Constraint Solving for Program Verification: Theory and Practice by Example

Andrey Rybalchenko

Technische Universität München

Abstract. Program verification relies on the construction of auxiliary assertions describing various aspects of program behaviour, e.g., inductive invariants, resource bounds, and interpolants for characterizing reachable program states, ranking functions for approximating number of execution steps until program termination, or recurrence sets for demonstrating non-termination. Recent advances in the development of constraint solving tools offer an unprecedented opportunity for the efficient automation of this task. This paper presents a series of examples illustrating algorithms for the automatic construction of such auxiliary assertions by utilizing constraint solvers as the basic computing machinery.

1 Introduction

Program verification has a long history of using constraint-based algorithms as main building blocks. In principle, constraint-based algorithms follow two major steps. First, during the constraint generation step a program property of interest is formulated as a set of constraints. Any solution to these constraints determines the property. During the second step, the constraints are solved. Usually, this step is executed using a separate constraint solving procedure. Such separation of concerns, i.e., constraint generation vs. solving, can liberate the designer of the verification tool from the tedious task of creating a dedicated algorithm. Instead, an existing off-the-shelf constraint solver can be put to work.

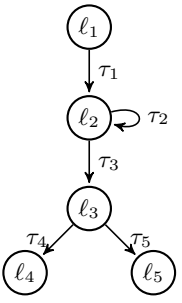
In this paper, we show how constraints can be used to prove program (non-)termination and safety by generating ranking functions, interpolants, invariants, resource bounds, and recurrence sets. First, we focus on assertions expressed in linear arithmetic, which form a practically important class, and then show extensions with uninterpreted function symbols. Our presentation uses a collection on examples to illustrate the algorithms.

The rest of the paper is organized as follows. Section 2 illustrates the generation of linear ranking functions. In Section 3, we show how linear interpolants can be computed. Section 4 presents linear invariant generation and an optimization technique that exploits program test cases. It also shows how invariant generation can be adapted to compute bounds on resource consumption. We use an additional, possibly non-terminating program in Section 5 to illustrate the construction of recurrence sets for proving non-termination. Section 6 shows how

T. Touili, B. Cook, and P. Jackson (Eds.): CAV 2010, LNCS 6174, pp. 57–71, 2010.

```
main(int x, int y, int z) {
  assume(y >= z);
  while (x < y) {
    x++;
  }
  assert(x >= z);
}
```

(a)



(b)

$$\rho_1 = (y \geq z \wedge x' = x \wedge y' = y \wedge z' = z)$$
$$\rho_2 = (x + 1 \leq y \wedge x' = x + 1 \wedge y' = y \wedge z' = z)$$
$$\rho_3 = (x \geq y \wedge x' = x \wedge y' = y \wedge z' = z)$$
$$\rho_4 = (x \geq z \wedge x' = x \wedge y' = y \wedge z' = z)$$
$$\rho_5 = (x + 1 \leq z \wedge x' = x \wedge y' = y \wedge z' = z)$$

(c)

Fig. 1. An example program (a), its control-flow graph (b), and the corresponding transition relations (c)

constraint-based algorithms for the synthesis of linear assertions can be extended to deal with the combination of linear arithmetic and uninterpreted functions. Here, we use the interpolation algorithm as an example.

We use the program shown in Figure 1 as a source of termination, interpolation and safety proving obligations. When translating the program instructions into the corresponding transition relations we approximate integer program variables by rationals, in order to reduce the complexity the resulting constraint generation and solving tasks. Hence, the relation ρ_2 has a guard $x + 1 \leq y$. Furthermore, the failure of the assert statement is represented by reachability of the control location ℓ_5.

2 Linear Ranking Functions

Program termination is an important property that ensures its responsiveness. Proving program terminations requires construction of ranking functions that over-approximate the number of execution steps that the program can make from a given state until termination. Linear ranking functions express such approximations by linear assertions over the program variables.

Input. We illustrate the construction of ranking functions on the while loop from the program in Figure 1, as shown below. See [5] for its detailed description and pointers to the related work.

```
while (x < y) {
  x++;
}
```

We deliberately choose a loop that neither contains further nesting loops nor branching control flow inside the loop body in order to highlight the main ideas of the constraint-based ranking function generation.

Our algorithm will search for a linear expression over the program variables that proves termination. Such an expression is determined by the coefficients of the occurring variables. Let f_x and f_y be the coefficients for the variables x and y, respectively. Since the program variable z does not play a role in the loop, to simplify the presentation we do not take it into consideration.

A linear expression is a ranking function if its value is bounded from below for all states on which the loop can make a step, and is decreasing by some a priory fixed positive amount. Let δ_0 be the lower bound for the value of the ranking function, and δ by the lower bound on the amount of decrease. Then, we obtain the following defining constraint on the ranking function coefficients and the bound values.

$$\begin{aligned} &\exists f_x\ \exists f_y\ \exists \delta_0\ \exists \delta \\ &\quad \forall x\ \forall y\ \forall x'\ \forall y' : \\ &\qquad (\delta \geq 1 \wedge \\ &\qquad \rho_2 \rightarrow (f_x x + f_y y \geq \delta_0 \wedge \\ &\qquad\qquad f_x x' + f_y y' \leq f_x x + f_y y - \delta)) \end{aligned} \tag{1}$$

Any satisfying assignment to f_x, f_y, δ_0 and δ determines a linear ranking function for the loop.

The constraint (1) contains universal quantification over the program variables and their primed version, which makes it difficult to solve directly using existing constraint solvers. At the next step, we will address this obstacle by eliminating the universal quantification.

Constraints. First, we represent the transition relation of the loop in matrix form, which will help us during the constraint generation. After replacing equalities by conjunctions of corresponding inequalities, we obtain the matrix form below.

$$\begin{aligned} \rho_2 &= (x + 1 \leq y \wedge x' = x + 1 \wedge y' = y) \\ &= (x - y \leq -1 \wedge -x + x' \leq 1 \wedge x - x' \leq -1 \wedge -y + y' \leq 0 \wedge y - y' \leq 0) \\ &= \begin{pmatrix} 1 & -1 & 0 & 0 \\ -1 & 0 & 1 & 0 \\ 1 & 0 & -1 & 0 \\ 0 & -1 & 0 & 1 \\ 0 & 1 & 0 & -1 \end{pmatrix} \begin{pmatrix} x \\ y \\ x' \\ y' \end{pmatrix} \leq \begin{pmatrix} -1 \\ 1 \\ -1 \\ 0 \\ 0 \end{pmatrix} \end{aligned}$$

The bound and decrease conditions from (1) produce the following matrix forms.

$$f_x x + f_y y \geq \delta_0 = \begin{pmatrix} -f_x & -f_y & 0 & 0 \end{pmatrix} \begin{pmatrix} x \\ y \\ x' \\ y' \end{pmatrix} \leq -\delta_0$$

$$f_x x' + f_y y' \leq f_x x + f_y y - \delta = \begin{pmatrix} -f_x & -f_y & f_x & f_y \end{pmatrix} \begin{pmatrix} x \\ y \\ x' \\ y' \end{pmatrix} \leq -\delta$$

Now we are ready to eliminate the universal quantification. For this purpose we apply Farkas' lemma, which formally states

$$((\exists x : Ax \leq b) \wedge (\forall x : Ax \leq b \rightarrow cx \leq \gamma)) \leftrightarrow (\exists \lambda : \lambda \geq 0 \wedge \lambda A = c \wedge \lambda b \leq \gamma) \ .$$

This statement asserts that every linear consequence of a satisfiable set of linear inequalities can be obtained as a non-negative linear combination of these inequalities. As an immediate consequence we obtain that for a non-satisfiable set of linear inequalities we can derive an unsatisfiable inequality, i.e.,

$$(\forall x : \neg(Ax \leq b)) \leftrightarrow (\exists \lambda : \lambda \geq 0 \wedge \lambda A = 0 \wedge \lambda b \leq -1) \ .$$

By applying Farkas' lemma on (1) we obtain the following constraint.

$$\begin{array}{l} \exists f_x \ \exists f_y \ \exists \delta_0 \ \exists \delta \\ \quad \exists \lambda \ \exists \mu : \\ \qquad (\delta \geq 1 \wedge \\ \qquad \lambda \geq 0 \wedge \\ \qquad \mu \geq 0 \wedge \\ \qquad \lambda \begin{pmatrix} 1 & -1 & 0 & 0 \\ -1 & 0 & 1 & 0 \\ 1 & 0 & -1 & 0 \\ 0 & -1 & 0 & 1 \\ 0 & 1 & 0 & -1 \end{pmatrix} = \begin{pmatrix} -f_x & -f_y & 0 & 0 \end{pmatrix} \wedge \lambda \begin{pmatrix} -1 \\ 1 \\ -1 \\ 0 \\ 0 \end{pmatrix} \leq -\delta_0 \wedge \\ \qquad \mu \begin{pmatrix} 1 & -1 & 0 & 0 \\ -1 & 0 & 1 & 0 \\ 1 & 0 & -1 & 0 \\ 0 & -1 & 0 & 1 \\ 0 & 1 & 0 & -1 \end{pmatrix} = \begin{pmatrix} -f_x & -f_y & f_x & f_y \end{pmatrix} \wedge \mu \begin{pmatrix} -1 \\ 1 \\ -1 \\ 0 \\ 0 \end{pmatrix} \leq -\delta \end{array} \tag{2}$$

This constraint contains only existentially quantified rational variables and consists of linear (in)equalities. Thus, it can be efficiently solved by the existing tools for Linear Programming over rationals.

Solution. We apply a linear constraint solver on (2) and obtain the following solution.

$$\begin{aligned}\lambda &= (1\ 0\ 0\ 0\ 0)\\ \mu &= (0\ 0\ 1\ 1\ 0)\\ f_x &= -1\\ f_y &= 1\\ \delta_0 &= 1\\ \delta &= 1\end{aligned}$$

This solution states that the expression $-x + y$ decreases during each iteration of the loop by at least 1, and is greater than 1 for all states that satisfy the loop guard.

3 Constraint Linear Interpolants

Interpolants are logical assertions over program states that can separate program states that satisfy a desired property from the ones that violate the property. Interpolants play an important role in automated abstraction of sets of program states and their automatic construction is a crucial building block for program verification tools. In this section we present an algorithm for the computation of linear interpolants. A unique feature of our algorithm is the possibility to bias the outcome using additional constraints.

In program verification, interpolants are computed for formulas that are extracted from program paths, i.e., sequences of program statements that follow the control flow graph of the program. We illustrate the interpolant computation algorithm using a program path from Figure 1, and refer to [7] for a detailed description of the algorithm and a discussion of the related work.

Input. We consider a path $\tau_1\tau_3\tau_5$, which corresponds to an execution of the program that does not enter the loop and fails the assert statement. This path does not modify the values of the program variables, but rather imposes a sequence of conditions $y \geq z \wedge x \geq y \wedge x + 1 \leq z$. Since this sequence is not satisfiable, a program verifier can issue an interpolation query that needs to compute a separation between the states that the program reaches after taking the transition τ_3 and the states that violate the assertion. Formally, we are interested in an inequality $i_x x + i_y y + i_z z \leq i_0$, called an interpolant, such that

$$\begin{aligned}&\exists i_x\ \exists i_y\ \exists i_z\ \exists i_0\\ &\quad \forall x\ \forall y\ \forall z\ :\\ &\qquad ((y \geq z \wedge x \geq y) \rightarrow i_x x + i_y y + i_z z \leq i_0) \wedge\\ &\qquad ((i_x x + i_y y + i_z z \leq i_0 \wedge x + 1 \leq z) \rightarrow 0 \leq -1)\end{aligned} \tag{3}$$

Furthermore, we require that $i_x x + i_y y + i_z z \leq i_0$ only refers to the variables that appear both in $y \geq z \wedge x \geq y$ and $x + 1 \leq z$, which are x and z. Hence, i_z needs to be equal to 0, which is ensured by the above constraint without any additional effort.

Constraints. First we represent the sequence of conditions in matrix form as follows.

$$
\begin{aligned}
&(y \geq z \wedge x \geq y \wedge x + 1 \leq z) = \\
&(-y + z \leq 0 \wedge -x + y \leq 0 \wedge x - z \leq -1) = \\
&\begin{pmatrix} 0 & -1 & 1 \\ -1 & 1 & 0 \\ 1 & 0 & -1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} \leq \begin{pmatrix} 0 \\ 0 \\ -1 \end{pmatrix}
\end{aligned}
$$

Since (3) contains universal quantification, we apply Farkas' to enable applicability of Linear Programming tools and obtain the following constraint.

$$
\begin{aligned}
&\exists i_x\ \exists i_y\ \exists i_z\ \exists i_0 \\
&\quad \exists \lambda\ \exists \mu : \\
&\qquad \lambda \geq 0 \wedge \mu \geq 0 \wedge \\
&\qquad (\lambda\ \mu) \begin{pmatrix} 0 & -1 & 1 \\ -1 & 1 & 0 \\ 1 & 0 & -1 \end{pmatrix} = 0 \wedge (\lambda\ \mu) \begin{pmatrix} 0 \\ 0 \\ -1 \end{pmatrix} \leq -1 \wedge \\
&\qquad (i_x\ i_y\ i_z) = \lambda \begin{pmatrix} 0 & -1 & 1 \\ -1 & 1 & 0 \end{pmatrix} \wedge i_0 = \lambda \begin{pmatrix} 0 \\ 0 \end{pmatrix}
\end{aligned}
\tag{4}
$$

This constraint uses two vectors λ and μ to represent the linear combination that derives the unsatisfiable inequality $0 \leq -1$. The vector λ tracks the first two inequalities, and μ tracks the third inequality.

Solution. By solving (4) we obtain

$$
\begin{aligned}
\lambda &= (1\ 1), \\
\mu &= 1\ , \\
i_x &= -1\ , \\
i_y &= 0\ , \\
i_z &= 1\ , \\
i_0 &= 0\ .
\end{aligned}
$$

The resulting interpolant is $-x + z \leq 0$.

The constraint-based approach to interpolant computation offers a unique opportunity to bias the resulting interpolant using additional constraints. That is, (4) can be extended with an additional constraint $C(\begin{smallmatrix} i \\ i_0 \end{smallmatrix}) \leq c$ that encode the bias condition.

4 Linear Invariants

Invariants are assertions over program variables whose value does not change during program execution. In program verification invariants are used to describe sets of reachable program states, and are an indispensable tool for reasoning about program correctness. In this section, we show how invariants proving the non-reachability of the error location in the program can be computed by using constraint-based techniques, and present a testing-based approach for simplifying the resulting constraint generation task. Furthermore, we briefly present

a close connection between invariant and bound generation. See [4,2] for the corresponding algorithms and further details.

We illustrate the invariant generation on the program shown in Figure 1 and construct an invariant that proves the non-reachability of the location ℓ_5, which serves as the error location.

Input. Our goal is to compute two linear inequalities over program variables $p_x x + p_y y + p_z z \leq p_0$ and $q_x x + q_y y + q_z z \leq q_0$ for the locations ℓ_2 and ℓ_3, respectively, such that these inequalities (1) represent all program states that are reachable at the respective locations, (2) serve as an induction hypothesis for proving (1) by induction over the number of program steps required to reach a program state, and (3) imply that no program execution can reach the error location ℓ_5. We encode the conditions (1–3) on the unknown invariant coefficients as the following constraint.

$$\begin{aligned}
&\exists p_x\ \exists p_y\ \exists p_z\ \exists p_0\ \exists q_x\ \exists q_y\ \exists q_z\ \exists q_0\\
&\quad \forall x\ \forall y\ \forall z\ \forall x'\ \forall y'\ \forall z' :\\
&\qquad (\rho_1 \rightarrow p_x x' + p_y y' + p_z z' \leq p_0) \wedge\\
&\qquad ((p_x x + p_y y + p_z z \leq p_0 \wedge \rho_2) \rightarrow p_x x' + p_y y' + p_z z' \leq p_0) \wedge\\
&\qquad ((p_x x + p_y y + p_z z \leq p_0 \wedge \rho_3) \rightarrow q_x x' + q_y y' + q_z z' \leq q_0) \wedge\\
&\qquad ((q_x x + q_y y + q_z z \leq p_0 \wedge \rho_4) \rightarrow 0 \leq 0) \wedge\\
&\qquad ((q_x x + q_y y + q_z z \leq p_0 \wedge \rho_5) \rightarrow 0 \leq -1)
\end{aligned} \tag{5}$$

For each program transition this constraint contains a corresponding conjunct. The conjunct ensures that given a set of states at the start location of the transition all states reachable by applying the transition are represented by the assertion associated with the destination location. For example, the first conjunct asserts that applying τ_1 on any state leads to a state represented by $p_x x + p_y y + p_z z \leq p_0$.

Constraints. Since (5) contains universal quantification, we resort to the Farkas' lemma-based elimination, which yields the following constraint.

$$\begin{aligned}
&\exists p_x\ \exists p_y\ \exists p_z\ \exists p_0\ \exists q_x\ \exists q_y\ \exists q_z\ \exists q_0\\
&\quad \exists \lambda_1\ \exists \lambda_2\ \exists \lambda_3\ \exists \lambda_4\ \exists \lambda_5 :\\
&\qquad \lambda_1 \geq 0 \wedge \cdots \wedge \lambda_5 \geq 0 \wedge\\
&\qquad \lambda_1 R_1 = (0\ p_x\ p_y\ p_z) \wedge \lambda_1 r_1 \leq p_0 \wedge\\
&\qquad \lambda_2 \begin{pmatrix} p_x\ p_y\ p_z\ 0 \\ R_2 \end{pmatrix} = (0\ p_x\ p_y\ p_z) \wedge \lambda_2 \begin{pmatrix} p_0 \\ r_2 \end{pmatrix} \leq p_0 \wedge\\
&\qquad \lambda_3 \begin{pmatrix} p_x\ p_y\ p_z\ 0 \\ R_3 \end{pmatrix} = (0\ q_x\ q_y\ q_z) \wedge \lambda_3 \begin{pmatrix} p_0 \\ r_3 \end{pmatrix} \leq q_0 \wedge\\
&\qquad \lambda_4 \begin{pmatrix} q_x\ q_y\ q_z\ 0 \\ R_4 \end{pmatrix} = 0 \wedge \lambda_4 \begin{pmatrix} q_0 \\ r_4 \end{pmatrix} \leq 0 \wedge\\
&\qquad \lambda_5 \begin{pmatrix} q_x\ q_y\ q_z\ 0 \\ R_5 \end{pmatrix} = 0 \wedge \lambda_5 \begin{pmatrix} q_0 \\ r_5 \end{pmatrix} \leq -1
\end{aligned} \tag{6}$$

Unfortunately, this constraint is non-linear since it contains multiplication between unknown components of $\lambda_1, \ldots, \lambda_5$ and the unknown coefficients $p_x, p_y, p_z, p_0, q_x, q_y, q_z, q_0$.

Solution. In contrast to interpolation or ranking function generation, we cannot directly apply Linear Programming tools to solve (6) and need to introduce additional solving steps, as described in Section 4.1 and [4]. These steps lead to the significant reduction of the number of non-linear terms, and make the constraints amenable to solving using case analysis on the remaining unknown coefficients for derivations.

For our program we obtain the following solution.

$$\begin{array}{cccc} & \lambda_1 = (1\ 1\ 1\ 1) & & \\ & \lambda_2 = (1\ 0\ 1\ 1\ 1) & & \\ & \lambda_3 = (1\ 1\ 1\ 1\ 1) & & \\ & \lambda_4 = (0\ 0\ 0\ 0\ 0) & & \\ & \lambda_5 = (1\ 1\ 0\ 0\ 0) & & \\ p_x = 0 & p_y = -1 & p_z = 1 & p_0 = 0 \\ q_x = -1 & q_y = 0 & q_z = 1 & q_0 = 0 \end{array}$$

This solution defines an invariant $-y + x \leq 0$ at the location ℓ_2 and $-x + z \leq 0$ at the location ℓ_3.

4.1 Static and Dynamic Constraint Simplification

Now we show how program test cases can be used to obtain additional constraint simplification when computing invariants.

We use the program in Figure 1 and consider a set of program states below, which can be recorded during a test run of the program.

$$\begin{aligned} s_1 &= (\ell_1, x = 1, y = 0, z = 2) \\ s_2 &= (\ell_2, x = 2, y = 0, z = 2) \\ s_3 &= (\ell_2, x = 2, y = 1, z = 2) \\ s_4 &= (\ell_2, x = 2, y = 1, z = 2) \\ s_5 &= (\ell_2, x = 2, y = 2, z = 2) \\ s_6 &= (\ell_3, x = 3, y = 2, z = 2) \end{aligned}$$

These states are reachable, hence any program invariant holds for these states. Hence, we perform a partial evaluation of the unknown invariant templates at locations ℓ_2 and ℓ_3 on states $s_2, \ldots, s_5$ and s_6, respectively:

$$\begin{aligned} \varphi_1 &= (p_x 1 + p_y 2 + p_z 1 \leq p_0) \\ \varphi_2 &= (p_x 1 + p_y 2 + p_z 1 \leq p_0) \\ \varphi_3 &= (p_x 2 + p_y 2 + p_z 1 \leq p_0) \\ \varphi_4 &= (q_x 2 + q_y 2 + q_z 1 \leq q_0) \end{aligned}$$

The obtained constraints are linear and they must hold for any template instantiation. Hence the constraint

$$p_x + 2p_y + p_z \leq p_0 \wedge p_x + 2p_y + p_z \leq p_0 \wedge 2p_x + 2p_y + p_z \leq p_0 \wedge \\ 2q_x + 2q_y + q_z \leq q_0$$

can be added to (6) as an additional strengthening without changing the set of solutions. Practically however this strengthening results in a series of simplifications of the non-linear parts of (6), which can dramatically increase the constraint solving efficiency.

4.2 Bound Generation as Unknown Assertion

Program execution can consume various resources, e.g., memory or time. For our example program in Figure 1, the number of loop iterations might be such a resource since it correlates with the program execution time. Resource bounds are logical assertions that provide an estimate on the resource consumption, and their automatic generation is an important task, esp. for program execution environments with limited resource availability.

There is a close connection between expressions describing resource bounds and program assertions specifying conditions on reachable program states. We can encode the check if a given bound holds for all program execution as a program assertion over auxiliary program variables that keep track of the resource consumption. In our example the assertion statement ensures that the consumption of time, as tracked by the variable x, is bounded from above by the value of the variable z.

Unknown resource bounds can be synthesized using our constraint-based invariant generation algorithm described above after a minor modification of the employed constraint encoding. Next we show how to modify our constraints (5) and (6) to identify a bound on the number of loop iterations, under the assumption that the assertion statement is not present in the program.

First, we assume that the unknown bound assertion is represented by an inequality

$$x \leq b_y y + b_z z + b_0 \ .$$

Now, our goal is to identify the values of the coefficients b_y, b_z, and b_0 together with an invariant that proves the validity of the bound.

We encode our goal as a constraint by replacing the last conjunct in (5), which was present due to the assertion statement, with the following implication.

$$q_x x + q_y y + q_z z \leq q_0 \rightarrow x \leq b_y \leq b_y y + b_z z + b_0$$

This implication requires that the program invariant at the location after the loop exit implies the bound validity.

After eliminating universal quantification from the modified constraint and a subsequent solving attempt we realize that no bound on x can be found. If we consider a modified program that includes an assume statement

```
assume(z>=x);
```

as its first instruction and reflect the modification in the constraints, then we will be able to compute the following bound.

$$x \leq y$$

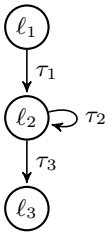
5 Recurrence Sets

Inherent limitations of the existing tools for proving program termination can lead to cases when non-conclusive results are reported. Since a failure to find a termination argument does not directly imply that the program does not terminate on certain inputs, we need dedicated methods that can prove non-termination of programs. In this section we present such a method. It is based on the notion of recurrence set that serves as a proof for the existence of a non-terminating program execution.

Input. We show how non-termination can be proved by constructing recurrence sets using the example in Figure 2. The complete version of the corresponding algorithm is presented in [3].

```
main(int x, int y, int z) {
  assume(y >= z);
  while (x < y) {
    x=x+1+z;
  }
}
```

(a)



(b)

$$\rho_1 = (y \geq z \wedge x' = x \wedge y' = y \wedge z' = z)$$
$$\rho_2 = (x + 1 \leq y \wedge x' = x + 1 + z \wedge y' = y \wedge z' = z)$$
$$\rho_3 = (x \geq y \wedge x' = x \wedge y' = y \wedge z' = z)$$

(c)

Fig. 2. A non-terminating example program (a), its control-flow graph (b), and the corresponding transition relations (c)

To prove non-termination we will compute a recurrence set consisting of program states that can be reached at the loop entry and lead to an additional loop iteration. We assume that a desired recurrence set can be expressed by a conjunction of two inequalities $pv \leq p_0 \wedge qv \leq q_0$ over the vector of program variables v consisting of x, y, and z, while p, p_0, q, and q_0 are unknown coefficients. To simplify notation, we write $Sv \leq s$ for the conjunction of $pv \leq p_0$ and $qv \leq q_0$. Then, the following constraint encodes the recurrence set condition.

$$\begin{aligned}
&\exists S\ \exists s: \\
&\quad (\exists v : Sv \leq s) \wedge \\
&\quad (\exists v\ \exists v' : \rho_1(v, v') \wedge Sv' \leq s) \wedge \\
&\quad (\forall v\ \exists v' : Sv \leq s \rightarrow (\rho_2(v, v') \wedge Sv' \leq s))
\end{aligned} \tag{7}$$

The first conjunct guarantees that the recurrence set is not empty. The second conjunct requires that the recurrence set contains at least one state that is reachable by following the transition τ_1, i.e., by when the loop is reached for the first time. The last conjunct guarantees that every state in the recurrence set can follow the loop transition τ_2 and get back to the recurrence set. Together, these properties guarantee that there exists an infinite program execution that can be constructed from the elements of the recurrence set.

Constraints. The constraint (7) contains universal quantification and quantifier alternation, which makes it difficult to solve using the existing quantifier elimination tools. As the first step, we simplify (7) by exploiting the structure of transition relations ρ_1 and ρ_2. Our simplification relies on the fact that the values of primed variables are defined by update expressions, and substitutes the primed variables by the corresponding update expressions. We obtain the following simplified equivalent of (7).

$$\begin{aligned}
&\exists S\ \exists s: \\
&\quad (\exists x\ \exists y\ \exists z : S \begin{pmatrix} x \\ y \\ z \end{pmatrix} \leq s) \wedge \\
&\quad (\exists x\ \exists y\ \exists z : y \geq z \wedge S \begin{pmatrix} x \\ y \\ z \end{pmatrix} \leq s) \wedge \\
&\quad (\forall x\ \forall y\ \forall z : S \begin{pmatrix} x \\ y \\ z \end{pmatrix} \leq s \rightarrow (x + 1 \leq y \wedge S \begin{pmatrix} x+1+z \\ y \\ z \end{pmatrix} \leq s))
\end{aligned}$$

Furthermore, we will omit the first conjunct since it is subsumed by the second conjunct.

Next, we eliminate universal quantification by applying Farkas' lemma. The application yields the following constraint. It only contains existential quantification and uses S_x, S_y, and S_z to refer to the first, second, and the third column of S, respectively.

$$\begin{aligned}
&\exists S\ \exists s: \\
&\quad (\exists x\ \exists y\ \exists z : S \begin{pmatrix} x \\ y \\ z \end{pmatrix} \leq s) \wedge \\
&\quad (\exists x\ \exists y\ \exists z : y \geq z \wedge S \begin{pmatrix} x \\ y \\ z \end{pmatrix} \leq s) \wedge \\
&\quad (\exists \lambda : \lambda \geq 0 \wedge \lambda S = \begin{pmatrix} 1 & -1 & 0 \end{pmatrix} \wedge \lambda s \leq -1) \wedge \\
&\quad (\exists \Lambda : \Lambda \geq 0 \wedge \Lambda S = \begin{pmatrix} S_x & S_y & S_z + S_x \end{pmatrix} \wedge \Lambda s \leq (s - S_x))
\end{aligned} \tag{8}$$

Solution. We apply solving techniques that we used for dealing with non-linear constraints during invariant generation, see Section 4, and obtain the following solution.

$$\begin{aligned} x &= -2 \\ y &= -1 \\ z &= -1 \\ \lambda &= (1\ 0) \\ \Lambda &= \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} \\ p &= (1\ -1\ 0) \\ p_0 &= -1 \\ q &= (0\ 0\ 1) \\ q_0 &= -1 \end{aligned}$$

This solution defines the recurrence set

$$x - y \leq -1 \wedge z \leq -1 \ ,$$

and states that the program does not terminate if executed from an initial state that assigns $x = -2$, $y = -1$, and $z = -1$.

6 Combination with Uninterpreted Functions

In the previous sections we showed how auxiliary assertions represented by linear inequalities can be generated using constraint-based techniques. In this section we show that these techniques can be directly extended to deal with assertions represented by linear arithmetic combined with uninterpreted functions. This combined theory plays in important role in program verification, where uninterpreted functions are used to abstract functions that too complex to be modeled precisely. The basis of the extension is the hierarchical approach to the combination of logical theories [6]. We refer to [7,1] for constraint-based interpolation and invariant generation algorithms for the combination of linear arithmetic and uninterpreted functions. Next, we will illustrate the interpolation algorithm for linear arithmetic and function symbols using a small example.

Input. The interpolation algorithm takes as input a pair of mutually unsatisfiable assertions φ and ψ shown below.

$$\begin{aligned} \varphi &= (x \leq a \wedge a \leq y \wedge f(a) \leq 0) \\ \psi &= (y \leq b \wedge b \leq x \wedge 1 \leq f(b)) \end{aligned}$$

The proof of unsatisfiability requires reasoning about linear arithmetic and uninterpreted function, which we represent by the logical consequence relation $\models_{\text{LI+UIF}}$.

$$\varphi \wedge \psi \models_{\text{LI+UIF}} \bot$$

The goal of the interpolation algorithm is to construct an assertion χ such that

$$\begin{aligned} &\varphi \models_{\text{LI+UIF}} \chi \ , \\ &\chi \wedge \psi \models_{\text{LI+UIF}} \bot \ , \\ &\chi \text{ is expressed over common symbols of } \varphi \text{ and } \psi \ . \end{aligned} \tag{9}$$

Constraints and solution. As common in reasoning about combined theories, we first apply a purification step that separates arithmetic constraints from the function applications as follows.

$$\begin{aligned} \varphi_{\text{LI}} = \ & (x \leq a \wedge a \leq y \wedge c \leq 0) \\ \psi_{\text{LI}} = \ & (y \leq b \wedge b \leq x \wedge 1 \leq d) \\ D = \ & \{c \mapsto f(a), d \mapsto f(b)\} \\ X = \ & \{a = b \rightarrow c = d\} \end{aligned}$$

The sets of inequalities φ_{LI} and ψ_{LI} do not have any function symbols, which were replaced by fresh variables. The mapping between these fresh variables and the corresponding function applications is give by the set D. The set X contains functionality axiom instances that we create for all pairs of occurrences of function applications. These instances are expressed in linear arithmetic. For our example there is only one such instance.

The hierarchical reasoning approach guarantees that instances collected in X are sufficient for proving the mutual unsatisfiability of the pure assertions φ_{LI} and ψ_{LI}, i.e.,

$$\varphi_{\text{LI}} \wedge \psi_{\text{LI}} \wedge \bigwedge X \models_{\text{LI}} \bot$$

Unfortunately we cannot apply an algorithm for interpolation in linear arithmetic on the unsatisfiable conjunction presented above since the axiom instance in X contains variables that appear both in φ_{LI} and ψ_{LI}, which will lead to an interpolation result that violates the third condition in 9.

Instead, we resort to a case-based reasoning as follows. First, we attempt to compute an interpolant by considering the pure assertions, but do not succeed since they are mutually satisfiable, i.e.,

$$\varphi_{\text{LI}} \wedge \psi_{\text{LI}} \not\models_{\text{LI}} \bot$$

Nevertheless, the conjunction of pure assertions implies the precondition for applying the functionality axiom instance from X, i.e.,

$$\varphi_{\text{LI}} \wedge \psi_{\text{LI}} \models_{\text{LI}} a = b$$

From this implication follows that we can compute intermediate terms that are represented over variables that are common to φ_{LI} and ψ_{LI}. Formally, we have

$$\begin{aligned} &\varphi_{\text{LI}} \wedge \psi_{\text{LI}} \models_{\text{LI}} a \leq y \wedge y \leq b \ , \\ &\varphi_{\text{LI}} \wedge \psi_{\text{LI}} \models_{\text{LI}} a \geq x \wedge x \geq b \ . \end{aligned}$$

We rearrange these implications and obtain the following implications.

$$\varphi_{\mathrm{LI}} \models_{\mathrm{LI}} x \leq a \wedge a \leq y$$
$$\psi_{\mathrm{LI}} \models_{\mathrm{LI}} y \leq b \wedge b \leq x$$

These implications are used by our interpolation algorithm to derive appropriate case reasoning, which will be presented later on. Furthermore, our algorithm creates an additional function application $f(y)$ together with a corresponding fresh variable e, which is used for the purification and is recorded in the set D.

$$D \;=\; \{c \mapsto f(a), d \mapsto f(b), e \mapsto f(y)\}$$

The first step of the case reasoning requires computing an interpolant for the following unsatisfiable conjunction.

$$(\varphi_{\mathrm{LI}} \wedge a = e) \wedge (\psi_{\mathrm{LI}} \wedge e = b) \models_{\mathrm{LI}} \bot$$

By applying the algorithm presented in Section 3 we obtain a partial interpolant $e \leq 0$ such that

$$\varphi_{\mathrm{LI}} \wedge a = e \models_{\mathrm{LI}} e \leq 0 \;,$$
$$e \leq 0 \wedge \psi_{\mathrm{LI}} \wedge e = b \models_{\mathrm{LI}} \bot \;.$$

The partial interpolant is completed using the case reasoning information as follows.

$$\chi_{\mathrm{LI}} \;=\; (x \neq y \vee (x = y \wedge e \leq 0))$$

After replacing the fresh variables by the corresponding function applications we obtain the following interpolant χ for the original input φ and ψ.

$$\begin{aligned}\chi = \;& (x \neq y \vee (x = y \wedge e \leq 0))[f(q)/e] \\ =\; & x \neq y \vee (x = y \wedge f(q) \leq 0)\end{aligned}$$

7 Conclusion

We presented a collection of examples demonstrating that several kinds of auxiliary assertions that play a crucial role in program verification can be effectively synthesized using constraint-based techniques.

Acknowledgment. I thank Byron Cook, Fritz Eisenbrand, Ashutosh Gupta, Tom Henzinger, Rupak Majumdar, Andreas Podelski, and Viorica Sofronie-Stokkermans for unconstrained satisfactory discussions.

References

1. Beyer, D., Henzinger, T.A., Majumdar, R., Rybalchenko, A.: Invariant synthesis for combined theories. In: Cook, B., Podelski, A. (eds.) VMCAI 2007. LNCS, vol. 4349, pp. 378–394. Springer, Heidelberg (2007)
2. Cook, B., Gupta, A., Magill, S., Rybalchenko, A., Simsa, J., Singh, S., Vafeiadis, V.: Finding heap-bounds for hardware synthesis. In: FMCAD (2009)

3. Gupta, A., Henzinger, T.A., Majumdar, R., Rybalchenko, A., Xu, R.-G.: Proving non-termination. In: POPL (2008)
4. Gupta, A., Majumdar, R., Rybalchenko, A.: From tests to proofs. In: TACAS (2009)
5. Podelski, A., Rybalchenko, A.: A complete method for the synthesis of linear ranking functions. In: Steffen, B., Levi, G. (eds.) VMCAI 2004. LNCS, vol. 2937, pp. 239–251. Springer, Heidelberg (2004)
6. Sofronie-Stokkermans, V.: Hierarchic reasoning in local theory extensions. In: Nieuwenhuis, R. (ed.) CADE 2005. LNCS (LNAI), vol. 3632, pp. 219–234. Springer, Heidelberg (2005)
7. Sofronie-Stokkermans, V., Rybalchenko, A.: Constraint solving for interpolation. J. of Symbolic Computation (to appear, 2010)