Hoarescope

Markus Triska (0225855)[1]

July 17, 2005

Abstract

This paper presents *Hoarescope*, a program that helps to prove partial correctness assertions of $AL(\mathcal{N})$-programs using Hoare calculus. This paper also presents *Presprover*, a program for proving satisfiability and validity of formulas of Presburger arithmetic.

1 Introduction

It is assumed that the reader is familiar with Hoare calculus. A thorough overview is provided in [Apt 1981] and the references included therein. Since *Hoarescope* is primarily intended as a teaching aid for students who take the "Theoretische Informatik 1" course at the Vienna University of Technology, we briefly outline the inference rules as stated in the course material, because they are referred to in the output of *Hoarescope*. Though we don't formally define neither syntax nor semantics of $AL(\mathcal{N})$ (Assignment Language over natural numbers) in this paper, both can be intuitively grasped to a large extent from the provided examples. The main rules named (H1) to (H4) are:

$$\frac{(P \supset Q[^t_v])}{P\ \{v \leftarrow t\}\ Q}\ (H1)$$

(variables of $v \leftarrow t$ must not be quantified in P or Q)

$$\frac{P\ \{\alpha\}\ Q \quad Q\ \{\beta\}\ R}{P\ \{\text{begin } \alpha; \beta \text{ end}\}\ R}\ (H2)$$

$$\frac{(P \wedge B)\ \{\alpha\}\ Q \quad (P \wedge \neg B)\ \{\beta\}\ Q}{P\ \{\text{if } B \text{ then } \alpha \text{ else } \beta\}\ Q}\ (H3)$$

$$\frac{(P \supset INV) \quad (INV \wedge B)\ \{\alpha\}\ INV \quad ((INV \wedge \neg B) \subset Q)}{P\ \{\text{while } B \text{ do } \alpha\}\ Q}\ (H4)$$

Additionally, we introduce auxiliary rules (T1) to (T3) that formalize the choice of a suitable *interpolant* Q when applying (H2) in certain cases:

[1] This project was done as "Wahlfachpraktikum", adviser G. Salzer.

$$\frac{(P \wedge v = t)\ \{\beta\}\ R}{P\ \{\text{begin } v \leftarrow t;\ \beta \text{ end}\}\ R}\ (T1)$$
$$(v \text{ must not occur in } P \text{ or } t)$$

$$\frac{P\ \{\alpha\}\ R[^t_v]}{P\ \{\text{begin } \alpha; v \leftarrow t \text{ end}\};\ R}\ (T2)$$

$$\frac{P\ \{\alpha\}\ INV \quad (INV \wedge B)\ \{\beta\}\ INV \quad ((INV \wedge \neg B) \supset R)}{P\ \{\text{begin } \alpha;\ \text{while } B \text{ do } \beta \text{ end}\}\ R}\ (T3)$$

2 Inference Engine

We present the inference engine of *Hoarescope* in tandem with an example derivation tree that is generated from the partial correctness assertion

$y > 1$ {begin $x \leftarrow 3$; if $x + y < 19$ then $x \leftarrow x + 19$ else $x \leftarrow y + y$ end} $x > y$.

This Hoare triple can be specified using *Hoarescope*'s syntax as follows:

```
y > 1 { begin x := 3; if x + y < 19 then x := x + 19 else    ↩
        x := y + y end } x > y
```

Many other examples are provided in the *Hoarescope* distribution to give a quick overview over syntactic possibilities. In particular, *aliases* for terms can be specified that help to keep the generated derivation tree concise:

```
y > 1 { begin x := 3; if x + y < 19 then x := x + 19 else    ↩
        x := y + y end } x > y WHERE y > 1 IS P,             ↩
        x > y IS Q, x+y<19 IS B
```

From this input, *Hoarescope* generates LaTeX output that, if rendered, looks like this:

$$\frac{\dfrac{\overset{(1)}{(((P \wedge (\text{x} = 3)) \wedge B) \supset (Q[^{(\text{x}+19)}_{\text{x}}]))}}{((P \wedge (\text{x} = 3)) \wedge B)\ \{\text{x} \leftarrow (\text{x}+19)\}\ Q}\ (\text{H1}) \quad \dfrac{\overset{(2)}{(((P \wedge (\text{x} = 3)) \wedge \neg B) \supset (Q[^{(\text{y}+\text{y})}_{\text{x}}]))}}{((P \wedge (\text{x} = 3)) \wedge \neg B)\ \{\text{x} \leftarrow (\text{y}+\text{y})\}\ Q}\ (\text{H1})}{\dfrac{(P \wedge (\text{x} = 3))\ \{\text{if } B \text{ then x} \leftarrow (\text{x}+19) \text{ else x} \leftarrow (\text{y}+\text{y})\}\ Q}{P\ \{\text{begin x} \leftarrow 3;\ \text{if } B \text{ then x} \leftarrow (\text{x}+19) \text{ else x} \leftarrow (\text{y}+\text{y}) \text{ end}\}\ Q}\ (\text{T1})}\ (\text{H3})$$

In addition to this derivation tree, *Hoarescope* emits status messages (that we omit here) from which the derivation process can be observed in a form suitable for further automatic processing, if required. These messages also hint at the formulas that are to be proved in order to complete the derivation (in this case, the formulas labelled (1) and (2) at the top of the tree). The next chapter introduces a means to do this automatically for a large class of formulas.

Hoarescope contains a few built-in heuristics to automatically deduce loop invariants for certain kinds of loops. In case no suitable invariant can be found, a status message is printed and the derivation continues with a symbolic name for the invariant.

3 Proving Formulas of Presburger Arithmetic

The first-order theory of natural numbers with addition, whose language consists of the non-logical constant symbols 0 and 1, the binary relation symbols $=$ and $\leq$, and the binary function symbol $+$, is commonly called *Presburger arithmetic*, named after Mojzesz Presburger. In contrast to general arithmetic, Presburger arithmetic is both consistent and complete, and also decidable. In the following, we present *Presprover*, a program that can (dis)prove satisfiability and validity of Presburger formulas. Though *Presprover* evolved in conjunction with *Hoarescope* to automatically complete certain kinds of derivation trees, both programs can be used independently, and in fact, there exists no hard-wired connection between them.

To decide a Presburger formula's satisfiability and validity, *Presprover* implements a simple and effective method described in [Com2001]. The idea is to associate with each Presburger formula φ a *finite automaton* $\mathcal{A}_\varphi$ that is built so that deciding the satisfiability of φ reduces to deciding the emptiness of $\mathcal{A}_\varphi$, and deciding validity of φ comes down to deciding satisfiability of (in essence) the complement automaton.

There are 2 subtleties involved in building the complement automaton, and we explain them here in detail because they are not mentioned in the cited paper. Consider the automaton associated with the formula $(\exists x)x + y > 3$, of which a deterministic version is depicted in figure 1.

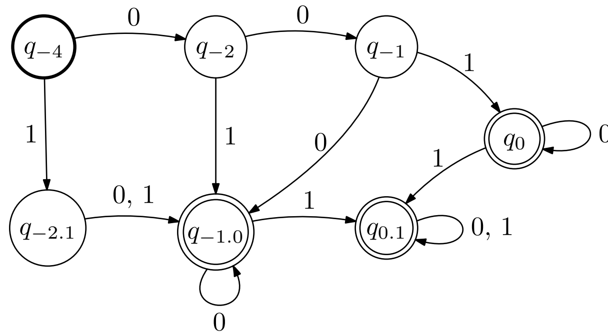

Figure 1: Deterministic automaton corresponding to $(\exists x)x + y > 3$

To build the complement of this deterministic and complete automaton, one would ordinarily turn states q_{-4}, q_{-2}, q_{-1} and $q_{-2.1}$ into accepting states and revoke accepting status from states $q_{-1.0}$, q_0 and $q_{0.1}$. In our case, however, the matter is not so easily settled. We assume that the reader is familiar with the way numbers are represented in the generated automata. From the automaton in figure 1, we see that the sequence "000" leads to an accepting state. As this sequence is one way of representing the number zero, the constant 0 is thus accepted by this automaton (corresponding to the fact that there exists an x such that $x + 0 > 3$). Therefore, the complement automaton must *not* accept any representation of 0, and in particular, it must not accept the sequence "00", and therefore q_{-1} can not be turned into an accepting state. For the same reason, the state q_{-2} can not become a halting state. Moreover, $q_{-2.1}$ can not turn accepting either, as this would imply that both the automaton and its complement accept the constant 1. More generally, an ordinary state from

which a final state can be reached through a path consisting only of zeros is *not* turned into an accepting state when complementing.

The second subtlety has to do with the initial state: If the initial state is an ordinary state, it can – when complementing – only be turned into an accepting state if it is reachable through an actual path (from itself), unless the problem is already reduced to a reachability instance, in which case its accepting status is simply toggled. This is because the empty word does not correspond to a number in the representation used.

The reader will notice that the formulas labelled (1) and (2) at the top of the derivation tree above can be easily rewritten into formulas of Presburger arithmetic using algebraic identities like $y > 1 \equiv 2 \leq y$, and that we should thus be able to use *Presprover* on them. In fact, *Presprover* already contains a simple term rewriting engine and we can type in the formulas virtually verbatim. Here are two example queries that also serve as a quick introduction to the syntax used by *Presprover*:

```
?- valid( y > 1 /\ x = 3 /\ x + y < 19 => x + 19 > y).

Yes

?- valid( y > 1 /\ x = 3 /\ not(x + y < 19) => y + y > y).

Yes
```

Free variables are implicitly all-quantified, and we can make this explicit as in:

```
?- valid(forall(y, exists(x, x + y > 3))).

Yes
```

4 Technicalities

Hoarescope is written in C/C++ and was successfully tested using GCC 3.4.4. The package should build on a variety of systems. *Presprover* was successfully tested with SWI-Prolog 5.5.19 and should also work with any later version. *Presprover* makes use of the finite domain constraint solver that ships with SWI-Prolog.

5 References

[Apt1981] K. R. Apt, *Ten Years of Hoare's Logic: A Survey, Part 1*, ACM Transactions on Programming Languages and Systems (TOPLAS), Volume 3 Issue 4, October 1981

[Com2001] H. Comon and C. Kirchner, *Constraint Solving on Terms*, in *Constraints in Computational Logics*, Lecture Notes in Computer Science, Springer 2001