j-th candidates for the predicates that appear in the body of a clause cl,

$$cl = \phi_0(v_0) \wedge \bigwedge_{i \in \{1 \ldots n\}} p_i(v_i) \rightarrow p(v) \ ,$$

compute the (j+1)-th candidate for the predicate from the head of the clause as $p^{j+1}(v) = p^j(v) \vee \bigvee_{cl \in HC} p^{\Delta}(v)$, where

$$p^{\Delta}(v) = \exists((v_0 \cup \ldots \cup v_n) \backslash v : \phi_0(v_0) \wedge \bigwedge p_i^j(v_i))$$

s3) Stop when we reach the k-th candidates for the uninterpreted predicates. The last candidates for the queries, i.e., the k-th candidates, represent the result of the bounded inference algorithm.

Example 5.

$$\begin{aligned} HC_3 = \{ \ & x = 0 \wedge y = n \rightarrow p(x, y, n), \\ & p(x, y, n) \wedge x' = x + 1 \wedge y' = y - 1 \wedge y \geq 0 \wedge n' = n \rightarrow p(x', y', n')\} \end{aligned}$$

We compute the bounded inference solution for $k = 2$ as follows.

- We start with $p_0(x, y, n) = \mathit{false}$.
- We compute the first candidate as a disjunction of the two formulas: $p^{11}(x, y, n) = \exists \emptyset : x = 0 \wedge y = n$ and $p^{12}(x, y, n) = \mathit{false}$. We obtain $p^1(x, y, n) = (x = 0 \wedge y = n)$.
- We compute the second candidate: $p^2(x', y', n') = \exists x, y, n : x = 0 \wedge y = n \wedge x' = x + 1 \wedge y' = y - 1 \wedge y \geq 0 \wedge n' = n$. We obtain $p^2(x, y, n) = (x = 1 \wedge n \geq 0 \wedge y = n - 1)$ and stop here, since we have reached the candidate $k = 2$.

Unbounded inference, symbolic reasoning This algorithms uses the steps s1) and s2) from the previous bounded case and a different stopping condition.

s3') Stop the inference for $p_i(v_i)$, when we reach the j-th candidate such that $p_i^{j+1}(v_i) = p_i^j(v_i)$. If the algorithm terminates, then the solution for $p_i(v_i)$ is the last computed candidate, i.e., $p_i^{j+1}(v_i)$.

This algorithm is not guaranteed to terminate.

Theorem 1 (Most precise solution)**.** Disjunction over all $p^k(v)$ gives the most precise solution or the least solution for $p(v)$ when considering all unfoldings (of arbitrary depth).

Exercise 1. Let us consider the following sets of Horn clauses.

$$HC_4 = \{\ x \geq 0 \rightarrow p(x),\\ p(x) \wedge x' = x + 1 \rightarrow p(x')\}$$

$$HC_5 = \{\ x = 0 \rightarrow p(x),\\ p(x) \wedge x' = x + 1 \rightarrow p(x')\}$$

Does the unbounded inference algorithm terminate for either HC_4 or HC_5?

Abstract inference, symbolic reasoning The distinguishing feature of this algorithm is that it extrapolates a candidate using an abstraction function. An abstraction function should have two properties:

- $\forall \phi(v) : \forall v : \phi(v) \rightarrow \alpha(\phi(v))$
- $\forall \phi(v) : \forall \psi(v) : (\phi(v) \rightarrow \psi(v)) \rightarrow \alpha(\phi(v)) \rightarrow \alpha(\psi(v))$ (monotonicity condition)

We use an abstraction function defined using a set of interpreted predicates:

$$\alpha_P(\phi(v)) = \bigwedge \{p(v) \mid p(v) \in P \text{ and } \forall v : \phi(v) \rightarrow p(v)\}$$

Example 6. For the set of predicates $P = \{s = 0, i = n\}$ and the formula $\phi(pc, s, i) = (pc = 1 \wedge s = 0 \wedge i = n)$, we obtain

$$\alpha_P(\phi(pc, s, i)) = \alpha_P(pc = 1 \wedge s = 0 \wedge i = n) = \bigwedge \{s = 0, i = n\} = (s = 0 \wedge i = n)$$

This abstract inference algorithm is some times called predicate abstraction, since the abstraction function is defined using predicates. The abstract inference algorithm consists of the steps s1) and s3') from the previous algorithm and a different second step.

s2') Based on the j-th candidates for the predicates that appear in the body of a clause cl,

$$cl = \phi_0(v_0) \wedge \bigwedge_{i \in \{1 \ldots n\}} p_i(v_i) \rightarrow p(v)\ ,$$

compute the (j+1)-th candidate for the predicate from the head of the clause as $p^{j+1}(v) = p^j(v) \vee \bigvee_{cl \in HC} p^{\Delta}(v)$, where

$$p^{\Delta}(v) = \alpha_P(\exists((v_0 \cup \ldots \cup v_n) \backslash v : \phi_0(v_0) \wedge \bigwedge p_i^j(v_i)))$$

Given HC, the solution for $p_i(v_i)$ is the last computed candidate. The abstract inference algorithm is guaranteed to terminate, but may not compute the most precise solution for the given set of Horn clauses.

Exercise 2. Apply the abstract inference algorithm for $P = \{x \geq 0, x \geq 1\}$ and the sets of Horn clauses given in the previous exercise, HC_4 and HC_5.

Table 1: Proof of abstraction property for predicate abstraction function

Show:	$\forall \phi : \forall v : \phi(v) \rightarrow \alpha_P(\phi(v))$.	
1.	$\phi^*(v^*)$	hypothesis; arbitrary ϕ^* and v^*
2.	$\forall v : \phi^*(v) \rightarrow p_i(v)$	from definition of $\alpha_P(\phi^*(v^*)) = \bigwedge_{i \in \{1..k\}} p_i(v^*)$
3.	$\phi^*(v^*) \rightarrow p_i(v^*)$	from 2, choose $v = v^*$; arbitrary $i \in \{1..k\}$
4.	$p_i(v^*)$	from 1,3; arbitrary $i \in \{1..k\}$
5.	$p_1(v^*) \wedge \cdots \wedge p_k(v^*)$	from 4
6.	$\alpha_P(\phi^*(v^*))$	from 2,5
7.	$\forall \phi : \forall v : \phi(v) \rightarrow \alpha_P(\phi(v))$	from 1,6 that hold for arbitrary ϕ^* and v^*

Table 2: Proof of monotonicity property for predicate abstraction function

Show:	$\forall \phi : \forall \psi : (\forall v : \phi(v) \rightarrow \psi(v)) \rightarrow (\forall v : \alpha_p(\phi(v)) \rightarrow \alpha_p(\psi(v)))$.	
1.	$\forall v : \phi^*(v) \rightarrow \psi^*(v)$	hypothesis; arbitrary ϕ^* and ψ^*
2.	$\forall v : \phi^*(v) \rightarrow p_i(v)$	from definition of $\alpha_P(\phi^*(v^*)) = \bigwedge_{p_i \in I} p_i(v^*)$
3.	$\forall v : \psi^*(v) \rightarrow p_j(v)$	from definition of $\alpha_P(\psi^*(v^*)) = \bigwedge_{p_j \in J} p_j(v^*)$
4.	$J \subseteq I$	proof by contradiction (Lemma 1 uses 1,2,3)
5.	$\forall v : \bigwedge_{p_i \in I} p_i(v) \rightarrow \bigwedge_{p_j \in J} p_j(v)$	from 4
6.	$\forall v : \alpha_P(\phi^*(v)) \rightarrow \alpha_P(\psi^*(v))$	from 2,3,5
7.	$\forall \phi : \forall \psi : (\forall v : \phi(v) \rightarrow \psi(v)) \rightarrow (\forall v : \alpha_p(\phi(v)) \rightarrow \alpha_p(\psi(v)))$	from 1,6 that hold for arbitrary ϕ^* and v^*

Checking property clauses Once the inference algorithms are terminated (either unbounded inference or abstract inference), we can check if the property clauses are satisfied as follows. For each property clause,

$$cl = \phi_0(v_0) \wedge \bigwedge p_i(v_i) \rightarrow \phi(v) \ ,$$

we use the solutions computed so far for $p_i(v_i)$ and substitute them in cl. If all property clauses are valid, then the solutions satisfy all the clauses from HC and we return the solution for HC.

Exercise 3. Prove that the predicate abstraction function has the two properties required to be an abstraction functions.

The proofs are given in Table 1 and Table 2. The second proof makes use of the sub-proof done by contradiction.

Lemma 1. *If $\forall v : \phi(v) \rightarrow \psi(v)$, and I the set of predicates corresponding to $\alpha_P(\phi(v))$ and J the set of predicates corresponding to $\alpha_P(\psi(v))$. Then $J \subseteq I$.*

Proof. Assume that there exists a predicate $p \in J \backslash I$. Then we have three facts: $\forall v : \phi(v) \rightarrow \psi(v)$ and $\forall v : \phi(v) \rightarrow p(v)$ (from $p \in J$) and $\neg(\forall v : \psi(v) \rightarrow p(v))$ (from $p \notin I$). In these three facts we have a contradiction. Therefore our assumption does not hold and $J \backslash I$ is an empty set. □

6 Programs

In this section, we will define programs as transition systems, then characterize some interesting program correctness properties.

Syntax A program $Prog = (V, pc, \phi_{init}, \mathcal{R}, \phi_{err})$ consists of

- V - a tuple of program variables.
- pc - a special program variable representing the program counter.
- ϕ_{init} - an initiation condition given by a formula over V.
- $\mathcal{R}$ - a set of "single-statement" transition relations, each of them given by a formula over V and V'.
- ϕ_{err} - an error condition given by a formula over V.

The previous representation for a program is called a transition system and it is often used as a model to describe the behavior of software or systems. Let $\rho_{\mathcal{R}}$ denote the program transition relation, i.e., the union of "single-statement" transition relations. We have $\rho_{\mathcal{R}} = \bigvee_{\rho_i \in \mathcal{R}} \rho_i$.

Example 7. We show the code for an example program and its transition system representation.

```
   main(int x, int y, int z) {
1:   assume(y>=z);
2:   while (x<y) {
3:     x++;
4:   }
5:   assert(x>=z);
6: }
```

For our example, $V = (x, y, z, pc)$, $\phi_{init}(V) = (pc = 1)$, $\mathcal{R} = \{\rho_1, \rho_2, \rho_3, \rho_4, \rho_5\}$, $\phi_{err}(V) = (pc = -1)$ with the following transition relations:

$$\rho_1(V, V') = move(1, 2) \wedge y \geq z \wedge skip(x, y, z)$$
$$\rho_2(V, V') = move(2, 2) \wedge x < y \wedge x' = x + 1 \wedge skip(y, z)$$
$$\rho_3(V, V') = move(2, 5) \wedge x \geq y \wedge skip(x, y, z)$$
$$\rho_4(V, V') = move(5, 6) \wedge x \geq z \wedge skip(x, y, z)$$
$$\rho_5(V, V') = move(5, -1) \wedge x < z \wedge skip(x, y, z)$$

In the previous relations, we used *move* and *skip* as abbreviations. They are defined as: $move(i, j) = (pc = i \wedge pc = j)$ and $skip(x, y, z) = (x' = x \wedge y' = y \wedge z' = z)$.

Semantics Each program variable is assigned a domain of values. A program state is a function that assigns to each program variable a value from its domain. Therefore, we can reuse the satisfaction relation symbol defined previously for an assignment M and a linear arithmetic formula in the context of "state s satisfies formula ϕ". A formula with free variables from V and V' represents a binary relation between pairs of states, where the first component of each pair assigns values to V and the second component assigns values to V'. In the following, we could restrict our attention to program variables assigned rational values and use $\models_{LI}$ for the satisfaction relation between (pairs of) states and formulas. For sake of generality, we assume a satisfaction relation defined over a first-order theory and we overload the symbol $\models$ to denote such a relation.

A computation is a sequence of program states $s_1, s_2, \ldots$ such that 1) s_1 is an initial state, i.e., $s_1 \models \phi_{init}$ and 2) each pair of consecutive states s_i and s_{i+1} from the sequence is connected by a transition relation, i.e., $(s_i, s_{i+1}) \models \rho$ with $\rho \in \mathcal{R}$. A state is reachable if it occurs in a program computation. Let ϕ_{reach} denote the set of reachable program states. A computation segment is a sequence of program states $s_i, \ldots$ such that s_i is a reachable state and each pair of consecutive states s_j and s_{j+1} in the sequence is connected by a transition relation, i.e., $(s_j, s_{j+1}) \models \rho$ with $\rho \in \mathcal{R}$. A computation is finite if some state element s_j from the sequence does not have any successors, i.e., there does not exist a state s such that $(s_j, s) \models \rho$ for any $\rho \in \mathcal{R}$.

Example 8. A computation of the program from Example 7 is the following sequence of program states, where we use a tuple of values to represent a state.

$$(1,3,2,1),(1,3,2,2),(2,3,2,2),(3,3,2,2),(3,3,2,3),(3,3,2,5),(3,3,2,6)$$

The tuple (1,3,2,1) corresponds to the program state where x, y, z and pc are assigned the values 1, 3, 2 and respectively 1.

Correctness properties We consider two properties of programs, safety and termination.

Definition 1. A program is safe if no error state is reachable.

To prove program safety, it suffices to find a symbolic description for the reachable states $R(V)$ such that the following three conditions hold.

$$\begin{aligned}
&\forall s : (s \models \phi_{init}(V)) \rightarrow (s \models R(V)) \\
&\forall s, s' : (s \models R(V)) \wedge ((s, s') \models \rho_i(V, V')) \rightarrow (s' \models R(V')) \quad \text{for } \rho_i \in \mathcal{R} \\
&\forall s : (s \models R(V)) \rightarrow (s \not\models \phi_{err}(V)
\end{aligned}$$

Given that a program state is a function defined over program variables, we obtain equivalent conditions for program safety universally quantified over program variables.

Theorem 2 (Soundness of safety checking)**.** *A program is safe if there exists a solution for the query $R(V)$ that satisfies the following conditions.*

$$\begin{aligned}
&\forall V : \phi_{init}(V) \rightarrow R(V) \\
&\forall V, V' : R(V) \wedge \rho_{\mathcal{R}}(V, V') \rightarrow R(V') \\
&\forall V : R(V) \wedge \phi_{err}(V) \rightarrow \mathit{false}
\end{aligned} \tag{4}$$

Proof. Omitted. □

Theorem 3 (Completeness of safety checking)**.** *If a program is safe then there exists a solution for the query $R(V)$ that satisfies the following conditions.*

$$\begin{aligned}
&\forall V : \phi_{init}(V) \rightarrow R(V) \\
&\forall V, V' : R(V) \wedge \rho_{\mathcal{R}}(V, V') \rightarrow R(V') \\
&\forall V : R(V) \wedge \phi_{err}(V) \rightarrow \mathit{false}
\end{aligned}$$

Proof. Omitted. □

A solution that satisfies the conditions from the safety proof rule is called an inductive state invariant of the program P.

Example 9. For the program given in Example 7, we apply an inference algorithm to compute a solution for $R(V)$ that satisfies the first two clauses from the safety proof rule (4). For this example, we use the unbounded inference algorithm and obtain the following four candidate formulas.

$$\begin{aligned}
&R^1(V) = (pc = 1) \\
&R^2(V) = (pc = 1) \vee (pc = 2 \wedge y \geq z) \\
&R^3(V) = (pc = 1) \vee (pc = 2 \wedge y \geq z) \vee (pc = 5 \wedge y \geq z \wedge x \geq y) \\
&R^4(V) = (pc = 1) \vee (pc = 2 \wedge y \geq z) \vee (pc = 5 \wedge y \geq z \wedge x \geq y) \vee (pc = 6 \wedge y \geq z \wedge x \geq y)
\end{aligned}$$

At the next step, the algorithm computes a formula $R^5(V)$ that is equivalent to $R^4(V)$ and the inference process stops here. We next check the property clause, the third clause from the proof rule (4). This clause is indeed satisfied, since $R^4(V) \wedge \phi_{err}(V)$ is unsatisfiable. From the existence of a solution for the query $R(V)$, we conclude that the given program is safe.

Definition 2. A program is terminating if all its computations are finite.

In the following, we use the notion of a well-founded relation, a relation that does not admit infinite chains. The termination condition is equivalent to the fact that the transition relation of P restricted to reachable states is well-founded. Here we give two proof rules that list conditions necessary and sufficient for program termination.

The first termination proof rule relies on a ranking function, a function that maps program states to an well-founded set, $(W, \prec)$. An example of a well-founded set is $(\mathbb{N}, <)$, while $(\mathbb{N}, \geq)$ is not a well-founded set. In other words, for termination we need to find a function that takes as argument a program state and returns an expression over the program variables such that the expression is provably decreasing throughout the states of a computation.

$$\begin{aligned} &\forall V : \phi_{init}(V) \rightarrow R(V) \\ &\forall V, V' : R(V) \wedge \rho_{\mathcal{R}}(V, V') \rightarrow R(V') \\ &\forall V, V' : R(V) \wedge \rho_{\mathcal{R}}(V, V') \rightarrow r(V) \succ r(V') \end{aligned} \tag{5}$$

Example 10. For the transition system given by $V = (x, y)$, $\phi_{init}(V) = \mathit{true}$ and $\rho_{\mathcal{R}}(V, V') = (x < y \wedge x' = x+1 \wedge y' = y)$, there exist a well-founded set $(\mathbb{N}, <)$, a ranking function $r(V) = (y-x)$ and a predicate $R(V) = \mathit{true}$ that satisfy the conditions from the proof rule (5).

In some cases, constructing the termination argument, i.e., the ranking function, for a complex transition relation is difficult. A second proof rule alleviates this problem by relying on a termination argument that is composed from smaller termination sub-arguments. A second proof rule guarantees program termination provided there exist a query $R(V)$ (that stands for reachable states) satisfying the first two conditions from equation (4) and a query $T(V, V')$ (that stands for reachable computation segments) that satisfies additional conditions.

$$\begin{aligned} &\forall V : \phi_{init}(V) \rightarrow R(V) \\ &\forall V, V' : R(V) \wedge \rho_{\mathcal{R}}(V, V') \rightarrow R(V') \\ &\forall V, V' : R(V) \wedge \rho_{\mathcal{R}}(V, V') \rightarrow T(V, V') \\ &\forall V, V', V'' : T(V, V') \wedge \rho_{\mathcal{R}}(V', V'') \rightarrow T(V, V'') \\ &\forall V, V' : T(V, V') \rightarrow \mathit{WF}_1(V, V') \vee \cdots \vee \mathit{WF}_n(V, V') \end{aligned} \tag{6}$$

Example 11. For the program given in Example 10, there exist $R(V) = \mathit{true}$, $T(V, V') = (x < y \wedge x' \geq x + 1 \wedge y' = y)$ and $\mathit{WF}(V, V') = y - x > 0 \wedge y' - x' < y - x$ that satisfy the conditions from the proof rule (6).

7 Programs with procedures

In this section, we describe procedural programs as transition systems then reason about their safety properties.

Syntax A procedural program is identified by a tuple $\mathit{Prog} = (P, V_G, V_p, \mathit{init}, \mathit{step}_p, \mathit{call}_{p,q}, \mathit{ret}_p, \mathit{local}_p, \mathit{error}_p)$ with the following tuple elements.

- P - a set of procedures that includes an initial procedure *main*.
- V_G - a tuple of global variables that includes a return variable *ret*.
- V_p - a tuple of local variables for each procedure $p \in P$ that includes a program counter variable pc_p.
- $\mathit{init}(V_G, V_{main})$ - an initialization condition for the program.

- $step_p(V_G, V_p, V'_G, V'_p)$ - a transition corresponding to intra-procedural steps for each procedure $p \in P$.
- $call_{p,q}(V_G, V_p, V_q)$ - a parameter passing transition for each pair of caller $p \in P$ and callee $q \in P$.
- $ret_p(V_G, V_p, V'_G)$ - a return value passing transition for each procedure $p \in P$.
- $local_p(V_p, V'_p)$ - a transition to capture the program counter value change across a call site of the procedure $p \in P$.
- $error_p(V_G, V_p)$ - an error condition for each procedure $p \in P$.

Example 12. We show the code for an example program and its transition system representation. The function f computes the sum of the natural numbers less or equal than i. The function *main* invokes f and then checks that the result of f satisfies a simple property. We use a global variable *ret* to save the return value of a function.

```
int ret;
f(int i) {
  if (i>0) {
    f(i-1);
    ret = ret+i;
  } else {
    ret = 0;
  }
}

void main() {
  int n;
  f(n);
  assert(ret>=0);
  ret = ret;
}
```

For our example, we have $P = \{f, main\}$, $V_G = \{ret\}$, $V_f = \{i, pc_f\}$, $V_{main} = \{n, pc_{main}\}$, $init(V_G, V_{main}) = (pc_{main} = l_1)$. In addition, we have the following representation for the procedure f.

$$
\begin{aligned}
step_f(V_G, V_f, V'_G, V'_f) &= (pc_f = m_1 \wedge pc'_f = m_2 \wedge i > 0 \wedge skip(ret, i)) \vee \\
&\quad (pc_f = m_1 \wedge pc'_f = m_3 \wedge i \leq 0 \wedge skip(ret, i)) \\
call_{f,f}(V_G, V_f, V'_f) &= (pc_f = m_2 \wedge pc'_f = m_1 \wedge i' = i - 1) \\
ret_f(V_G, V_f, V'_G) &= (pc_f = m_3 \wedge ret' = 0 \vee pc_f = m_4 \wedge ret' = ret + i) \\
local_f(V_f, V'_f) &= (pc_f = m_2 \wedge pc'_f = m_4 \wedge i' = i) \\
error_f(V_G, V_f) &= false
\end{aligned}
$$

For the procedure *main*, we have the following transitions:

$$
\begin{aligned}
step_{main}(V_G, V_{main}, V'_G, V'_{main}) &= (pc_{main} = l_2 \wedge pc'_{main} = l_3 \wedge ret \geq 0 \wedge skip(ret, n)) \\
call_{main,f}(V_G, V_{main}, V_f) &= (pc_{main} = l_1 \wedge pc_f = m_1 \wedge i = n) \\
ret_{main}(V_G, V_{main}, V'_G) &= (pc_{main} = l_3 \wedge ret' = ret) \\
local_{main}(V_{main}, V'_{main}) &= (pc_{main} = l_1 \wedge pc'_{main} = l_2) \\
error_{main}(V_G, V_{main}) &= (pc_{main} = l_2 \wedge ret < 0)
\end{aligned}
$$

Semantics As before a program variable is assigned a domain of values. A state of a procedural program consists of values for program variables in scope plus a stack of calls whose execution is not yet finished. Every frame of the stack contains values for the local variables of the pending calls. We denote the empty stack by ϵ, while $(l_p \cdot st)$ represents the stack st with an additional frame added on top corresponding to procedure p.

A computation is a sequence of states $s_1, s_2, \ldots$ such that s_1 is an initial state as follows.

$$\frac{(g, l_{main}) \models init(V_G, V_{main})}{((g, l_{main}), \epsilon) \in Reach_{main}}$$

Pairs of states s_i and s_{i+1} are part of a computation, and therefore elements of the reachable states set $Reach_p$ if one of three conditions hold. The three conditions correspond to an intra-procedural step via *step* or a procedure call via *call*, or to a return from a procedure call via *ret*.

$$\frac{((g, l_p), st) \in Reach_p \quad ((g, l_p), (g', l'_p)) \models step_p(V_G, V_p, V'_G, V'_p)}{((g', l'_p), st) \in Reach_p}$$

$$\frac{((g, l_p), st) \in Reach_p \quad (g, l_p, l_q) \models call_{p,q}(V_G, V_p, V_q)}{((g, l_q), l_p \cdot st) \in Reach_q}$$

$$\frac{((g, l_q), l_p \cdot st) \in Reach_q \quad (g, l_q, g') \models ret_q(V_G, V_q, V'_G) \quad (l_p, l'_p) \models local_p(V_p, V'_p)}{((g', l'_p), st) \in Reach_p}$$

Example 13. For the program considered in Example 12, given an initial state $s_1 = ((ret \mapsto 0, n \mapsto 1, pc_{main} \mapsto l_1), \epsilon)$, below are the states from a possible computation starting with s_1.

$$\begin{aligned}
s_2 &= ((ret \mapsto 0, i \mapsto 1, pc_f \mapsto m_1), (n \mapsto 1, pc_{main} \mapsto l_1) \cdot \epsilon) \\
s_3 &= ((ret \mapsto 0, i \mapsto 1, pc_f \mapsto m_2), (n \mapsto 1, pc_{main} \mapsto l_1) \cdot \epsilon) \\
s_4 &= ((ret \mapsto 0, i \mapsto 0, pc_f \mapsto m_1), (i \mapsto 1, pc_f \mapsto m_2) \cdot (n \mapsto 1, pc_{main} \mapsto l_1) \cdot \epsilon) \\
s_5 &= ((ret \mapsto 0, i \mapsto 0, pc_f \mapsto m_3), (i \mapsto 1, pc_f \mapsto m_2) \cdot (n \mapsto 1, pc_{main} \mapsto l_1) \cdot \epsilon) \\
s_6 &= ((ret \mapsto 0, i \mapsto 1, pc_f \mapsto m_4), (n \mapsto 1, pc_{main} \mapsto l_1) \cdot \epsilon) \\
s_7 &= ((ret \mapsto 1, n \mapsto 1, pc_{main} \mapsto l_2), \epsilon) \\
s_8 &= ((ret \mapsto 1, n \mapsto 1, pc_{main} \mapsto l_3), \epsilon)
\end{aligned}$$

Summarization proof rule To prove program safety, it suffices to find symbolic formulas $Summ_p$ for each procedure p such that the following five conditions hold.

$$\begin{aligned}
&C1\text{:} \quad init(V_G, V_{main}) \wedge skip(V_G, V_{main}) \rightarrow Summ_{main}(V_G, V_{main}, V'_G, V'_{main}) \\
&C2\text{:} \quad Summ_p(V_G, V_p, V'_G, V'_p) \wedge step_p(V'_G, V'_p, V''_G, V''_p) \rightarrow Summ_p(V_G, V_p, V''_G, V''_p) \quad \text{foreach } p \\
&C3\text{:} \quad Summ_p(V_G, V_p, V'_G, V'_p) \wedge call_{p,q}(V'_G, V'_p, V_q) \rightarrow Summ_q(V'_G, V_q, V'_G, V_q) \quad \text{foreach } p, q \\
&C4\text{:} \quad Summ_p(V_G, V_p, V'_G, V'_p) \wedge call_{p,q}(V'_G, V'_p, V_q) \wedge Summ_q(V'_G, V_q, V''_G, V'_q) \wedge \quad \text{foreach } p, q \\
&\qquad ret_q(V''_G, V'_q, V'''_G) \wedge local_p(V'_p, V''_p) \rightarrow Summ_p(V_G, V_p, V'''_G, V''_p) \\
&C5\text{:} \quad Summ_p(V_G, V_p, V'_G, V'_p) \wedge error_p(V'_G, V'_p) \rightarrow false \quad \text{foreach } p
\end{aligned}$$

Example 14. For the program from Example 12, we run the abstract inference algorithm with a predicate abstraction function defined using predicates over the program counter variables pc_{main} and pc_f. We obtain:

$$
\begin{aligned}
Summ_{main}(V_G, V_{main}, V'_G, V'_{main}) = \; & (pc_{main} = l_1 \wedge pc'_{main} = l_1) \vee \\
& (pc_{main} = l_1 \wedge pc'_{main} = l_2) \vee \\
& (pc_{main} = l_1 \wedge pc'_{main} = l_3) \\
Summ_f(V_G, V_f, V'_G, V'_f) = \; & (pc_f = m_1 \wedge pc'_f = m_1) \vee \\
& (pc_f = m_1 \wedge pc'_f = m_2) \vee \\
& (pc_f = m_1 \wedge pc'_f = m_3) \vee \\
& (pc_f = m_1 \wedge pc'_f = m_4)
\end{aligned}
$$

We observe that the formula $Summ_{main}(V_G, V_{main}, V'_G, V'_{main}) \wedge error_{main}(V'_G, V'_{main})$ is satisfiable, so we cannot conclude safety of the program.

Running the abstract inference algorithm with three additional predicates $\{i' \geq 0, i' + ret' \geq 0, ret' \geq 0\}$, we obtain the following. (The additional formulas are written in blue.)

$$
\begin{aligned}
Summ_{main}(V_G, V_{main}, V'_G, V'_{main}) = \; & (pc_{main} = l_1 \wedge pc'_{main} = l_1) \vee \\
& (pc_{main} = l_1 \wedge pc'_{main} = l_2 \wedge ret' \geq 0) \vee \\
& (pc_{main} = l_1 \wedge pc'_{main} = l_3 \wedge ret' \geq 0) \\
Summ_f(V_G, V_f, V'_G, V'_f) = \; & (pc_f = m_1 \wedge pc'_f = m_1) \vee \\
& (pc_f = m_1 \wedge pc'_f = m_2 \wedge i' \geq 0) \vee \\
& (pc_f = m_1 \wedge pc'_f = m_3) \vee \\
& (pc_f = m_1 \wedge pc'_f = m_4 \wedge i' + ret' \geq 0)
\end{aligned}
$$

This time, the clause $C5$ is satisfied, so we can conclude that the given program is indeed safe.