An iterative approach to precondition inference using constrained Horn clauses

Bishoksan Kafle
The University of Melbourne

John P. Gallagher
Roskilde University and IMDEA Software Institute

Graeme Gange, Peter Schachte, Harald Søndergaard, Peter J. Stuckey
The University of Melbourne

submitted 1 January 2003; revised 1 January 2003; accepted 1 January 2003

Abstract

We present a method for automatic inference of conditions on the initial states of a program that guarantee that the safety assertions in the program are not violated. Constrained Horn clauses (CHCs) are used to model the program and assertions in a uniform way, and we use standard abstract interpretations to derive an over-approximation of the set of *unsafe* initial states. The precondition then is the constraint satisfied by the complement of that set under-approximating the set of *safe* initial states. This idea is not new, but previous attempts to exploit it have suffered from loss of precision. Here we develop an iterative refinement algorithm for non-linear CHCs and show that it produces much more precise, and in some cases optimal, approximations of the safety conditions, and can scale to larger programs. The refinement algorithm uses partial evaluation and a form of counterexample-guided abstraction refinement to focus on the relevant program states. Disjunctive constraints, which are essential to achieve good precision, are generated in a controlled way by program transformations that perform polyvariant specialisation. The algorithm is implemented and tested on a benchmark suite of programs from the literature in precondition inference and software verification competitions.

KEYWORDS: Precondition inference, backwards analysis, abstract interpretation, refinement, program specialisation, program transformation.

1 Introduction

Given a program with properties required to hold at specific program points, *precondition analysis* derives the conditions on the initial states ensuring that the properties hold. This has important applications in program verification, symbolic execution, program understanding and debugging. While forward abstract interpretation approximates the set of reachable states of a program, *backward* abstract interpretations approximates the set of states that can reach some target state. Both forward and backward analyses may produce over- or under-approximations, and forward and backward analysis may profitably be combined (Cousot and Cousot 1992; Cousot et al. 2011; Bakhirkin and Monniaux 2017).

Most approaches that apply backward analysis, possibly in conjunction with forward analysis, use over-approximations, and as a result derive *necessary* pre-conditions. Less attention has been given to under-approximating backwards analyses, with the goal of finding *sufficient* pre-conditions. However, it is natural to try to derive guarantees of safe behaviour of a program. Often we would like to know which initial states *must* be safe, in the sense that no computation starting from such a state can possibly reach a specified error state, that is, we desire to find (non-trivial) sufficient conditions for safety.

If analysis uses an abstract domain which is *complemented*, duality enables sufficient conditions to be derived from necessary conditions and *vice versa*. However, complemented abstract domains are very rare, and approximation of a complement tends to introduce considerable lack of precision. The under-approximating backward abstract interpretation of Howe et al. (2004) utilises the fact that the abstract domain *Pos* is pseudo-complemented (Marriott and Søndergaard 1993), but pseudo-complementation too is very rare. Moy (2008) presents a method for deriving sufficient preconditions (for use with a theorem prover), employing weakest-precondition reasoning and forward abstract interpretation to attempt to generalise conditions at loop heads. Bakhirkin et al. (2014) observe that there may be an advantage in generalising an abstract complement operation to (abstract) logical subtraction, as this may improve opportunities to find a tighter approximation of a set of states.

Miné (2012a) infers sufficient conditions for safety, not by instantiating a generic mechanism for complementation, but by designing all required purpose-built backward transfer functions. He does this for three numeric abstract domains: intervals, octagons and convex polyhedra—a substantial effort, as the purpose-built operations, including widening, can be rather intricate.

We share Miné's goal but use program transformation and over-approximating abstract interpretation over a Horn clause program representation. This allows us to apply a range of established tools and techniques beyond abstract interpretation, including query-answer transformation, partial evaluation and abstraction refinement. We offer an iterative approach that successively specialises a program and also incorporates counterexample-based refinement (CEGAR) (Clarke et al. 2003). The approach of iteratively specialising a program represented as Horn clauses has also been pursued by De Angelis et al. (2014) in order to verify program properties. Their techniques also incorporate forward and backward propagation of constraints, but rather than explicitly using abstract interpretation, their specialisation algorithm involves a special constraint generalisation method.

We shall use the example in Figure 1 to demonstrate our approach. The left side shows a C program fragment, and the right its constrained Horn clause (CHC) representation. Given an imperative program (with assertions), its translation to CHCs can be obtained using various approaches (Peralta et al. 1998; Grebenshchikov et al. 2012; Gurfinkel et al. 2015; De Angelis et al. 2017). The predicates capture the reachable states of the computation and a predicate `false` represents an error state. Henceforth whenever we refer to a program, we refer to its CHC version.

```
int a, b;
if (a ≤ 100)
  a = 100 − a;
else a = a − 100;
while (a ≥ 1)
  {a = a − 1; b = b − 2; }
assert(b ≠ 0);
```

```
c1. init(A,B) ← true.
c2. if(A,B) ← init(C,B), C ≤ 100, A = 100 − C.
c3. if(A,B) ← init(C,B), C ≥ 101, A = C − 100.
c4. while(A,B) ← if(A,B).
c5. while(A,B) ← C ≥ 1, A = C − 1, B = D − 2,
                 while(C,D).
c6. false ← A ≤ 0, while(A,B), B = 0.
```

Fig. 1: Running example: (left) original program, (right) translation to CHCs

For the given program, we want to ensure that b is non-zero after the loop. The goal is to derive initial conditions on a and b, sufficient to ensure that the assertion is never violated. We note that the assertion will *not* be violated provided the following three conditions are met: (i) if $a = 100$ then $b \neq 0$, (ii) if $a < 100$ then $2a \neq 200 - b$ and (iii) if $a > 100$ then $2a \neq 200 + b$. The conjunction of these three conditions, or equivalently $b \neq |2a - 200|$, ensures that the assertion is never violated. Automating the required reasoning is challenging because: (i) the desired result is a disjunctive constraint over expressions that need an expressive domain; (ii) the disjuncts cannot be represented as intervals, octagons or difference bound matrices (Miné 2006); (iii) information need to be propagated forwards and backwards because we lose information on b and a in the forward and in the backward direction respectively. In what follows, we show how to derive the conditions automatically.

The key contribution of this paper is a framework for deriving sufficient preconditions without a need to calculate weakest preconditions or rely on abstract domains with special properties or intricate transfer functions. This is achieved through a combination of program transformation and abstract interpretation, with the derived preconditions being successively refined through iterated transformation.

After Section 2's preliminaries, we discuss, in Section 3.1, the required transformation techniques. Section 3.2 gives iterative refinement algorithms that derive successively better (weaker) preconditions. Section 4 is an account of experimental evaluation, demonstrating practical feasibility of the technique. Section 5 concludes.

2 Preliminaries

A constrained Horn clause (CHC) is a first-order predicate logic formula of the form $\forall \mathbf{x_0} \ldots \mathbf{x_k} (p_1(\mathbf{x_1}) \wedge \ldots \wedge p_k(\mathbf{x_k}) \wedge \phi \rightarrow p_0(\mathbf{x_0}))$, where ϕ is a finite conjunction of *constraints* with respect to some constraint theory $\mathbb{T}$, $\mathbf{x_0}, \ldots, \mathbf{x_k}$ are (possibly empty) tuples of *variables*, $p_0, \ldots, p_k$ are *predicate symbols*, $p_0(\mathbf{x_0})$ is the *head* of the clause and $p_1(\mathbf{x_1}) \wedge \ldots \wedge p_k(\mathbf{x_k}) \wedge \phi$ is the *body*. Quantification over tuples of variables denotes the obvious quantification over the variables. Following the conventions of Constraint Logic Programming (CLP), such a clause is written as $p_0(\mathbf{x_0}) \leftarrow \phi, p_1(\mathbf{x_1}), \ldots, p_k(\mathbf{x_k})$. For concrete examples of CHCs we use Prolog-like syntax and typewriter font, and capital letters for variable names.

An atomic formula, or simply *atom*, is a formula $p(\mathbf{x})$ where p is a predicate symbol and $\mathbf{x}$ a tuple of arguments. *Integrity constraints* are a special kind of clause whose head is the predicate `false`. A *constrained fact* is a clause of the form $p_0(\mathbf{x_0}) \leftarrow \phi$. A set of CHCs is also called a program.

Figure 1 (right) contains an example of a set of constrained Horn clauses. The first five clauses define the behaviour of the program in Figure 1 (left) and the last clause represents a property of the program (that the variable B is non-zero after executing the program) expressed as an integrity constraint.

CHC semantics. The semantics of CHCs is obtained using standard concepts from predicate logic semantics. An *interpretation* assigns to each predicate a relation over the domain of the constraint theory $\mathbb{T}$. The predicate `false` is always interpreted as *false*. We assume that $\mathbb{T}$ is equipped with a decision procedure and a projection operator, and that it is closed under negation. We use notation $\phi|_V$ to represent the constraint formulae ϕ projected onto variables V.

An interpretation *satisfies* a set of formulas if each formula in the set evaluates to *true* in the interpretation in the standard way. In particular, *a model* of a set of CHCs is an interpretation in which each clause evaluates to *true*. A set of CHCs P is *consistent* if and only if it has a model. Otherwise it is *inconsistent*. A set of CHCs has a minimal model, which is the intersection of all its models.

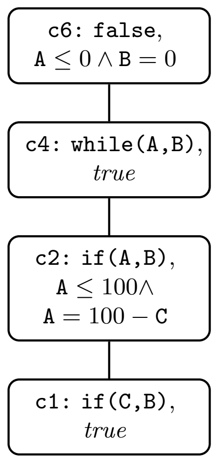
When modelling safety properties of systems using CHCs, the consistency of a set of CHCs corresponds to *safety* of the system. Thus we also refer to CHCs as being *safe* or *unsafe* when they are consistent or inconsistent respectively.

AND-trees and trace trees. Derivations for CHCs are represented by AND-trees. The following definitions of derivations and trace trees are adapted from Gallagher and Lafave (1996).

An *AND-tree* for a set of CHCs is a tree whose nodes are labelled as follows.

1. each non-leaf node corresponds to a clause (with variables suitably renamed) of the form $A \leftarrow \phi, A_1, \ldots, A_k$ and is labelled by an atom A, ϕ, and has children labelled by $A_1, \ldots, A_k$,
2. each leaf node corresponds to a clause of the form $A \leftarrow \phi$ (with variables suitably renamed) and is labelled by an atom A and ϕ, and
3. each node is labelled with the clause identifier of the corresponding clause.

Given an AND-tree t, $\mathsf{constr}(t)$ represents the conjunction of the constraints in its node labels. t is *feasible* if and only $\mathsf{constr}(t)$ is satisfiable over $\mathbb{T}$. We also represent a conjunction of constraints as a set of constraints, for example, $a = 0 \wedge b \geq 1$ as $\{a = 0, b \geq 1\}$.



Definition 1
For an atom $p(\mathbf{x})$ and a set of CHCs P we write $P \vdash_{\mathbb{T}} p(\mathbf{x})$ if there exists a feasible AND-tree with root labelled by $p(\mathbf{x})$.

The soundness and completeness of derivation trees (Jaffar et al. 1998) implies that P is inconsistent if and only if $P \vdash_{\mathbb{T}}$ `false`.

On the right is an AND-tree corresponding to the derivations of `false` using the clauses `c6` followed by `c4, c2` and `c1` from the program in Figure 1 (right).

Definition 2 (*Initial clauses and nodes*)
Let P be a set of CHCs, with a distinguished predicate p^I in P which we call the *initial predicate*. The *constrained facts* $\{(p^I(\mathbf{x}) \leftarrow \theta) \mid (p^I(\mathbf{x}) \leftarrow \theta) \in P\}$ are called the *initial clauses* of P. Let t be an AND-tree for P. A node labelled by a clause $p^I(\mathbf{x}) \leftarrow \theta$ is an *initial node* of t. We extend the term "initial predicate" and use the symbol p^I to refer also to renamed versions of the initial predicate that arise during clause transformations.

3 Precondition Inference

This section describes an approach to precondition generation. We limit our attention to sets of clauses for which every AND-tree for `false` (whether feasible or infeasible) has at least one initial node. Although it is not decidable for an arbitrary set of CHCs P whether every derivation of `false` uses the initial predicate, the above condition on AND-trees can be checked syntactically from the predicate dependency graph for P.

Definition 3 (*Safe precondition*)
Let P be a set of CHCs. Let ϕ be a constraint over $\mathbb{T}$, and let P' be the set of clauses obtained from P by replacing the initial clauses $\{(p^I(\mathbf{x}) \leftarrow \theta_i) \mid 1 \leq i \leq k\}$ by $\{(p^I(\mathbf{x}) \leftarrow \theta_i \wedge \phi) \mid 1 \leq i \leq k\}$. Then ϕ is a *safe precondition* for P if $P' \not\vdash_{\mathbb{T}}$ `false`.

Thus a safe precondition is a constraint that, when conjoined with the constraints on the initial predicate, is sufficient to block derivations of `false` (given that we assume clauses for which p^I is essential for any derivation of `false`).

Ideally we would like to find the most general, or *weakest* safe precondition. This is not computable so we aim to find a condition that is as weak as possible. The constraint *false* is always a safe precondition, albeit an uninteresting one. On the other hand, if $P \not\vdash_{\mathbb{T}}$ `false` then any constraint, including *true*, is a safe precondition for P.

We first show how a safe precondition can be derived from a set of clauses.

Definition 4 (*Safe precondition* $\mathsf{presafe}(P)$ *extracted from a set P of clauses*)
Let P be a set of clauses. The safe precondition $\mathsf{presafe}(P)$ is defined as:

$$\mathsf{presafe}(P) = \neg \bigvee \{\theta \mid (p^I(\mathbf{x}) \leftarrow \theta) \in P\}.$$

$\mathsf{presafe}(P)$ is clearly a safe precondition for P since for each initial clause $p^I(\mathbf{x}) \leftarrow \theta$ the conjunction $\mathsf{presafe}(P) \wedge \theta$ is *false*. This precondition trivially blocks any derivation of `false` since we assume that every derivation of `false` uses an initial clause. We next show how to construct a sequence $P = P_0, P_1, \ldots, P_m$ where each element of the sequence is more specialised with respect to derivations of `false`, and as a consequence, the constraints in the initial clauses are stronger. Applying Definition 4 to P_m thus yields a weaker safe precondition for P.

3.1 Specialisation of Clauses

Definition 5 (*Specialisation transformation*)
Let P be a set of clauses, and let A be an atom. We write $P \Longrightarrow_A P'$ for a *specialisation transformation* of P with respect to A, yielding a set of clauses P', such that the following holds.

- $P \vdash_{\mathbb{T}} A$ if and only if $P' \vdash_{\mathbb{T}} A$; and
- if $p^I(\mathbf{x}) \leftarrow \theta \in P$ then there exists an initial clause $p^I(\mathbf{x}) \leftarrow \phi \in P'$, where $\models_{\mathbb{T}} \phi \rightarrow \theta$.

Note that a specialisation requires not only that derivations of A are preserved, but also that the initial clauses are preserved and possibly strengthened.

Lemma 1
Let $P \Longrightarrow_{\texttt{false}} P'$ be a specialisation transformation with respect to `false`. Then $\models_{\mathbb{T}} \mathsf{presafe}(P) \rightarrow \mathsf{presafe}(P')$.

We now present specific transformations for CHCs that satisfy Definition 5. Applying these transformations enables the derivation of more precise safe preconditions. These are adapted from established techniques from the literature on CLP and Horn clause verification and analysis.

3.1.1 Specialising CHCs by Partial Evaluation (PE)

Partial evaluation (Jones et al. 1993) is a program transformation that *specialises* a program by restricting its meaning in some way. Here, we consider the partial evaluation of a set P of CHCs with respect to derivations of `false`. Informally, we wish to transform P so that derivations of `false` are preserved, but not necessarily other derivations.

The partial evaluation algorithm described here is an instantiation of the "basic algorithm" for partial evaluation of logic programs in Gallagher (1993). The basic algorithm is parameterised by an "unfolding rule" unfold_P and an abstraction operation $\mathsf{abstract}_\Psi$.

The unfolding rule unfold_P takes a set of constrained facts S, and "partially evaluates" each element of S, using the following unfolding rule. For each $(p(\mathbf{x}) \leftarrow \theta) \in S$, first construct the set of clauses $p(\mathbf{x}) \leftarrow \psi', B'$ where $p(\mathbf{x}) \leftarrow \psi, B$ is a clause in P, and ψ', B' is obtained by unfolding $\psi \wedge \theta, B$ by selecting atoms so long as they are deterministic (atoms defined by a single clause) and is not a call to an initial predicate or a recursive predicate, and ψ' is satisfiable in $\mathbb{T}$. Unfolding with this rule is guaranteed to terminate; unfold_P returns the set of constrained facts $q(\mathbf{y}) \leftarrow \psi'|_{\mathbf{y}}$ where $q(\mathbf{y})$ is an atom in B'.

Given an initial set S_0, the closure of the unfold_P operation is $\mathsf{lfp}\ \lambda S.\ S_0 \cup \mathsf{unfold}_P(S)$. However this is in general infinite. To ensure termination, we compute a set $\mathsf{cfacts}(S_0) = \mathsf{lfp}\ \lambda S.\ S_0 \cup \mathsf{abstract}_\Psi(\mathsf{unfold}_P(S))$, where the abstraction operation $\mathsf{abstract}_\Psi$ performs property-based abstraction (Grebenshchikov et al. 2012) with respect to a finite set of properties Ψ. Ψ is a finite set of constrained facts, and $\mathsf{abstract}_\Psi$ is defined as follows.

$$\begin{array}{lcl} \mathsf{abstract}_\Psi(S) & = & \{\mathsf{rep}_\Psi(p(\mathbf{x}) \leftarrow \theta) \mid (p(\mathbf{x}) \leftarrow \theta) \in S\} \\ \textit{where} & & \\ \mathsf{rep}_\Psi(p(\mathbf{x}) \leftarrow \theta) & = & p(\mathbf{x}) \leftarrow \bigwedge\{\psi \mid (p(\mathbf{x}) \leftarrow \psi) \in \Psi, \mathbb{T} \wedge \theta \models \psi\} \end{array}$$

The effect of $\mathsf{abstract}_\Psi(S)$ is to generalise each $q(\mathbf{y}) \leftarrow \theta \in S$ to $q(\mathbf{y}) \leftarrow \psi$, where ψ is the conjunction of properties in Ψ that are implied by θ. Thus only a finite number of "versions" of $q(\mathbf{y})$ can be generated, thus ensuring that $\mathsf{cfacts}(S_0)$ is finite. (Note: only one version of the predicate `false` can arise. Constrained facts for the initial predicates are a special case, and we do not abstract them. This propagates constraints to the initial clauses faster without affecting termination.)

In the implemented algorithm, the set Ψ contains the following constrained facts, generated from each clause $p(\mathbf{x}) \leftarrow \phi, p_1(\mathbf{x_1}), \ldots, p_n(\mathbf{x_n}) \in P$.

- $p(\mathbf{x}) \leftarrow \phi|_{\mathbf{x}}$ and for each $z \in \mathbf{x}$, $p(\mathbf{x}) \leftarrow \phi|_{\{z\}}$
- for $1 \leq i \leq n$, $p_i(\mathbf{x_i}) \leftarrow \phi|_{\mathbf{x_i}}$ and for each $z \in \mathbf{x_i}$, $p_i(\mathbf{x_i}) \leftarrow \phi|_{\{z\}}$.

The effect of property-based abstraction using this choice for Ψ is to create a finite number (at most $2^{|\Psi|}$) of different versions of a predicate for different call contexts and answer constraints. This choice of Ψ has been found to provide a good compromise between efficiency and precision.

Finally, partial evaluation returns the set of clauses $\mathsf{renameunfold}_{\Psi,P}(\mathsf{cfacts}(S_0))$, where $\mathsf{renameunfold}_{\Psi,P}(S)$ applies the unfolding rule to each element of S and renames the predicates in the resulting clauses according to the different versions produced by $\mathsf{abstract}_\Psi$. (The single version of `false` is not renamed.)

Example 1
Consider the partial evaluation of the clauses in Figure 1. The set Ψ consists of the following nine constrained facts extracted from the clauses as explained above:

$$\left\{ \begin{array}{l} \mathtt{if(A,B)} \leftarrow \mathtt{A} \geq \mathtt{0},\ \mathtt{if(A,B)} \leftarrow \mathtt{A} \geq \mathtt{1},\ \mathtt{init(A,B)} \leftarrow \mathtt{A} \leq \mathtt{100}, \\ \mathtt{init(A,B)} \leftarrow \mathtt{A} \geq \mathtt{101},\ \mathtt{while(A,B)} \leftarrow \mathtt{A} \geq \mathtt{0},\ \mathtt{while(A,B)} \leftarrow \mathtt{A} \geq \mathtt{1}, \\ \mathtt{while(A,B)} \leftarrow \mathtt{A} \leq \mathtt{0} \wedge \mathtt{B} = \mathtt{0},\ \mathtt{while(A,B)} \leftarrow \mathtt{A} \leq \mathtt{0},\ \mathtt{while(A,B)} \leftarrow \mathtt{B} = \mathtt{0} \end{array} \right\}$$

The partial evaluation of the clauses generate the clauses (with versions renamed) and (abstracted) constrained facts as shown in Figure 2.
Note that three versions of the `init` predicate are generated (from the new constrained facts generated in steps 3 and 4), each having different constraints. As we will see in the next section, this allows the extraction of more precise preconditions for safety of the clauses than could be obtained from the original clauses.

Lemma 2
Partial evaluation using the procedure described above is a specialisation transformation (Definition 5).

The safe precondition of the partially evaluated clauses is $\neg(\mathtt{A} \leq \mathtt{99} \vee \mathtt{A} \leq \mathtt{100} \vee \mathtt{A} \geq \mathtt{101})$, which is equivalent to *false* (over the integers). Thus partial evaluation

	Clauses	**New constrained facts**
1	$\mathtt{false} \leftarrow \mathtt{A} \leq 0, \mathtt{B} = 0, \mathtt{while\_7(A,B)}.$	$\mathtt{while(A,B)} \leftarrow \mathtt{A} \leq 0, \mathtt{B} = 0.$
2	$\mathtt{while\_7(A,B)} \leftarrow \mathtt{A} \leq 0, \mathtt{B} = 0, \mathtt{if\_6(A,B)}.$ $\mathtt{while\_7(A,B)} \leftarrow \mathtt{A} = 0, \mathtt{B} = 0, \mathtt{C} = 1, \mathtt{D} = 2,$ $\mathtt{while\_5(C,D)}.$	$\mathtt{while(A,B)} \leftarrow \mathtt{A} \geq 1.$ $\mathtt{if(A,B)} \leftarrow \mathit{true}.$
3	$\mathtt{while\_5(A,B)} \leftarrow \mathtt{A} \geq 1, \mathtt{if\_2(A,B)}.$ $\mathtt{while\_5(A,B)} \leftarrow \mathtt{A} \geq 1, \mathtt{C} - \mathtt{A} = 1, \mathtt{D} - \mathtt{B} = 2,$ $\mathtt{while\_5(C,D)}.$ $\mathtt{if\_6(A,B)} \leftarrow \mathtt{A} \geq 0, \mathtt{A} + \mathtt{C} = 100, \mathtt{init\_4(C,B)}.$ $\mathtt{if\_6(A,B)} \leftarrow \mathtt{A} \geq 1, \mathtt{C} - \mathtt{A} = 100, \mathtt{init\_3(C,B)}.$	$\mathtt{if(A,B)} \leftarrow \mathtt{A} \geq 1.$ $\mathtt{init(A,B)} \leftarrow \mathtt{A} \geq 101$ $\mathtt{init(A,B)} \leftarrow \mathtt{A} \leq 100.$
4	$\mathtt{if\_2(A,B)} \leftarrow \mathtt{A} \geq 1, \mathtt{A} + \mathtt{C} = 100, \mathtt{init\_1(C,B)}.$ $\mathtt{if\_2(A,B)} \leftarrow \mathtt{A} \geq 1, \mathtt{C} - \mathtt{A} = 100, \mathtt{init\_3(C,B)}.$ $\mathtt{init\_4(A,B)} \leftarrow \mathtt{A} \leq 100.$ $\mathtt{init\_3(A,B)} \leftarrow \mathtt{A} \geq 101.$	$\mathtt{init(A,B)} \leftarrow \mathtt{A} \leq 99.$
5	$\mathtt{init\_1(A,B)} \leftarrow \mathtt{A} \leq 99.$	None

Fig. 2: Steps performed during the run of partial evaluation

$$\begin{array}{l}
\mathtt{false} \leftarrow \mathtt{A} \geq 0, \mathtt{p(A,B)}.\\
\mathtt{p(A,B)} \leftarrow \mathtt{C} \geq \mathtt{A}, \mathtt{p(C,B)}.\\
\mathtt{p(A,B)} \leftarrow \mathtt{A} = \mathtt{B}.
\end{array}
\qquad
\begin{array}{l}
\mathtt{false} \leftarrow \mathtt{A} \geq 0, \underline{\mathtt{B} \geq \mathtt{A}, \mathtt{A} \geq 0}, \mathtt{p(A,B)}.\\
\mathtt{p(A,B)} \leftarrow \mathtt{C} \geq \mathtt{A}, \underline{\mathtt{B} \geq \mathtt{C}, \mathtt{C} \geq 0}, \mathtt{p(C,B)}.\\
\mathtt{p(A,B)} \leftarrow \mathtt{A} = \mathtt{B}, \underline{\mathtt{B} \geq \mathtt{A}, \mathtt{A} \geq 0}.
\end{array}$$

Fig. 3: Example program (left) and its constraint specialised version (right)

has not improved the safe precondition compared to the original clauses in Figure 1. However, the splitting of the initial clauses enables a further specialisation, which is described next.

3.1.2 Transforming CHCs by Constraint Specialisation (CS)

Constraint specialisation is a transformation that strengthens the constraints in a set of CHCs, while preserving derivations of a given atom. Consider the following simple example in Figure 3 (left) that motivates the principles of the transformation. Assume we wish to preserve derivations of `false`. The transformation in Figure 3 (right) is a constraint specialisation with respect to `false`. The strengthened constraints are obtained by recursively propagating $\mathtt{A} \geq 0$ top-down from the goal `false` and $\mathtt{A} = \mathtt{B}$ bottom-up from the constrained fact. An invariant $\mathtt{B} \geq \mathtt{A}, \mathtt{A} \geq 0$ for the derived answers of the recursive predicate `p(A,B)` in derivations of `false` is computed and conjoined to each call to `p` in the clauses (underlined in the clauses in Figure 3 (right)).

Definition 6 (*Constraint specialisation*)
A constraint specialisation of P with respect to a goal A is a transformation in which each constraint ϕ in a clause of P is replaced by a constraint ψ where $\models_{\mathbb{T}} \psi \rightarrow \phi$, such that the resulting set of clauses is a specialisation transformation (Definition 5) of P with respect to A.

In our experiments, the combined top-down and bottom-up propagation of constraints illustrated above is achieved by abstract interpretation over the domain

```
false ← A = 0, B = 0, while_7(A, B).   while_7(A, B) ← A = 0, B = 0, if_6(A, B).
while_7(A, B) ← A = 0, B = 0, C = 1, D = 2, while_5(C, D).
if_6(A, B) ← A = 0, B = 0, C = 100, init_4(C, B).
while_5(A, B) ← A ≥ 1, 2A − B = 0, if_2(A, B).
while_5(A, B) ← A ≥ 1, 2A = B, C − A = 1, D − 2A = 2, while_5(C, D).
if_2(A, B) ← A ≥ 1, 2A = B, A + C = 100, init_1(C, B).
if_2(A, B) ← A ≥ 1, 2A = B, C − A = 100, init_3(C, B).
init_4(A, B) ← A = 100, B = 0.   init_3(A, B) ← A ≥ 101, 2A − B = 200.
init_1(A, B) ← A ≤ 99, 2A + B = 200.
```

Fig. 4: Constraint specialisation of the partially evaluated clauses in Figure 2

of convex polyhedra applied to a query-answer transformed version of the set of CHCs. The method is described in detail in Kafle and Gallagher (2017a). The result of applying constraint specialisation to the output of partial evaluation of the running example is shown in Figure 4. Note that the second clause for `if_6` has been eliminated, since its constraint was specialised to *false*.

The safe precondition derived after constraint specialisation from the initial clauses in Figure 4 is $\neg((\mathtt{A} = 100 \wedge \mathtt{B} = 0) \vee (\mathtt{A} \leq 99 \wedge 2\mathtt{A} + \mathtt{B} = 200) \vee (\mathtt{A} \geq 101 \wedge 2\mathtt{A} - \mathtt{B} = 200))$. This simplifies (over the integers) to $\mathtt{B} \neq |2\mathtt{A} - 200|$, which is the condition obtained in Section 1 and is optimal (weakest).

3.1.3 Transforming CHCs by Trace Elimination (TE)

Let P be a set of CHCs and let t be an AND-tree for P. It is possible to construct a set of clauses P' which preserves the set of AND-trees (modulo predicate renaming) of P, apart from t. The transformation from P to P' is called *trace elimination* (of t). We have previously described a technique for trace elimination (Kafle and Gallagher 2017b), based on the difference operation on finite tree automata. In that work, trace elimination played the role of a refinement operation, in which infeasible traces were removed from a set of CHCs in a counterexample-guided verification algorithm in the CEGAR style (Clarke et al. 2003).

For the purpose of deriving safe preconditions of a set of clauses P, we apply trace elimination to eliminate both infeasible and feasible AND-trees. AND-trees for `false` are obtained naturally from transformations such as partial evaluation or constraint specialisation. First consider the elimination of an infeasible AND-tree.

Lemma 3
Let P' be the result of eliminating an infeasible AND-tree t for `false` from P. Then $P \Longrightarrow_{\mathtt{false}} P'$.

Hence in this case we have $\models_{\mathbb{T}} \mathsf{presafe}(P) \rightarrow \mathsf{presafe}(P')$. However, the elimination of a feasible AND-tree t for `false` is not as straightforward. Nevertheless, we can still use this transformation to derive safe preconditions, by the following lemma.

Lemma 4
Let P' be the result of eliminating a feasible AND-tree t for `false` from P. Let $p^I(\mathbf{x})$ be the atom label of an initial node of t and let $\theta = \mathsf{constr}(t)|_{\mathbf{x}}$. Then $\mathsf{presafe}(P) = \mathsf{presafe}(P') \wedge \neg\theta$.

The usefulness of trace elimination is twofold. Firstly, it can cause splitting of the initial predicates, resulting in disjunctive pre-conditions. Secondly, the elimination of a feasible trace acts as a decomposition of the problem.

3.2 Inferring Weaker Preconditions

We can combine the various transformations to derived weaker preconditions, as shown in the following two propositions.

Proposition 1
Let $P = P_0$ and let the sequence $P_0, P_1, \ldots, P_m$ be a sequence such that $P_i \Longrightarrow_{\texttt{false}} P_{i+1}$ $(0 \leq i < m)$. Then $\models_{\mathbb{T}} \mathsf{presafe}(P) \rightarrow \mathsf{presafe}(P_m)$.

Proposition 1 allows the use of partial evaluation, constraint specialisation and elimination of infeasible traces, in any order, in order to derive a weaker safe pre-condition.

If we also eliminate feasible traces, then we have to keep track of the substitutions arising from the eliminated trees.

Proposition 2
Let $P = P_0$, $\psi_0 = true$ and let the sequence $(P_0, \psi_0), (P_1, \psi_1), \ldots, (P_m, \psi_m)$ be a sequence of pairs where for $(0 \leq i < m)$

- either $P_i \Longrightarrow_{\texttt{false}} P_{i+1}$ and $\psi_i = \psi_{i+1}$, or
- P_{i+1} is obtained by eliminating a feasible trace t from P_i , and $\psi_{i+1} = \psi_i \wedge \neg\theta$, where $\neg\theta$ is the constraint extracted from t, as in Lemma 4.

Then $\models_{\mathbb{T}} \mathsf{presafe}(P) \rightarrow (\mathsf{presafe}(P_m) \wedge \psi_m)$.

Proposition 1 is a special case of Proposition 2, as if we do not eliminate any feasible trees, then ψ_m is *true* and so $\models_{\mathbb{T}} \mathsf{presafe}(P) \rightarrow \mathsf{presafe}(P_m)$.

As we showed, applying partial evaluation followed by constraint specialisation for our running example was sufficient to derive the weakest safe precondition. However, in more complex cases we need one or more iterations of these operations, possibly with the elimination of feasible AND-trees as well. In the appendix, we show an example (Figure 5) in which repeated application of partial evaluation followed by constraint specialisation does not achieve a useful result, but where the elimination of a single feasible AND-tree causes an optimal precondition to be generated. However, there is a performance-precision trade off when removing a feasible AND-tree. Trace elimination helps derive precise preconditions at the cost of performance; the *Fischer* protocol is an example of this. It requires 4 iterations of PE followed by CS to generate the optimal precondition (obtained in ≈8 seconds), whereas these iterations interleaved by trace elimination require only 3 iterations (but obtained in ≈30 seconds).

4 Experimental Evaluation

Benchmarks. We experimented with three kinds of benchmark. (1) *Unsafe I*: Examples that are known to be unsafe, where the initial states are over-general. In such cases the aim of safe precondition generation is to find out whether there is a useful subset of the initial states that is safe. (2) *Unsafe II*: Examples that are known to be unsafe, where the initial state is a counterexample state from which `false` can be derived. In this case it is pointless to try to find a safe subset as above, so we remove the given constraint on the initial state, and then try to derive a non-trivial safe precondition. (3) *Safe*: Examples that are safe for given initial states. In such cases, our aim is to try to weaken the conditions on the initial states. This is done by removing the given constraints from the initial states and then deriving safe preconditions. If we can generate safe preconditions that are more general than the original constraints then we have generalised the program without losing safety.

For the experiments, we collected a set of 241 (188 safe/53 unsafe) programs from a variety of sources. Most are from the repositories of state-of-the-art software verification tools such as DAGGER (Gulavani et al. 2008), TRACER (Jaffar et al. 2012), InvGen (Gupta and Rybalchenko 2009), and from the TACAS 2013 Software Verification Competition (Beyer 2013), with size up to approximately 500 lines of code.[1] Other examples are from the literature on precondition generation, backwards analysis or parameter synthesis (Bakhirkin et al. 2014; Miné 2012a; Miné 2012b; Moy 2008; Bakhirkin and Monniaux 2017; Cassez et al. 2017) and manually translated to CHCs. Finally there are examples crafted by us; these are simple but non-trivial examples whose precondition is easy to derive manually.

Implementation. We implemented an algorithm that builds a sequence as defined in Proposition 2, of length $3n+2$ ($n \geq 0$), iteratively applying the transformations *pe* (partial evaluation), *cs* (constraint specialisation) and *te* (trace elimination). The safe precondition for P is $\mathsf{presafe}(cs \circ pe \circ (te \circ cs \circ pe)^n(P))$ ($n \geq 0$). The implementation is based on components from the RAHFT verifier (Kafle et al. 2016). This accepts CHCs (over the background theory of linear arithmetic) as input and returns a Boolean combination of linear constraints in terms of the initial state variables as a precondition. The tool is written in Ciao Prolog (Bueno et al. 1997) and uses Yices 2.2 (Dutertre 2014) and the Parma Polyhedra Library (Bagnara et al. 2008) for constraint manipulation. The experiments were carried out on a MacBook Pro with a 2.7 GHz Intel Core i5 processor and 16 GB memory running OS X 10.11.6, with a timeout of 300 seconds for each example. The results are shown in Table 1, for varying number of specialisation iterations n.

Discussion. The classifications "more general" and "non-trivial" in Table 1 relate the derived precondition I with the original condition on the initial states O. If $\models_{\mathbb{T}} I \not\equiv \mathit{false}$ then the result is non-trivial. If $\models_{\mathbb{T}} O \rightarrow I$ then the derived precon-

[1] Translated to CHCs using the program specialisation approach presented in De Angelis et al. (2017). Thanks to the authors of that work for providing these benchmarks.

	$n=0$	$n=1$	$n=2$	$n=3$
Safe instances (188)				
non-trivial (more general)	119 (101)	143 (125)	156 (129)	160 (131)
trivial/timeouts	69/0	45/3	32/10	28/16
avg. time (sec.)	1.45	14.69	27.52	36.73
Unsafe I instances (17)				
non-trivial	16	17	17	17
trivial/timeouts	1/0	0/0	0/0	0/0
avg. time (sec.)	0.23	0.82	1.64	3.35
Unsafe II instances (36)				
non-trivial	9	12	12	12
trivial/timeouts	27/0	24/2	24/7	24/7
avg. time (sec.)	3.38	50.41	64.72	70.91

Table 1: Results on 241 (188 safe and 53 unsafe) programs; timeout 5 minutes

dition is more general than the given initial states. For the *safe* benchmarks, the "more general" results are a subset of the "non-trivial" results, while for the *unsafe* benchmarks, the result cannot be more general than the original (unsafe) condition and so there are no "more general" results.

For the *safe* benchmarks, the algorithm succeeds for $n = 3$ in generalising the safe initial conditions in 131 of the 188 benchmarks, and returns a non-trivial safe precondition in 160 of them. The remainder either return trivial results or a timeout. A higher proportion of the *unsafe* benchmarks return a trivial safe precondition, even when the initial state constraints are removed. A possible reason is that some of these unsafe programs are designed with an internal bug, and thus have no safe initial states. If the analysis returns a trivial safe precondition, it could be due to imprecision of the analysis, but could also be an indication to the programmer to look for the problem elsewhere than in the initial states.

The results in the column $n = 0$ show that the specialisation ($cs \circ pe$) alone can infer non-trivial preconditions for a large number of benchmarks, namely 63% (safe) and 37% (unsafe) instances both in less than 10 seconds. Among 119 non-trivial safe instances, 101 are generalised constraints.

Further specialisation ($n > 0$) increases the number of non-trivial and generalised preconditions by relatively small percentages of the total. The increased precision of the preconditions comes at a significant cost in time. For Safe, Unsafe I, and Unsafe II instances, the average time goes from 1.45, 0.23 and 3.38 seconds, respectively, when $n = 0$, to 36.73, 3.35 and 70.91 seconds, when $n = 3$. However, our prototype implementation is amenable to much optimisation, including sharing results from one iteration to the next, which could reduce the overhead. When there is a timeout in iteration n, we present the precondition generated in iteration $n - 1$. Therefore, since none of the instances timed out in iteration $n = 0$, the numbers of trivial and non-trivial instances sums up to the total number of instances.

For the categories of literature and hand-crafted benchmarks in which we know the weakest safe precondition, the tool is able to reproduce the results from the literature in $n \leq 1$ iterations, except for *Fischer's protocol* which requires 3 iterations (see Appendix).

5 Concluding Remarks

We have presented an iterative framework for computing a sufficient precondition of a program with respect to assertions. Rather than relying on weakest precondition calculation or intricate transfer functions, it uses off-the-shelf components from program transformation and abstract interpretation. It does not depend on specific abstract domain properties such as pseudo-complementation, but is domain-independent and generic. The results on set of benchmarks are promising. We are currently investing the conditions under which the derived preconditions are the weakest possible, as well as other termination criteria for the refinement. The benefit of such improved criteria is to generate optimal preconditions.

References

Bagnara, R., Hill, P. M., and Zaffanella, E. 2008. The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. *Science of Computer Programming 72,* 1–2, 3–21.

Bakhirkin, A., Berdine, J., and Piterman, N. 2014. Backward analysis via over-approximate abstraction and under-approximate subtraction. In *SAS*, M. Müller-Olm and H. Seidl, Eds. LNCS, vol. 8723. Springer, 34–50.

Bakhirkin, A. and Monniaux, D. 2017. Combining forward and backward abstract interpretation of Horn clauses. In *SAS*, F. Ranzato, Ed. LNCS, vol. 10422. Springer, 23–45.

Ball, T., Cook, B., Levin, V., and Rajamani, S. K. 2004. SLAM and static driver verifier: Technology transfer of formal methods inside Microsoft. In *IFM 2004*, E. A. Boiten, J. Derrick, and G. Smith, Eds. LNCS, vol. 2999. Springer, 1–20.

Beyer, D. 2013. SV-COMP 2013. In *TACAS*, N. Piterman and S. A. Smolka, Eds. LNCS, vol. 7795. Springer, 594–609.

Beyer, D., Henzinger, T. A., Majumdar, R., and Rybalchenko, A. 2007. Path invariants. In *PLDI*, J. Ferrante and K. S. McKinley, Eds. ACM, 300–309.

Bueno, F., Cabeza, D., Carro, M., Hermenegildo, M., López-García, P., and Puebla, G. 1997. The Ciao Prolog system. ref. manual. Tech. Rep. CLIP3/97.1, School of Computer Science, UPM. `http://www.clip.dia.fi.upm.es/`.

Cassez, F., Jensen, P. G., and Larsen, K. G. 2017. Refinement of trace abstraction for real-time programs. In *RP*, M. Hague and I. Potapov, Eds. LNCS, vol. 10506. Springer, 42–58.

Clarke, E. M., Grumberg, O., Jha, S., Lu, Y., and Veith, H. 2003. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM 50,* 5, 752–794.

Cousot, P. and Cousot, R. 1992. Abstract interpretation and application to logic programs. *J. Log. Program. 13,* 2&3, 103–179.

Cousot, P., Cousot, R., and Logozzo, F. 2011. Precondition inference from intermittent assertions and applications to contracts on collections. In *VMCAI*, R. Jhala and D. Schmidt, Eds. LNCS, vol. 6538. Springer, 150–168.

De Angelis, E., Fioravanti, F., Pettorossi, A., and Proietti, M. 2014. Program verification via iterated specialization. *Sci. Comput. Program. 95*, 149–175.

De Angelis, E., Fioravanti, F., Pettorossi, A., and Proietti, M. 2017. Semantics-based generation of verification conditions via program specialization. *SCP 147*, 78–108.

Dutertre, B. 2014. Yices 2.2. In *CAV*, A. Biere and R. Bloem, Eds. Lecture Notes in Computer Science, vol. 8559. Springer, 737–744.

Gallagher, J. P. 1993. Specialisation of logic programs: A tutorial. In *Proc. ACM SIGPLAN Symp. PEPM'93*. ACM Press, 88–98.

Gallagher, J. P. and Lafave, L. 1996. Regular approximation of computation paths in logic and functional languages. In *Partial Evaluation*, O. Danvy, R. Glück, and P. Thiemann, Eds. LNCS, vol. 1110. Springer, 115–136.

Grebenshchikov, S., Lopes, N. P., Popeea, C., and Rybalchenko, A. 2012. Synthesizing software verifiers from proof rules. In *ACM SIGPLAN PLDI*, J. Vitek, H. Lin, and F. Tip, Eds. ACM, 405–416.

Gulavani, B. S., Chakraborty, S., Nori, A. V., and Rajamani, S. K. 2008. Automatically refining abstract interpretations. In *TACAS*, C. R. Ramakrishnan and J. Rehof, Eds. LNCS, vol. 4963. Springer, 443–458.

Gupta, A. and Rybalchenko, A. 2009. Invgen: An efficient invariant generator. In *CAV*, A. Bouajjani and O. Maler, Eds. LNCS, vol. 5643. Springer, 634–640.

Gurfinkel, A., Kahsai, T., Komuravelli, A., and Navas, J. A. 2015. The SeaHorn verification framework. In *CAV , Part I*, D. Kroening and C. S. Păsăreanu, Eds. LNCS, vol. 9206. Springer, 343–361.

Howe, J. M., King, A., and Lu, L. 2004. Analysing logic programs by reasoning backwards. In *PDCL*, M. Bruynooghe and K. Lau, Eds. LNCS, vol. 3049. Springer, 152–188.

Jaffar, J., Maher, M., Marriott, K., and Stuckey, P. 1998. The semantics of constraint logic programs. *Journal of Logic Programming 37,* 1–3, 1–46.

Jaffar, J., Murali, V., Navas, J. A., and Santosa, A. E. 2012. Tracer: A symbolic execution tool for verification. In *CAV*, P. Madhusudan and S. A. Seshia, Eds. LNCS, vol. 7358. Springer, 758–766.

Jhala, R. and McMillan, K. L. 2006. A practical and complete approach to predicate refinement. In *TACAS*, H. Hermanns and J. Palsberg, Eds. LNCS, vol. 3920. Springer, 459–473.

Jones, N., Gomard, C., and Sestoft, P. 1993. *Partial Evaluation and Automatic Software Generation*. Prentice Hall.

Kafle, B. and Gallagher, J. P. 2017a. Constraint specialisation in Horn clause verification. *Sci. Comput. Program. 137*, 125–140.

Kafle, B. and Gallagher, J. P. 2017b. Horn clause verification with convex polyhedral abstraction and tree automata-based refinement. *Computer Languages, Systems & Structures 47*, 2–18.

Kafle, B., Gallagher, J. P., and Morales, J. F. 2016. RAHFT: A tool for verifying Horn clauses using abstract interpretation and finite tree automata. In *CAV, Part I*, S. Chaudhuri and A. Farzan, Eds. LNCS, vol. 9779. Springer, 261–268.

Marriott, K. and Søndergaard, H. 1993. Precise and efficient groundness analysis for logic programs. *LOPLAS 2(1-4)*, 181–196.

Miné, A. 2006. The octagon abstract domain. *Higher-Order and Symbolic Computation 19,* 1, 31–100.

Miné, A. 2012a. Inferring sufficient conditions with backward polyhedral underapproximations. *Electr. Notes Theor. Comput. Sci. 287*, 89–100.

Miné, A. 2012b. Sufficient condition polyhedral prototype analyzer. `https://www-apr.lip6.fr/~mine/banal/syntax.html`. Accessed: 2018-01-23.

Moy, Y. 2008. Sufficient preconditions for modular assertion checking. In *VMCAI*, F. Logozzo, D. A. Peled, and L. D. Zuck, Eds. LNCS, vol. 4905. Springer, 188–202.

Peralta, J., Gallagher, J. P., and Sağlam, H. 1998. Analysis of imperative programs through analysis of constraint logic programs. In *SAS*, G. Levi, Ed. LNCS, vol. 1503. 246–261.

Appendix I: Examples

The example in Figure 5 is taken from Beyer et al. (2007). The optimal precondition for this program is $\mathtt{init(I,A,B,N)} \leftarrow \mathtt{N} \leq \mathtt{I} \wedge \mathtt{A}+\mathtt{B} = 3*\mathtt{N}$. In order to derive this, one needs to propagate constraints from the third and the fourth clauses (constrained facts corresponding to the predicate `l`) to the `init` clause. Since these constraints are disjunctive (arising from two different clauses), the propagation should be able to split the `init` predicate. The role of PE was to do that but it will not since the `l` predicate is recursive and is not unfolded to control the blowup.

```
false ← init(I, A, B, N), l(I, A, B, N).
l(I, A, B, N) ← I < N, l_body(A, B, A1, B1), I1 = I + 1, l(I1, A1, B1, N).
l(I, A, B, N) ← I ≥ N, A + B > 3 ∗ N.
l(I, A, B, N) ← I ≥ N, A + B < 3 ∗ N.
l_body(A0, B0, A1, B1) ← A1 = A0 + 1, B1 = B0 + 2.
l_body(A0, B0, A1, B1) ← A1 = A0 + 2, B1 = B0 + 1.
init(I, A, B, N).
```

Fig. 5: Example requiring trace elimination.

The constraint specialisation of the program in Figure 5 is shown in Figure 6. If we derive a precondition from this program, we will get trivial *false*. As a next step, we search for a derivation (counterexample) violating the safety. The trace (represented as a term), namely, `c1(c10,c2(c8,c5(c8,c5(c8,c5(c8,c6))))))` is a counterexample. Then we remove this from the program in Figure 6 using the automata theoretic approach described in Kafle and Gallagher (2017b).

```
c1.  false ← init(A, B, C, D), l_3(A, B, C, D).
c2.  l_3(A, B, C, D) ← −C + F >= 1, −A + D > 0, C − F >= −2, A − E = −1,
       B + C − F − G = −3, l_body_2(B, C, G, F), l_1(E, G, F, D).
c3.  l_3(A, B, C, D) ← B + C − 3 ∗ D > 0, A − D >= 0.
c4.  l_3(A, B, C, D) ← −B − C + 3 ∗ D > 0, A − D >= 0.
c5.  l_1(A, B, C, D) ← −C + F >= 1, −A + D > 0, C − F >= −2, A − E = −1,
       B + C − F − G = −3, l_body_2(B, C, G, F), l_1(E, G, F, D).
c6.  l_1(A, B, C, D) ← B + C − 3 ∗ D > 0, −A + D > −1, A − D >= 0.
c7.  l_1(A, B, C, D) ← −B − C + 3 ∗ D > 0, −A + D > −1, A − D >= 0.
c8.  l_body_2(A, B, C, D) ← A − C = −1, B − D = −2.
c9.  l_body_2(A, B, C, D) ← A − C = −2, B − D = −1.
c10. init(A, B, C, D).
```

Fig. 6: The constraint specialisation of the program in Figure 5, the clauses are numbered for presentation purpose.

The removal causes the splitting of the predicate `l`, which the partial evaluation can take advantage of in the next iteration. Re-application of PE followed by CS generates the following clauses for `init` predicates (other clauses are not shown).

```
init_1(A, C, D, B) ← B > A.
init_2(A, C, D, B) ← A >= B, C + D > 3B.
init_3(A, C, D, B) ← A >= B, 3 ∗ B > C + D.
```

Then the precondition would be:

$$\mathtt{init}(\mathtt{A},\mathtt{C},\mathtt{D},\mathtt{B}) \leftarrow \neg((\mathtt{B} > \mathtt{A}) \vee (\mathtt{A} \geq \mathtt{B} \wedge \mathtt{C} + \mathtt{D} > 3\mathtt{B}) \vee (\mathtt{A} \geq \mathtt{B} \wedge 3 * \mathtt{B} > \mathtt{C} + \mathtt{D}))$$

Simplifying the formula and mapping to the original variables, we obtain the following formula as the final precondition

$$\mathtt{init}(\mathtt{I},\mathtt{A},\mathtt{B},\mathtt{N}) \leftarrow \mathtt{N} \leq \mathtt{I} \wedge \mathtt{A} + \mathtt{B} = 3 * \mathtt{N}.$$

Program	**Precondition**
bakhirkin-fig3 (Bakhirkin et al. 2014)	$(1 \leq a \leq 99 \rightarrow b \geq 1) \ \wedge \ (a \leq 0 \rightarrow b \neq 0)$
bakhirkin (Bakhirkin et al. 2014)	$1 \leq a \leq 60 \vee a \geq 100$
mine (Miné 2012a)	$0 \leq a \leq 5$
mon_fig1 (Bakhirkin and Monniaux 2017)	$a = b \ \wedge \ a \geq 0$
moy (Moy 2008)	$b < 1 \ \vee \ (b < 2 \wedge a > 0)$
navas2 (crafted)	$a \leq 99 \ \vee \ b \geq 100$
simple_function (Miné 2012b)	$6 \leq a \leq 61$
test_both_branches (Miné 2012b)	$3 \leq a \leq 17$
test_nondet_body (Miné 2012b)	$6 \leq a \leq 13$
test_nondet_cond (Miné 2012b)	$3 \leq a \leq 17$
test_then_branch (Miné 2012b)	$10 \leq a \leq 20$
fischer (Cassez et al. 2017)	$a + 2 * c < b \vee a < 0 \vee b < 0 \vee c \leq 0$
Jhala (Jhala and McMillan 2006)	$a < 0 \vee a \geq b \vee c \neq d$
Ball SLAM (Ball et al. 2004)	$b < c$
client ssh protocol	$b < a \vee b < 2 \vee a > 3$
Beyer et al. (2007)	$n \leq i \wedge a + b = 3 * n$

Fig. 7: Derived safe preconditions for a set of examples from the literature. Results were generated in at most 1 iteration in less than 1 second, except for *fischer* which requires 3 iterations and 35 seconds.

Appendix II: Proofs

Lemma 1. Let $P \Longrightarrow_{\mathtt{false}} P'$ be a specialisation transformation with respect to `false`. Then $\models_{\mathbb{T}} \mathsf{presafe}(P) \rightarrow \mathsf{presafe}(P')$

Proof
This follows immediately from Definitions 4 and 5. □

Lemma 2. Partial evaluation is a specialisation transformation (Definition 5).

Proof
The algorithm satisfies the standard condition of partial evaluation that it preserves derivations of the given goal atom. The strengthening of the initial clauses follows from the fact that our unfolding rule does not unfold the initial predicate. Hence the result contains the initial clauses from the original, with constraints possibly strengthened by the call constraints in the algorithm. (If a clause is never called, its constraint is strengthened to *false*). □

Lemma 3. Let P' be the result of eliminating an infeasible AND-tree t for `false` from P. Then $P \Longrightarrow_{\mathtt{false}} P'$.

Proof
All derivations of `false` are preserved, and the transformation generates only predicate-renamed copies of the original clauses, hence the initial clauses are preserved. $\square$

Lemma 4. Let P' be the result of eliminating a feasible AND-tree t for `false` from P. Let $p^I(\mathbf{x})$ be the atom label of an initial node of t and let $\theta = \mathsf{constr}(t)|_{\mathbf{x}}$. Then $\mathsf{presafe}(P) = \mathsf{presafe}(P') \wedge \neg\theta$.

Proof
$\neg\theta$ is a sufficient condition, when conjoined with the body of the clause labelling the initial node, to make t infeasible. All other derivations of `false` from P are preserved in P'. Hence the conjunction of $\neg\theta$ and $\mathsf{presafe}(P')$ is a safe precondition for P. $\square$

Proposition 1. Let $P = P_0$ and let the sequence $P_0, P_1, \ldots, P_m$ be a sequence such that $P_i \Longrightarrow_{\texttt{false}} P_{i+1}$ $(0 \leq i < m)$. Then $\models_{\mathbb{T}} \mathsf{presafe}(P) \rightarrow \mathsf{presafe}(P_m)$.

Proof
By induction on the length of the sequence, applying Lemma 1. $\square$

Proposition 2. Let $P = P_0, \psi_0 = \mathit{true}$ and let the sequence $(P_0, \psi_0), (P_1, \psi_1), \ldots,$ (P_m, ψ_m) be a sequence of pairs where for $(0 \leq i < m)$

- either $P_i \Longrightarrow_{\texttt{false}} P_{i+1}$ and $\psi_i = \psi_{i+1}$, or
- P_{i+1} is obtained by eliminating a feasible trace t from P_i , and $\psi_{i+1} = \psi_i \wedge \neg\theta$, where $\neg\theta$ is the constraint extracted from t, as in Lemma 4.

Then $\models_{\mathbb{T}} \mathsf{presafe}(P) \rightarrow (\mathsf{presafe}(P_m) \wedge \psi_m)$.

Proof
By induction on the length of the sequence, applying Lemma 1 and Lemma 4. $\square$