Verification Condition Generation and Discharge in a Hoare Logic with Recursion

ANDREAS LUNDBLAD

Master's Thesis at NADA
Supervisor: Dilian Gurov
Examiner: Karl Meinke

TRITA xxx yyyy-nn

Abstract

The standard approach to automated Hoare-style program verification is to combine a weakest precondition calculus with a first order logic theorem prover. The calculus either needs explicit loop invariant annotations, or else an additional facility for loop invariant generation.

The alternative scheme we present in this thesis is to use a weakest precondition calculus which produces assertions in a richer language, which does not require loop invariant annotation or generation, but instead a more powerful theorem prover.

The main task of the thesis has been to investigate this approach and to evaluate its effectiveness on concrete examples. Experience from the prototype tool developed show that the approach has a few advantages but requires more research before it can solve any real interesting problems.

Referat

Generering och avslutning av verifikationsvillkor i en Hoare-logik med rekursion

Den vanliga ansatsen till automatiserad, Hoare-baserad, programverifiering är att kombinera en kalkyl för svagaste förvillkor med en teorembevisare för första ordningens logik. Kalkylen behöver antingen explicita annoteringar för loop-invarianter eller någon facilitet för att generera sådana.

Den alternativa ansatsen vi presenterar här använder en kalkyl för svagaste förvillkor som producerar försäkringar i en rikare logik, vilken inte behöver någon annotering eller generering av loop-invarianter, men istället en mer kraftfull teorembevisare.

Huvuduppgiften i denna uppsats har varit att utforska denna ansats och evaluera dess effektivitet på konkreta exempel. Erfarenhet från prototypverktyget visar att ansatsen har några fördelar med behöver mer utforskning för att kunna lösa några riktigt intressanta problem.

Contents

Chapter 1

Introduction

Every day we put ourselves in situations where we entrust computers and software with our lives. The cost of failures in these situations is potentially very high. Since empirical tests of systems can only expose errors, not absence of them, we can benefit from verifying critical parts of these systems (hardware as well as software) formally.

Applying formal methods involves using mathematically rigorous techniques for specification, design and verification. The phrase "mathematically rigorous" means that specifications are expressed in well-formed statements in a mathematical logic and that verification is carried out by using deductive rules for this logic (and thus can be checked by a mechanical process) [1].

Many theories have been developed to facilitate this task. Among the successful ones we find: *Floyd-Hoare Logics*, *Petri nets*, *Hennessy-Milner Logic*, *the Z-notation*, *the B-method*, *Calculus of Communicating Systems and the π-calculus*. Several useful tools have been implemented in aid of applying these theories.

Techniques for automated verification and proof generation generally fall into two categories:

- *Model checking*, in which a system verifies certain properties by means of an exhaustive search of all possible states that a system could enter during its execution.
- *Automated theorem proving*, in which a system attempts to produce a formal proof given a description of the system, a set of logical axioms, and a set of inference rules.

This thesis focuses on the latter one.

Automated theorem proving is one of the most well-developed fields of automated reasoning and concerns the proving of mathematical theorems by a computer program. Depending on the underlying logic, the problem of deciding the validity of a theorem varies from trivial to impossible.

The suggested approach to program verification concerns an extension of Hoare logic based on a logic called modal μ-calculus [2] together with so-called updates [3].

This logic is rich enough to express weakest preconditions of loops without involving the loop invariant. This means that no loop invariant annotation or generation is needed, but instead a more powerful theorem prover. The main task of this thesis has been to investigate this approach by developing a proof system and implementing a specialized theorem prover for it. Experience from the prototype tool shows that this approach has a few advantages but requires more research before it can solve any real interesting problems.

In the next section we briefly go through the preliminaries of the thesis. Section 1.2 summarizes the state of the art in the classical approach. In section 1.3 we specify the working of our approach and the goal of the thesis. Our logic, its accompanying proof system and a few examples is presented in sections 2.2, 2.3 and 2.4. A note on the implementation is brought up in section 3. In section 4 we present our results and conclusions.

1.1 Preliminaries

It is assumed that the reader has knowledge of

- Basic programming
- Discrete math and logic
- Syntax and semantics of programming languages

1.1.1 Transition Systems

A program *state* is a mapping from program *locations* (variable identifiers) to integers. It could be thought of as the content of the memory in a computer. Assignments such as $A := A+1$ alter the current state as a program executes. We will let σ and σ' range over the set of states. If a program is in a state σ, and executes $A := 5$ we write the resulting state as $\sigma[5/A]$. The meaning of this so-called *substitution* is

$$\sigma[a/X](Y) = \begin{cases} a & \text{if } X \equiv Y \\ \sigma(Y) & \text{otherwise} \end{cases}$$

The set of all states is denoted by $\mathbf{\Sigma}$.

A labeled transition system (LTS) induced by a program is a graph that describes how the program state changes during execution. A formal definition follows here.

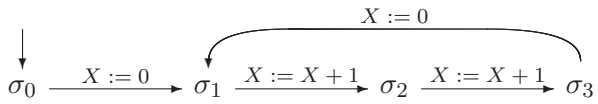
Definition (LTS) An LTS is a labled directed graph composed of three components: A set of states (vertices), S, a set of labels, L, and a transition relation (edges), $\rightarrow$ which is a subset of $S \times L \times S$. □

In our case, S will equal the entire set of states $\mathbf{\Sigma}$, L will equal the set of all assignments ($\mathbf{Loc} \times \mathbf{Aexp}$) and $\rightarrow$ will equal the transitions induced by the rules of the standard small-step operational semantics of the language [4].

As an example we have depicted the LTS for the program

$$
\begin{array}{l}
X := 0; \\
\textbf{while true do} \\
\quad \textbf{if } X = 2 \textbf{ then} \\
\qquad X := 0 \\
\quad \textbf{else} \\
\qquad X := X + 1
\end{array}
$$

in figure 1.1 where the initial state σ_0 has been chosen.



where $\sigma_0 =$ initial state
$\sigma_1 = \sigma_0[0/X]$
$\sigma_2 = \sigma_1[1/X] = \sigma_0[1/X]$
$\sigma_3 = \sigma_2[2/X] = \sigma_0[2/X]$

Figure 1.1. Example of a transition system.

1.1.2 IMP

The toy-language we adopt in this thesis is called *IMP*. All definitions (syntax as well as semantics) are taken directly from [4]. The syntax is presented in BNF notation and the semantics in the denotational style.

We will assume the following notation:

Set	Ranged over by	Description
N	n	Positive and negative integers
Intvar	i	Integer variables
Loc	X, Y	Locations (program variable identifiers)
T	t	The truth values: **tt** and **ff**.

Syntax

IMP programs are built from three different types of expressions: arithmetic expressions, boolean expressions and command expressions. Each type has its own grammatical specification defined in table 1.1.

Description	Name	Grammar
Arithmetic	**Aexp**	$a ::= n \mid X \mid a + a \mid a - a \mid a \times a$
Booleans	**Bexp**	$b ::= \mathbf{tt} \mid \mathbf{ff} \mid a = a \mid a \leq a \mid b \wedge b \mid b \vee b \mid \neg b$
Commands	**Cexp**	$c ::= \mathbf{skip} \mid X := a \mid c; c \mid$ $\mathbf{if}\ b\ \mathbf{then}\ c\ \mathbf{else}\ c \mid \mathbf{while}\ b\ \mathbf{do}\ c$

Table 1.1. Syntactical specification of IMP.

Denotational Semantics

The denotational semantics consists of semantic functions mapping expressions to their mathematical meanings. The meaning of an arithmetic expression in a specific state is an integer, thus the semantical function, $\mathcal{A}$, has the signature:

$$\mathcal{A} : \mathbf{Aexp} \to (\mathbf{\Sigma} \to \mathbf{N})$$

Similarly, the meaning of a boolean expression in a specific state is a boolean value, thus the semantical function for boolean expressions, $\mathcal{B}$, has the following signature:

$$\mathcal{B} : \mathbf{Bexp} \to (\mathbf{\Sigma} \to \mathbf{T})$$

The meaning of a command expression could be seen as a function mapping initial states to final states. However since not all programs terminate, we create an undefined state called $\perp$ and define $\mathbf{\Sigma}_\perp$ as $\mathbf{\Sigma} \cup \{\perp\}$. The semantical function for command expressions, $\mathcal{C}$ then has the following signature:

$$\mathcal{C} : \mathbf{Cexp} \to (\mathbf{\Sigma} \to \mathbf{\Sigma}_\perp)$$

For the definition of these three functions we refer to appendix A.

1.1.3 Hoare Logic

Hoare logic (also known as *Floyd-Hoare logic*) is a formal system developed by C. A. R. Hoare in 1969 [5]. It is used to prove correctness of programs with respect to specifications expressed as pre- and postconditions, given as *assertions* over states.

An assertion is a logical formula with the following syntax:

Description	Name	Grammar
Ext. arithmetic	**Aexpv**	$a' ::= n \mid X \mid i \mid a' + a' \mid a' - a' \mid a' \times a'$
Assertions	**Assn**	$\phi ::= \mathbf{tt} \mid \mathbf{ff} \mid a' = a' \mid a' \leq a' \mid \phi \wedge \phi \mid$ $\phi \vee \phi \mid \neg\phi \mid \forall i.\phi \mid \exists i.\phi$

Its semantics is given by the denotational function

$$\mathcal{A}n : \mathbf{Assn} \to ((\mathbf{Intvar} \to \mathbf{N}) \to 2^{\mathbf{\Sigma}})$$

defined in appendix A.5. A state satisfies an assertion ϕ iff it is in $\mathcal{A}n[\![\phi]\!]$.

When verifying programs there are two types of correctness:

- *Total correctness*

  The initial state satisfies the precondition implies that the program terminates and that its final state satisfies the postcondition.

- *Partial correctness*

  The initial state satisfies the precondition and that the program terminates implies that its final state satisfies the postcondition.

A precondition, a postcondition and a program constitute a *Hoare triple*.

Definition (Hoare Triple) A Hoare triple is written on the form

$$\{A\}P\{B\}$$

where P is a program and A and B are pre- and postconditions respectively. A Hoare triple is valid, denoted $\vDash \{A\}P\{B\}$, in terms of total correctness iff:

$$\sigma \vDash A \wedge \mathcal{C}[\![P]\!]\sigma = \sigma' \quad \text{implies} \quad \sigma' \neq \bot \wedge \sigma' \vDash B$$

and in terms of partial correctness iff:

$$\sigma \vDash A \wedge \mathcal{C}[\![P]\!]\sigma = \sigma' \wedge \sigma' \neq \bot \quad \text{implies} \quad \sigma' \vDash B$$

□

To verify the validity of a Hoare triple, it is common to use a so-called *weakest precondition calculus*, which involves the function

$$wp : (\mathbf{Com}, \mathbf{Assn}) \rightarrow \mathbf{Assn}$$

The function takes as argument a program, P, and a postcondition, B, and returns a formula, A, denoting $\{\sigma \in \mathbf{\Sigma} \mid \mathcal{C}[\![P]\!]\sigma \neq \bot \wedge \mathcal{C}[\![P]\!]\sigma \vDash B\}$. Such a formula does always exists due to the fact that **Assn** is *expressive* [4]. The formula is called the *weakest precondition*. For partial correctness the similar function *wlp* is used instead. The formula returned by *wlp* is called the *weakest liberal precondition*.

A Hoare triple is valid iff the precondition implies the weakest precondition of the program and the postcondition:

$$A \Rightarrow wp(P, B) \qquad \Leftrightarrow \qquad \vDash \{A\}P\{B\} \tag{1.1}$$

A *loop invariant* is an assertion that is true before, during, and after an execution of a loop statement. More precisely: given the Hoare triple $\{A\}$**while** b **do** $c\{B\}$, I is a loop invariant if the following three conditions are met:

1. $A \Rightarrow I$ (the invariant holds before loop entrance)
2. $I \wedge b \Rightarrow wp(c, I)$ (the invariant holds after a loop iteration)
3. $I \wedge \neg b \Rightarrow B$ (the invariant holds after loop termination)

1.1.4 Gentzen's Sequent Calculus

Gentzen's sequent calculus, also known as LK, is a deduction system for first order logic. It consists of *sequents* and a set of inference rules.

Definition (Sequent) A sequent has the form $\Gamma \vdash \Delta$ where both Γ and Δ are sets of formulas. Γ is called the *antecedent* and Δ is called the *succedent*. A sequent is valid if $\bigwedge \Gamma$ implies $\bigvee \Delta$. □

Inference rules will be written on the following form:

$$\frac{\Gamma_0 \vdash \Delta_0 \quad \dots \quad \Gamma_n \vdash \Delta_n}{\Gamma \vdash \Delta}$$

where the upper sequents denotes the premises and the lower sequent, the conclusion.

1.1.5 Fixpoints

Let Z be a propositional variable and V be a valuation function for such variables, that is Z is satisfied by all states in $V(Z)$. Further, we let $\langle - \rangle \phi$ be satisfied by those states that have a successor satisfying ϕ.

For example, let us consider the LTS in figure 1.1. If we let $V(Z) = \{\sigma_1, \sigma_3\}$ then $V(\langle - \rangle Z) = \{\sigma_0, \sigma_2, \sigma_3\}$.

Now consider the recursive equation

$$Z = X \geq 0 \wedge \langle - \rangle Z$$

What set of states could $V(Z)$ be in this case? One answer is $\{\sigma_1, \sigma_2, \sigma_3\}$ since in σ_1, $X \geq 0$ and a successor (σ_2) is in the set. Same reasoning applies for σ_2 and σ_3. Another correct solution would be the empty set. Solutions to equations of this form are called *fixpoints*. Due to the Knaster-Tarski theorem [4], we know that there is always a unique minimal fixpoint and a unique maximal fixpoint. These will be denoted by $\mu Z.\phi$ and $\nu Z.\phi$ respectively where ϕ is the formula $X \geq 0 \wedge \langle - \rangle Z$ in our example above.

Approximants

There is a way of viewing fixpoints as the limit of an iterative construction (which terminates for finite-state systems). This construction algorithm is easiest explained with an example. We continue to study the LTS from above.

Lets say we are interested in the greatest fixpoint of $\phi \equiv X \geq 0 \wedge \langle - \rangle Z$, that is $\nu Z.\phi$. We start off with $\mathbf{tt}$ which is satisfied by all states, $\{\sigma_0, \sigma_1, \sigma_2, \sigma_3\}$. This is our first *approximant* and it is written as $\nu^0 Z.\phi$.

$$\nu^0 Z.\phi = \mathbf{tt}, \quad V(\nu^0 Z.\phi) = \{\sigma_0, \sigma_1, \sigma_2, \sigma_3\}$$

We now calculate the next approximant, $\nu^1 Z.\phi$ by unfolding ϕ once:

$$\nu^1 Z.\phi = \phi[\nu^0 Z.\phi/Z] = X \geq 0 \wedge \langle - \rangle \mathbf{tt}, \quad V(\nu^1 Z.\phi) = \{\sigma_1, \sigma_2, \sigma_3\}$$

When we calculate the third approximant we get

$$\nu^2 Z.\phi = \phi[\nu^1 Z.\phi/Z] = X \geq 0 \wedge \langle - \rangle \langle - \rangle \mathbf{tt}, \quad V(\nu^2 Z.\phi) = \{\sigma_1, \sigma_2, \sigma_3\}$$

which is the same as the second approximant. This means that we have found the greatest solution to the recursive equation.

In case we are interested in the least fixpoint of ϕ we use the same method but we start off with $\mu^0 Z.\phi = \mathbf{ff}$. Unfolding once gives $\mu^1 Z.\phi = \phi[\mu^0 Z.\phi/Z] = \mathbf{ff}$ which is the same as the first approximant, thus the $\emptyset$ is the least solution to the equation.

For further details we refer to the tutorial *Modal logics and μ-calculi: an introduction* by Bradfield and Stirling [2].

1.2 Background

There are many ways of automatically verifying that a program implements a specification. One is the standard Hoare logic approach presented below.

1.2.1 Classical Approach

The classical approach is to express the weakest precondition in first order logic and continue by delegating any proof obligations to a suitable theorem prover, as described in [4].

This approach has two catches:

- The weakest precondition of a while loop involves the loop invariant (see [4] section 6.4), which is rarely available and hard to generate.
- The proof obligations that fall out are not always easy to prove even for modern theorem provers. In fact Blass and Gurevich [6] recently proved that as a consequence of Cook's completeness theorem there exists a single while loop program whose loop invariant is undecidable.

The approach can be summarized with the chart in figure 1.2.

1.2.2 State of the Art

The main problem with the classical approach is the loop invariant generation. We here present a few examples of techniques that are of current interest in loop invariant generation.

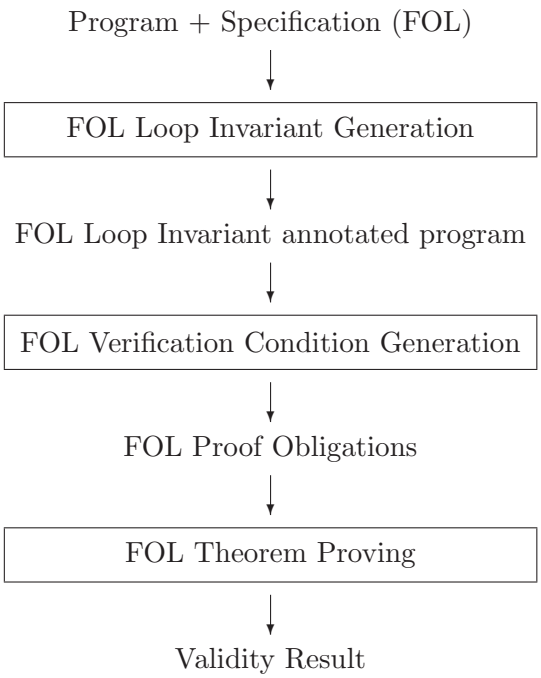


Figure 1.2. Classical approach to Hoare-style verification.

Induction-Iteration

Suzuki and Ishihata originally proposed the *induction-iteration* technique in 1977 for checking array programs, [7]. It is simple and straightforward yet, with some tricks and optimizations, quite efficient.

The method proceeds by searching for the weakest precondition of the loop. The calculation is performed through back-substitution, starting with the postcondition of the loop. The weakest precondition of the loop **while** b **do** c and postcondition B can be expressed as the conjunction of all $W(i)$'s, where $W(i)$ is defined as:

$$W(0) = wlp(c, B)$$
$$W(i+1) = wlp(c, W(i))$$

Luckily the entire (infinite) conjunction is not always needed. When the conjunction is strong enough to meet the conditions mentioned in section 1.1.3 the iteration stops. Further iterations are unnecessary as we are only looking for the weakest precondition.

The major problems with this algorithm is that

- It may never converge.
- Depending on the design of the theorem prover the set of added $W(i)$'s could create an exponentially large set of clauses.

The induction-iteration algorithm was in year 2000 enhanced by Xu, et al [8]. Their work concerned saftey property verification of machine code, which included branches, loops and inner loops. The algorithm worked fairly well on their problem domain. A summary of a few of their tests (performed on a 440 MHz Sun Ultra 10 machine) is show below:

	Array sum	Paging policy	Hash tbl lookup	Bubble sort	Btree traversal	Heap sort	MD5
Instructions:	13	20	25	25	51	95	883
Branches:	2	5	4	5	11	16	11
Loops:	1	2	1	2	2	4	5
Inner loops:	0	1	0	1	1	2	2
Procedure calls:	0	0	1	0	4	0	6
Saftey conditions:	4	9	14	19	42	84	135
Verification time (s):	0.06	0.47	0.39	0.48	0.53	3.67	13.95

A few of the drawbacks they discovered were:

- The method does not naturally extend to working with nested loops.
- To avoid exponential blow-up in the theorem prover, the conjunction has to be simplified as much as possible.
- Compiler optimization might complicate the task of verifying the machine code.

All issues mentioned above might be topics for future research.

Abstract Interpretation

As many other formal methods *abstract interpretations* are useful both in program verification and in compiler design.

An abstract interpretation refers to a way of interpreting the code. As the denotation of the language defines the result of normal computation, an abstract interpretation is a specialized denotation describing only a specific aspect of the computation. The information gathered from analyzing code with an abstract interpretation will not be as informative as a real execution. In fact, it might be inaccurate [9].

An intuitive example is the rule of signs, borrowed from [10]. We consider basic arithmetic on integers and we define a valuation function $\mathcal{V}$ in the following way:

$$\mathcal{V}[\![p+q]\!] = \mathcal{V}[\![p]\!] + \mathcal{V}[\![q]\!] \qquad \mathcal{V}[\![0]\!] = 0, \mathcal{V}[\![1]\!] = 1, \mathcal{V}[\![2]\!] = 2, \ldots$$
$$\mathcal{V}[\![p \times q]\!] = \mathcal{V}[\![p]\!] \times \mathcal{V}[\![q]\!] \qquad \mathcal{V}[\![-1]\!] = -1, \mathcal{V}[\![-2]\!] = -2, \ldots$$

When evaluating the expression $a \times a + b \times b$ when a is 3 and b is -4 we get 25.

$$\begin{aligned} & \mathcal{V}[\![3 \times 3 + -4 \times -4]\!] \\ = \; & \mathcal{V}[\![3 \times 3]\!] + \mathcal{V}[\![-4 \times -4]\!] \\ = \; & \mathcal{V}[\![3]\!] \times \mathcal{V}[\![3]\!] + \mathcal{V}[\![-4]\!] \times \mathcal{V}[\![-4]\!] \\ = \; & 3 \times 3 + -4 \times -4 \\ = \; & 25 \end{aligned}$$

We now define a less informative valuation function $\mathcal{S}$ which only comprehend the signs. Our domain is no longer integers but instead $\{pos, neg, posneg\}$.

$$\begin{array}{ll} \mathcal{S}[\![p+q]\!] = \mathcal{S}[\![p]\!] + \mathcal{S}[\![q]\!] & \mathcal{S}[\![0]\!] = \mathcal{S}[\![1]\!] = \mathcal{S}[\![2]\!] = \ldots = pos \\ \mathcal{S}[\![p \times q]\!] = \mathcal{S}[\![p]\!] \times \mathcal{S}[\![q]\!] & \mathcal{S}[\![-1]\!] = \mathcal{S}[\![-2]\!] = \ldots = neg \\ pos \times pos = pos & neg \times neg = pos \\ pos \times neg = neg & neg + neg = neg \\ pos + neg = posneg & pos + pos = pos \end{array}$$

The sign valuation of the expression mentioned above, $a \times a + b \times b$ looks as follows

$$\begin{aligned} & \mathcal{S}[\![3 \times 3 + -4 \times -4]\!] \\ = \; & \mathcal{S}[\![3 \times 3]\!] + \mathcal{S}[\![-4 \times -4]\!] \\ = \; & \mathcal{S}[\![3]\!] \times \mathcal{S}[\![3]\!] + \mathcal{S}[\![-4]\!] \times \mathcal{S}[\![-4]\!] \\ = \; & pos \times pos + neg \times neg \\ = \; & pos \end{aligned}$$

Thus the expression evaluates to a positive integer. This information is valuable if, for instance, the expression is passed as an argument to a square-root function. This valuation checks for sufficient conditions and fails to prove that the expression $(a \times b) \times (a \times b) - b \times b = pos$ for which a more refined model is required.

Abstract interpretation can be utilized in several ways for generating loop invariants. One way, *Loop invariants on demand* [11], was developed by K. Rustan M. Leino and Francesco Logozzo in 2005. The basic idea is to generate a verification condition and pass this to an automatic theorem prover. The theorem prover will either prove the correctness of the program or produce a set of traces that lead to an error. Instead of just giving up, reporting these traces as errors, an abstract interpreter is invoked on loops along the traces. This might generate a stronger loop invariant that will allow the theorem prover to make more progress toward a proof.

Gröbner Basis

A Gröbner basis is a particular kind of generating subset of an ideal in a polynomial ring. The theory around Gröbner bases has shown to be useful when generating non-linear loop invariants (referred to as inductive assertions). S. Sankaranarayanan, H. B. Sipma and Z. Manna, [12], recently (2004) developed a technique using this theory

for reducing the non-linear invariant generation problem to a numerical constraint-solving problem. In [12] they also discuss solution techniques for these constraints and show that any solution to these constraints is an invariant.

The technique has many advantages. First of all, the degree of the desired invariant does not affect the constraint problem. Secondly, the constraint solving problem can almost always be handled by simple elimination techniques.

A drawback is that the technique it is not complete. This may seem like a harsh requirement but in this case it can sometimes cause the method to miss "obvious" invariants for subtle reasons.

In [12] they do not present any real verification benchmarks but state that they are confident that the technique will scale to larger examples.

Critics and Failed Proof Attempts

One way to improve the search for invariants is to utilize knowledge learned from failed attempts. If the theorem prover fails to prove a subgoal it can remember what went wrong, go back, perform a qualified widening guess and try again. This theorem proving approach has been extensively studied and is generally known as *proof planning*.

To improve this technique further one can use so-called *critics*. Critics are methods for "patching" existing proof attempts. For example, say that a rule has two out of three premises fulfilled, thus is almost applicable. Instead of discarding the attempt by backtracking, a critic may try to fulfill the third premise as well.

Ireland and Stark [13] made a contribution to this technique as they utilized a method called *rippling*. By exploiting syntactical similarities between given formulas and desired formulas rippling guides the theorem prover by suggesting rewrite rules. The three basic rippling strategies are rippling-out, rippling-in and rippling-sideways (see table 1.2).

Strategy	Before	After
out	$f_1(\ldots(f_n(c_1(\ldots)))\ldots)$	$c_n(f_1(\ldots(f_n(\ldots))\ldots))$
sideways	$f_1(c_1(\ldots),\ldots,f_n(\ldots),\ldots)$	$f_1(\ldots,\ldots,c_n(f_n(\ldots)),\ldots)$
in	$c_n(f_1(\ldots f_n(\ldots)\ldots))$	$f_1(\ldots f_n(c_1(\ldots))\ldots)$

Table 1.2. The three basic rippling strategies.

When using the postcondition as starting guess, rippling performs fairly well in most cases. One observation made by Ireland and Stark is that many critics may succeed in guiding the theorem prover. However some critics require less overhead and fewer rewrite rules. What critic to use when could be a topic for future research.

1.3 Problem Statement

Due to standard results on undecidability (as of the Halting Problem) we know that verification in general cannot be fully automated. What *can* be automated, is however verification for certain classes of programs. To figure out ways of verifying new classes is the topic of extensive research in the program verification community.

1.3.1 Alternative approach

The approach presented here follows the same general idea as the classical one described in section 1.2.1 as it utilizes a weakest precondition calculus to generate proof obligations. In contrast to the classical approach however, it uses a more expressive logic involving fixpoints and recursion called μML. This logic is general enough to express the weakest (liberal) precondition of while statements directly, thus liberating us from loop invariant generation. As we eliminate the first problem we complicate the other, since proving statements in a more expressive logic is harder.

The outline of this approach is illustrated in the chart in figure 1.3.

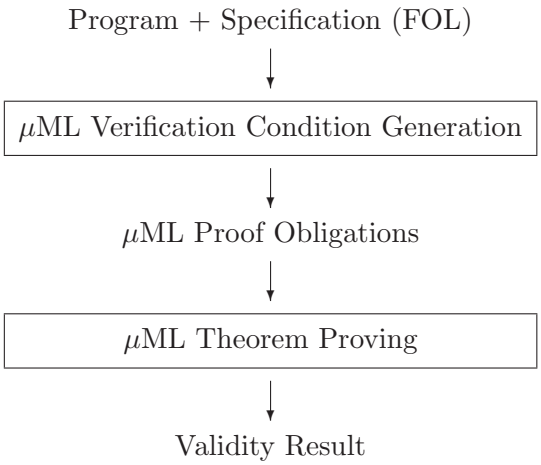


Figure 1.3. Our suggested approach to Hoare-style verification.

1.3.2 Goal of the Thesis

The goal for this thesis is to investigate this alternative approach and to evaluate its effectiveness on concrete examples. This is accomplished by developing the theory and implementing the following two components:

1. A verification condition generator or weakest precondition extractor

2. A specialized theorem prover for μML

As input we consider programs written in IMP [4] and specifications as pre- and postconditions from **Assn**.

1.3.3 Contributions

My supervisor, Dilian Gurov, invented the approach and has contributed with most of the theoretical results. Andreas Lundblad has explored and documented the approach in this thesis and developed a prototype tool that automates the proof search.

Chapter 2

Logic

This section presents the underlying frameworks, syntax and semantics of our logic.

2.1 Underlying Frameworks

The following frameworks stand as basis for the logic we present in the next section.

- Hoare Logic

  Our approach is built on Hoare logic since we consider programs as predicate transformers and use specifications in the form of pre- and postconditions.

- Updates

  Our logic utilizes *updates* in the same way as Angela Wallenburg does in [3]. These updates are essentially pairs of program locations and arithmetic expressions written as $\{X := e\}$. As we will show in the next section we will allow assertions of the form $\{X := e\}\phi$ which is satisfied by a state iff ϕ holds after execution of the assignment $X := e$.

- μ-Calculus

  As basis for our proof system we use the semantics and deductive rules presented in [14]. These include fixpoints and discharge conditions for recursive reasoning. The grammar used in [14] looks like this:

  $$\phi ::= \phi \wedge \phi \mid \neg\phi \mid \langle\alpha\rangle\phi \mid X \mid \mu X.\phi \mid (\mu X.\phi)^{\kappa}$$

  which is almost the same as the one we employ. The two essential differences are:

  - Updates as modalities
    Since we use updates as modalities some rules have different shape. When we verify programs by using equation 1.1 we go from checking that a program corresponds to a specification to checking that one formula implies

another. In this new context there is no LTS corresponding to the original program. Instead one could say that we consider a generalized LTS where all transitions always exist and are always enabled. This means that updates actually correspond to both diamond and box modalities.

– No propositional variables in the scope of negation

This is not an imposed restriction, but a consequence of the nature of our weakest precondition calculus. The calculus does instead, as you will see later in this section, mention the dual logical and fixpoint operators.

2.2 Extended Assertions

2.2.1 Syntax

The grammar for our extended specification language looks like this (note that ϕ is defined in the grammar for **Assn** in section 1.1.2):

Description	Name	Grammar
Extended assertions	**EAssn**	$\psi ::= \phi \mid Z \mid \psi \wedge \psi \mid \psi \vee \psi \mid \{X := a\}\psi \mid$ $\mu Z.\psi \mid \nu Z.\psi \mid \mu^{\kappa} Z.\psi \mid \nu^{\kappa} Z.\psi$

where Z ranges over the set of all propositional variables
κ ranges over the set of all ordinal variables

2.2.2 Semantics

We now define the denotational semantics for the extended assertion expressions. As usual a state satisfies an extended assertion iff it is in the denotation of the expression. The semantical function has the signature

$$\mathcal{A}x : \mathbf{EAssn} \to ((\mathbf{Intvar} \to \mathbf{N}) \to 2^{\mathbf{\Sigma}})$$

Definition ($\mathcal{A}x$) If V denotes our propositional variable valuation function and I our integer variable valuation function the $\mathcal{A}x$ is defined in the following way:

$$\mathcal{A}x[\![\psi]\!]^I_V = \mathcal{A}n[\![\psi]\!]^I \quad \text{if } \psi \in \mathbf{Assn}$$

$$\mathcal{A}x[\![Z]\!]^I_V = V(Z)$$

$$\mathcal{A}x[\![\psi_0 \wedge \psi_1]\!]^I_V = \mathcal{A}x[\![\psi_0]\!]^I_V \bigcap \mathcal{A}x[\![\psi_1]\!]^I_V$$

$$\mathcal{A}x[\![\psi_0 \vee \psi_1]\!]^I_V = \mathcal{A}x[\![\psi_0]\!]^I_V \bigcup \mathcal{A}x[\![\psi_1]\!]^I_V$$

$$\mathcal{A}x[\![\{X := e\}\psi]\!]^I_V = \{\sigma \mid \exists\sigma'.(\sigma \xrightarrow{X:=e} \sigma' \wedge \sigma' \in \mathcal{A}x[\![\psi]\!]^I_V)\}$$

$$\mathcal{A}x[\![\mu Z.\psi]\!]^I_V = \bigcap\{F \subseteq 2^{\mathbf{\Sigma}} \mid \mathcal{A}x[\![\psi]\!]^I_{V[F/Z]} \subseteq F\}$$

$$\mathcal{A}x[\![\nu Z.\psi]\!]^I_V = \bigcup\{F \subseteq 2^{\mathbf{\Sigma}} \mid \mathcal{A}x[\![\psi]\!]^I_{V[F/Z]} \supseteq F\}$$

$$\mathcal{A}x[\![\mu^\kappa Z.\psi]\!]^I_V = \begin{cases} \emptyset & \text{if } V(\kappa) = 0 \\ \mathcal{A}x[\![\psi]\!]^I_{V[\mathcal{A}x[\![\mu^\kappa Z.\psi]\!]^I_{V[\beta/\kappa]}/Z]} & \text{if } V(\kappa) = \beta + 1 \\ \bigcup\{\mathcal{A}x[\![\mu^\kappa Z.\psi]\!]^I_{V[\beta/\kappa]} \mid \beta < V(\kappa)\} & \text{if } V(\kappa) \text{ is a limit ordinal} \end{cases}$$

$$\mathcal{A}x[\![\nu^\kappa Z.\psi]\!]^I_V = \begin{cases} \mathbf{\Sigma} & \text{if } V(\kappa) = 0 \\ \mathcal{A}x[\![\psi]\!]^I_{V[\mathcal{A}x[\![\nu^\kappa Z.\psi]\!]^I_{V[\beta/\kappa]}/Z]} & \text{if } V(\kappa) = \beta + 1 \\ \bigcap\{\mathcal{A}x[\![\nu^\kappa Z.\psi]\!]^I_{V[\beta/\kappa]} \mid \beta < V(\kappa)\} & \text{if } V(\kappa) \text{ is a limit ordinal} \end{cases}$$

□

2.2.3 *wp-/wlp*-calculus

As mentioned earlier, our weakest precondition calculus handles formulas expressed in the extended assertion language, which includes least and greatest fixpoints. Below we define the weakest (liberal) preconditions of the commands in IMP.

Definition (wp/wlp for **Com**) The functions wp and wlp both have the same signature:

$$wp, wlp : (\mathbf{Com}, \mathbf{EAssn}) \to \mathbf{EAssn}$$

Given a formula $B \in \mathbf{EAssn}$ and a command, wp returns the weakest precondition:

$$wp(\mathbf{skip}, B) = B$$
$$wp(X := e, B) = \{X := e\}B$$
$$wp(c_0; c_1, B) = wp(c_0, wp(c_1, B))$$
$$wp(\mathbf{if}\ b\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1, B) = (b \wedge wp(c_0, B)) \vee (\neg b \wedge wp(c_1, B))$$
$$wp(\mathbf{while}\ b\ \mathbf{do}\ c, B) = \mu Z.(\neg b \wedge B) \vee (b \wedge wp(c, Z))$$

wlp takes a formula $B \in \mathbf{EAssn}$ and a command and returns the weakest liberal precondition:

$$wlp(\mathbf{skip}, B) = B$$
$$wlp(X := e, B) = \{X := e\}B$$
$$wlp(c_0; c_1, B) = wlp(c_0, wlp(c_1, B))$$
$$wlp(\mathbf{if}\ b\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1, B) = (b \wedge wlp(c_0, B)) \vee (\neg b \wedge wlp(c_1, B))$$
$$wlp(\mathbf{while}\ b\ \mathbf{do}\ c, B) = \nu Z.(\neg b \wedge B) \vee (b \wedge wlp(c, Z))$$

□

Correctness

We now show that the functions mentioned above actually do return the weakest precondition.

Theorem (Correctness of wp/wlp) For any command, c, and any extended assertions, B, the following holds:

$$\mathcal{A}x[\![wlp(c, B)]\!]^I_V = wlp^I_V[\![c, B]\!]$$

where $wlp^I_V[\![c, B]\!]$ denotes the set of all states, σ, such that $\mathcal{C}[\![c]\!]\sigma \vDash B$. □

Proof The proof proceeds by structural induction on the command.

Base case (skip and assignment)

$$\begin{aligned}\mathcal{A}x[\![wlp(\mathbf{skip}, B)]\!]^I_V &= \{\sigma \in \mathbf{\Sigma} \mid \mathcal{C}[\![\mathbf{skip}]\!]\sigma \vDash^I B\} \\ &= \{\sigma \in \mathbf{\Sigma} \mid \sigma \vDash^I B\}\end{aligned}$$

$$\begin{aligned}\mathcal{A}x[\![wlp(X := e, B)]\!]^I_V &= \{\sigma \in \mathbf{\Sigma} \mid \mathcal{C}[\![X := e]\!]\sigma \vDash^I B\} \\ &= \{\sigma \in \mathbf{\Sigma} \mid \sigma[\mathcal{A}[\![e]\!]\sigma/X] \vDash^I B\} \\ &= \{\sigma \in \mathbf{\Sigma} \mid \sigma \vDash^I \{X := e\}B\}\end{aligned}$$

Induction step (sequence, if, while)

$$\begin{aligned}\mathcal{A}x[\![wlp(c_0; c_1, B)]\!]^I_V &= \{\sigma \in \mathbf{\Sigma} \mid \mathcal{C}[\![c_1]\!](\mathcal{C}[\![c_0]\!]\sigma) \vDash^I B\} \\ &= \{\sigma \in \mathbf{\Sigma} \mid \mathcal{C}[\![c_0]\!]\sigma \vDash^I wlp(c_1, B)\} \\ &= \{\sigma \in \mathbf{\Sigma} \mid \sigma \vDash^I wlp(c_0, wlp(c_1, B))\}\end{aligned}$$

$$\begin{aligned}&\mathcal{A}x[\![wlp(\mathbf{if}\ b\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1, B)]\!]^I_V = \\ &\quad= \{\sigma \in \mathbf{\Sigma} \mid \mathcal{C}[\![\mathbf{if}\ b\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1, B]\!]\sigma \vDash^I B\} \\ &\quad= \{\sigma \in \mathbf{\Sigma} \mid (\mathcal{B}[\![b]\!]\sigma \vDash^I \mathbf{tt} \wedge \mathcal{C}[\![c_0]\!]\sigma \vDash^I B) \vee (\mathcal{B}[\![b]\!]\sigma \vDash^I \mathbf{ff} \wedge \mathcal{C}[\![c_1]\!]\sigma \vDash^I B)\} \\ &\quad= \{\sigma \in \mathbf{\Sigma} \mid (\sigma \vDash^I b \wedge \sigma \vDash^I wlp(c_0, B)) \vee (\sigma \vDash^I \neg b \wedge \sigma \vDash^I wlp(c_1, B))\} \\ &\quad= \{\sigma \in \mathbf{\Sigma} \mid \sigma \vDash^I (b \wedge wlp(c_0, B)) \vee (\neg b \wedge wlp(c_1, B))\}\end{aligned}$$

$$\mathcal{A}x[\![wlp(\mathbf{while}\ b\ \mathbf{do}\ c, B)]\!]^I_V = \mathcal{A}x[\![\nu Z.(\neg b \wedge B) \vee (b \wedge wlp(c, Z))]\!]^I_V$$

For the proof of the last case we refer to [15]. □

Correctness of wp is proved in a dual way.

2.3 Proof system

To prove the validity of formulas given in **EAssn** an adapted version of the Gentzen based proof system presented in [14] is used. Before continuing a few definitions are in place.

Definition (Assertion) Assertions come in three different shapes:

- *Satisfaction assertions*, has the form $\sigma : \psi$, where $\sigma \in \mathbf{\Sigma}$ and $\psi \in \mathbf{EAssn}$. It is valid for an propositional valuation function V and an integer valuation function I, iff $\sigma \in \mathcal{A}x[\![\psi]\!]^I_V$.
- *Transition assertions*, has the form $\sigma \xrightarrow{X:=e} \sigma'$, where $\sigma, \sigma' \in \mathbf{\Sigma}$. It is valid iff $\sigma' = \sigma[\mathcal{A}[\![e]\!]\sigma/X]$.
- *Ordinal constraint assertions*, has the form $\kappa < \kappa'$, and is valid for V if $V(\kappa) < V(\kappa')$. □

2.3.1 Derivation Rules

The derivation rules can be divided into three categories:

- Structural rules

  The rules in this category concern the structure of the sequent. These rules may restructure the sequent independently of what formulas are in it.

- Logical rules

  To this category those rules concerning the logical meaning of the sequent belong. These rules could be said to break down the sequent to smaller subgoals.

- Ordinal constraint rules

  Only one rule belongs to this category and it is a rule concerning the transitivity of ordinals.

We here bring up those three rules that are special for our proof system. All three belong to the logical category. For the entire list of rules we refer to appendix B.

First up is the Tr-L which states that a postcondition in the antecedent can be pulled back to become a precondition.

$$\frac{\Gamma, \sigma : \phi[e/X] \vdash \Delta}{\Gamma, \sigma \xrightarrow{X:=e} \sigma', \sigma' : \phi \vdash \Delta} \text{ Tr} - \text{L}$$

The Tr-R rules state the same thing about a postcondition in the succedent.

$$\frac{\Gamma \vdash \sigma : \phi[e/X], \Delta}{\Gamma, \sigma \xrightarrow{X:=e} \sigma' \vdash \sigma' : \phi, \Delta} \text{ Tr} - \text{R}$$

Finally we have the Up-R which transform an update in the succedent into a transition and a postcondition. Note that the introduced state may not be mentioned elsewhere in the sequent.

$$\frac{\Gamma, \sigma \xrightarrow{X:=e} \sigma' \vdash \sigma' : \phi, \Delta}{\Gamma \vdash \sigma : \{X := e\}\phi, \Delta} \text{ Up} - \text{R}, \ \sigma' \text{fresh}$$

2.3.2 Discharge Rules

The discharge rule described in [16] is a global rule based on fixpoint induction. If $\nu^{\kappa'} Z.\phi \Rightarrow \nu^{\kappa} Z.\phi$ and $\kappa' < \kappa$ we can deduce that $\nu^0 Z.\phi \Rightarrow \nu^{\kappa} Z.\phi$ and since $\nu^0 Z.\phi = \mathbf{tt}$, $\nu^{\kappa} Z.\phi$ must be true. The rule guarantees that the proof tree embodies a valid proof by well-founded induction.

2.3.3 Soundness

Local soundness

For soundness of all rules except Tr-R, Tr-L and Up-R we refer to [14].

- Tr-R:

$$\frac{\Gamma \vdash \sigma : \phi[e/X], \Delta}{\Gamma, \sigma \xrightarrow{X:=e} \sigma' \vdash \sigma' : \phi, \Delta} \text{ Tr} - \text{R}$$

  To show soundness of this rule we assume the premise:

$$\bigwedge \Gamma \Rightarrow \bigvee \sigma : \phi[e/X], \Delta \tag{2.1}$$

  and show the conclusion:

$$\bigwedge \Gamma, \sigma \xrightarrow{X:=e} \sigma' \Rightarrow \bigvee \sigma' : \phi, \Delta \tag{2.2}$$

To show 2.2 we assume the antecedent:

$$\bigwedge \Gamma, \sigma \xrightarrow{X:=e} \sigma' \tag{2.3}$$

and show the succedent:

$$\bigvee \sigma' : \phi, \Delta \tag{2.4}$$

2.1 and 2.3 gives

$$\bigvee \sigma : \phi[e/X], \Delta \tag{2.5}$$

Since 2.3 states that $\sigma' = \sigma[\mathcal{A}[\![e]\!]\sigma/X]$ we have left to show that $\sigma[\mathcal{A}[\![e]\!]\sigma/X] : \phi \vee \Delta$ which is the same as 2.5 (see [4] section 6.5).

- Tr-L

$$\frac{\Gamma, \sigma : \phi[e/X] \vdash \Delta}{\Gamma, \sigma \xrightarrow{X:=e} \sigma', \sigma' : \phi \vdash \Delta} \text{ Tr} - \text{L}$$

To show soundness of this rule we assume the premise:

$$\bigwedge \Gamma, \sigma : \phi[e/X] \Rightarrow \bigvee \Delta \tag{2.6}$$

and show the conclusion:

$$\bigwedge \Gamma, \sigma \xrightarrow{X:=e} \sigma', \sigma' : \phi \Rightarrow \bigvee \Delta \tag{2.7}$$

To show 2.7 we assume the antecedent:

$$\bigwedge \Gamma, \sigma \xrightarrow{X:=e} \sigma', \sigma' : \phi \tag{2.8}$$

Since 2.8 states that $\sigma' = \sigma[\mathcal{A}[\![e]\!]\sigma/X]$, 2.6 and 2.8 gives the succedent $\bigvee \Delta$ (see [4] section 6.5).

- Up-R

$$\frac{\Gamma, \sigma \xrightarrow{X:=e} \sigma' \vdash \sigma' : \phi, \Delta}{\Gamma \vdash \sigma : \{X := e\}\phi, \Delta} \text{ Up} - \text{R}, \ \sigma' \text{fresh}$$

Since σ' does not occur in Γ or Δ and $\sigma \xrightarrow{X:=e} \sigma'$ means that $\sigma' = \sigma[\mathcal{A}[\![e]\!]\sigma/X]$ the premise states that $\bigwedge \Gamma \Rightarrow \bigvee \sigma[\mathcal{A}[\![e]\!]\sigma/X] : \phi, \Delta$. Which is the same as the conclusion (again, see [4] section 6.5).

Global soundness

Global soundness refers to the soundness of the proof in its entirety. Apart from the local soundness this requires that the discharges are sound as well. This is proven in [16].

2.4 Examples

As a demonstration of how our program verification method works, we here give three examples.

2.4.1 Example 1 - Non-Terminating Program

The program in this example will not terminate if $X < 0$ in the initial state. The code follows here:

$$\begin{aligned} NonTerm \equiv \textbf{while } \neg(X = 0) \textbf{ do} \\ X := X - 1 \end{aligned}$$

To prove this we show that the following triple is valid:

$$\{X < 0\}\ NonTerm\ \{\mathbf{ff}\}$$

The weakest liberal precondition of $NonTerm$ and $\mathbf{ff}$ is:

$$\begin{aligned} & wlp(NonTerm, \mathbf{ff}) \\ = \ & \nu Z.(\neg b \wedge B) \vee (b \wedge wlp(X := X - 1, \mathbf{ff})) \\ = \ & \nu Z.(\neg b \wedge B) \vee (b \wedge \{X := X - 1\}\mathbf{ff})) \end{aligned}$$

2.4.2 Example 2 - Cut Required

The program in this example will check if X is even or odd.

$$\begin{aligned} EvenCheck \equiv \textbf{while } X >= 2 \textbf{ do} \\ X := X - 2 \end{aligned}$$

The proof in figure 2.2 demonstrates the necessity of the Cut-rule. The triple we prove correct is:

$$\{X = i \wedge X > 0\}\ EvenCheck\ \{(X = 0 \wedge even(i)) \vee (X = 1 \wedge odd(i))\}$$

Where $even(x)$ and $odd(x)$ are defined as $\exists i.x = 2i$ and $\neg even(x)$ respectively.

2.4.3 Example 3 - Nested Loops

We now give an example of nested loops.

$$\begin{aligned} Nested \equiv \textbf{while } X > 0 \textbf{ do} \\ \textbf{while } Y >= X \textbf{ do} \\ Y := Y - X; \\ X := X - 1 \end{aligned}$$

Given an initial state satisfying $X \geq 0 \wedge Y \geq 0 \wedge (X = 0 \Rightarrow Y = 0)$, this program will reset X and Y to zero. This is proven in figure 2.3

$$\dfrac{\dfrac{\checkmark}{\sigma : X < 0 \vdash \sigma : X - 1 < 0}}{\sigma : X < 0, \sigma \xrightarrow{X:=X-1} \sigma' \vdash \sigma' : X < 0}\ \mathrm{Tr-R} \qquad \dfrac{\checkmark}{\kappa' < \kappa \vdash \kappa' < \kappa}\ \mathrm{Id}$$

$$\dfrac{\sigma : X < 0, \kappa' < \kappa, \sigma : \neg(X = 0), \sigma \xrightarrow{X:=X-1} \sigma' \vdash \sigma' : \nu^{\kappa'} Z.\phi}{\sigma : X < 0, \kappa' < \kappa, \sigma : \neg(X = 0) \vdash \sigma : \{X := X - 1\}\nu^{\kappa'} Z.\phi}\ \mathrm{Up-R}$$

$$\dfrac{\dfrac{\checkmark}{\sigma : X < 0, \sigma : \neg(\neg(X = 0)) \vdash \sigma : \mathbf{ff}} \qquad \sigma : X < 0, \kappa' < \kappa \vdash (\neg(X = 0) \wedge \{X := X - 1\}\nu^{\kappa'} Z.\phi) \vee (X = 0 \wedge \mathbf{ff})}{\sigma : X < 0, \kappa' < \kappa \vdash (\neg(X = 0) \wedge \{X := X - 1\}\nu^{\kappa'} Z.\phi) \vee (X = 0 \wedge \mathbf{ff})}\ \mathrm{CondCut}$$

$$\dfrac{\sigma : X < 0 \vdash \sigma : \nu^{\kappa} Z.\phi}{\sigma : X < 0 \vdash \sigma : \nu Z.\phi}\ \nu^{\kappa}\mathrm{R}$$

$$\dfrac{\sigma : X < 0 \vdash \sigma : \nu^{\kappa} Z.\phi}{\sigma : X < 0 \vdash \sigma : \nu Z.\phi}\ \nu\mathrm{R}$$

Where $\phi \equiv (\neg(X = 0) \wedge \{X := X - 1\}Z) \vee (X = 0 \wedge \mathbf{ff})$.

Figure 2.1. Proof of example 1 - Non Terminating.

$$\dfrac{\dfrac{\dfrac{\checkmark}{\sigma : X > 1 \vdash \sigma : X - 2 \geq 0}}{\sigma : X > 1, \sigma \xrightarrow{X:=X-2} \sigma' \vdash \sigma' : X \geq 0}\ \mathrm{Tr-R} \quad \dfrac{\dfrac{\checkmark}{\sigma : even(X) \leftrightarrow even(i), \sigma : X > 1 \vdash \sigma : even(X-2) \leftrightarrow even(i)}}{\sigma : even(X) \leftrightarrow even(i), \sigma \xrightarrow{X:=X-2} \sigma', \sigma : X > 1 \vdash \sigma' : even(X) \leftrightarrow even(i)}\ \mathrm{Tr-R} \quad \dfrac{\checkmark}{\kappa' < \kappa \vdash \kappa' < \kappa}\ \mathrm{Id}}{\dfrac{\sigma : X > 1, \kappa' < \kappa, \sigma \xrightarrow{X:=X-2} \sigma' \vdash \sigma' : \nu^{\kappa'} Z.\phi}{(2)}}\ \mathrm{DisCut}$$

$$\dfrac{\dfrac{\dfrac{(2)}{\sigma : even(X) \leftrightarrow even(i), \kappa' < \kappa, \sigma : X > 1 \vdash \sigma : \{X := X-2\}\nu^{\kappa'} Z.\phi}\ \mathrm{Up-R} \quad \dfrac{\checkmark}{\sigma : X \geq 0, \sigma : even(X) \leftrightarrow even(i), \sigma : \neg(X > 1) \vdash \sigma : \neg(X > 1) \wedge ((X = 0 \wedge even(i)) \vee (X = 1 \wedge odd(i))}}{\sigma : X \geq 0, \sigma : even(X) \leftrightarrow even(i), \kappa' < \kappa \vdash \sigma : (X > 1 \wedge \{X := X-2\}\nu^{\kappa'} Z.\phi) \vee (\neg(X > 1) \wedge ((X = 0 \wedge even(i)) \vee (X = 1 \wedge odd(i)))}\ \mathrm{CondCut}}{\dfrac{\sigma : X \geq 0, \sigma : even(X) \leftrightarrow even(i) \vdash \sigma : \nu^{\kappa} Z.\phi}{(1)}}\ \nu^{\kappa}\mathrm{R}$$

$$\dfrac{\dfrac{\dfrac{\checkmark}{\sigma : X = i, \sigma : X \geq 0 \vdash \sigma : even(X) \leftrightarrow even(i), \sigma : \nu Z.\phi} \quad \dfrac{(1)}{\sigma : X = i, \sigma : X \geq 0, \sigma : even(X) \leftrightarrow even(i) \vdash \sigma : \nu Z.\phi}\ \nu\mathrm{R}}{\sigma : X = i, \sigma : X \geq 0 \vdash \sigma : \nu Z.\phi}\ \mathrm{Cut}}{\sigma : x = i \wedge X \geq 0 \vdash \sigma : \nu Z.\phi}\ \wedge\mathrm{L}$$

Where $\phi \equiv (X > 1 \wedge \{X := X - 2\}Z) \vee (\neg(X > 1) \wedge ((X = 0 \wedge even(i)) \vee (X = 1 \wedge odd(i)))$.

Figure 2.2. Proof of example 2 - Cut Required.

$$\dfrac{\dfrac{\dfrac{\checkmark}{\sigma:\phi,\sigma:X>0,\sigma:Y<X\vdash\sigma:\phi[X-1/X]}}{\sigma:\phi,\kappa'<\kappa,\sigma:X>0,\kappa'''<\kappa'',\sigma:Y<X,\sigma\xrightarrow{X:=X-1}\sigma'\vdash\sigma':\phi}\,\mathrm{Tr-R}\quad\dfrac{\checkmark}{\kappa'<\kappa\vdash\kappa'<\kappa}\,\mathrm{Id}}{\sigma:\phi,\kappa'<\kappa,\sigma:X>0,\sigma:Y<X,\sigma\xrightarrow{X:=X-1}\sigma'\vdash\sigma':\nu^{\kappa'}Z_0.\psi}\,\mathrm{DisCut}$$

(2)

$$\dfrac{\dfrac{\dfrac{\dfrac{\checkmark}{\sigma:\phi\vdash\sigma:\phi[Y-X/Y]}}{\sigma:\phi,\sigma\xrightarrow{Y:=Y-X}\sigma'\vdash\sigma':\phi}\,??\quad\dfrac{\checkmark}{\kappa'<\kappa\vdash\kappa'<\kappa}\,\mathrm{Id}\quad\dfrac{\dfrac{\checkmark}{\sigma:X>0\vdash\sigma:X>0}\,\mathrm{Id}}{\sigma:X>0,\sigma\xrightarrow{Y:=Y-X}\sigma'\vdash\sigma':X>0}\,\mathrm{Tr-R}}{\sigma:\phi,\kappa'<\kappa,\sigma:X>0,\kappa'''<\kappa'',\sigma:Y\geq X,\sigma\xrightarrow{Y:=Y-X}\sigma'\vdash\sigma':\nu^{\kappa'''}Z_1.\chi[\nu^{\kappa'}Z_0.\psi/Z_0]}\,\mathrm{DisCut}}{\sigma:\phi,\kappa'<\kappa,\sigma:X>0,\kappa'''<\kappa'',\sigma:Y\geq X\vdash\sigma:\{Y:=Y-X\}\nu^{\kappa'''}Z_1.\chi[\nu^{\kappa'}Z_0.\psi/Z_0]}\,\mathrm{Up-R}\quad\dfrac{(2)}{\sigma:\phi,\kappa'<\kappa,\sigma:X>0,\sigma:Y<X\vdash\sigma:\{X:=X-1\}\nu^{\kappa'}Z_0.\psi}\,\mathrm{Up-R}}{\dfrac{\sigma:\phi,\kappa'<\kappa,\sigma:X>0,,\kappa'''<\kappa''\vdash\sigma:(Y\geq X\wedge\{Y:=Y-X\}\nu^{\kappa'''}Z_1.\chi[\nu^{\kappa'}Z_0.\psi/Z_0])\vee(Y<X\wedge\{X:=X-1\}\nu^{\kappa'}Z_0.\psi)}{\sigma:\phi,\kappa'<\kappa,\sigma:X>0\vdash\sigma:\nu^{\kappa''}Z_1.\chi[\nu^{\kappa'}Z_0.\psi/Z_0]}\,\nu^{\kappa}\mathrm{R}}\,\mathrm{CondCut}$$

(1)

$$\dfrac{\dfrac{\dfrac{(1)}{\sigma:\phi,\kappa'<\kappa,\sigma:X>0\vdash\sigma:\nu Z_1.\chi[\nu^{\kappa'}Z_0.\psi/Z_0]}\,\nu\mathrm{R}\quad\dfrac{\checkmark}{\sigma:\phi,\kappa'<\kappa,\sigma:X\leq 0\vdash\sigma:X=0\wedge Y=0}}{\sigma:\phi,\kappa'<\kappa\vdash\sigma:(X>0\wedge\nu Z_1.\chi[\nu^{\kappa'}Z_0.\psi/Z_0])\vee(X\leq 0\wedge X=0\wedge Y=0)}\,\mathrm{CondCut}}{\dfrac{\sigma:\phi\vdash\sigma:\nu^{\kappa}Z_0.\psi}{\sigma:\phi\vdash\sigma:\nu Z_0.\psi}\,\nu\mathrm{R}}\,\nu^{\kappa}\mathrm{R}$$

Where

$$\phi\equiv X\geq 0\wedge Y\geq 0\wedge(X=0\Rightarrow Y=0)$$
$$\psi\equiv(X>0\wedge\nu Z_1.\chi)\vee(\neg(X>0)\wedge(X=0\wedge Y=0))$$
$$\chi\equiv(Y\geq X\wedge\{Y:=Y-X\}Z_1)\vee(\neg(Y\geq X)\wedge\{X:=X-1\}Z_0)$$

Figure 2.3. Proof for example 3 - Nested Loops.

Chapter 3

Implementation

The verification condition generator is a straightforward implementation of the *wlp*-function defined above and will not be further discussed. Instead we explain the implementation of the specialized automated theorem prover.

There are already several very good theorem provers for this kind of problem. *Simplify*, *HOL*, *Gandalf*, *PVS*, *E* to mention a few. One way of implementing our proposed approach would be to configure one of these existing systems to work with our proof system. Many of these systems have good heuristics and proof planning which would probably lead to a good result. However, this is not the approach explored in this thesis. Due to time limits we have chosen to treat the fixpoint reasoning separately in a Java program and delegate the rest to an automated theorem prover called *Isabelle*.

3.1 Tactics

During proof construction some structures of formulae are recurring more often than other. For instance, the formula $(b \wedge A) \vee (\neg b \wedge B)$ occurs once for every **if**/**while** statement. It is nice to have good handling for this type of formula. This is achieved by providing a tactic (a rule combination / tree expansion macro) for this type of formula. In this case the tactic is called CondCut. Other tactics we have implemented are DisCut and Forward.

3.1.1 CondCut

As stated above, **if** and **while** statements give rise to formulas of the form $(b \wedge A) \vee (\neg b \wedge B)$. In all situations we have run into such a formula it has turned out to be rewarding to use the CondCut tactic to prove it. The tactic is described in figure 3.1.

3.1.2 DisCut

After unfolding a fixpoint formula a discharge should eventually be applicable. The DisCut tactic is used when the discharge seems to be close. The tactic is explained in figure 3.2.

3.1.3 Forward

The Forward-tactic involves *forward reasoning* and requires some intelligence since ϕ (in figure 3.3) is not given.

Normally we apply backward reasoning as we drag the postcondition, ϕ, backward:

$$\sigma \xrightarrow{X:=e} \sigma', \sigma' : \phi \qquad \text{gives} \qquad \sigma : \phi[e/X]$$

The result $\phi[e/X]$ is the weakest precondition and the triple $\{\chi\}X := e\{\phi\}$ is true if χ implies $\phi[e/X]$. When applying forward reasoning we push the precondition forward:

$$\sigma : \phi, \sigma \xrightarrow{X:=e} \sigma' \qquad \text{gives} \qquad \sigma' : \psi$$

The result ψ is called strongest postcondition if it is the strongest assertion satisfying $\{\phi\}X := e\{\psi\}$.

There is no way of generating the strongest postcondition as it is with weakest precondition, thus generally the backward reasoning is preferred.

3.2 Automation

Our theorem prover is written in Java and designed to take care of the fixpoint formulas. The **Assn**-formulas are delegated to an automated theorem prover called *Isabelle* [17], which is a joint project between Lawrence C. Paulson (University of Cambridge) and Tobias Nipkow (Technical University of Munich). This approach is natural since most of the recursive reasoning can be separated from the FOL reasoning.

The theorem prover applies rules to sequents as long as they contain fixpoint operators. When a subgoal is reached that is in **Assn** it is delegated to the external theorem prover.

3.2.1 Search Tree Structure

Our implementation searches for a proof by exploring a tree of sequent nodes (proof goals) and rule nodes (rule applications). The root of the tree is a sequent node corresponding to the proof goal generated by the verification condition generator. The children of this node are rule nodes and represent all possible rule applications. The children of the rule application nodes are sequent nodes and represent the subgoals of the rule applications. An example of a tree is depicted in figure 3.4.

A valid proof is found when the root node has been *flagged proven*. A sequent node is flagged proven if *any* of its children is flagged proven (only one way of

$$\dfrac{\dfrac{\dfrac{\dfrac{\checkmark}{\Gamma, \sigma : b \vdash \sigma : b, \Delta}\,\mathrm{Id} \quad \Gamma, \sigma : b \vdash \sigma : \phi, \Delta}{\Gamma, \sigma : b \vdash \sigma : b \wedge \phi, \Delta}\,\wedge\mathrm{R}}{\Gamma \vdash \sigma : \neg b, \sigma : b \wedge \phi, \Delta}\,\neg\mathrm{R}}{\Gamma \vdash \sigma : \neg b, \sigma : b \wedge \phi, \sigma : \neg b \wedge \psi, \Delta}\,\mathrm{W}-\mathrm{R} \qquad \dfrac{\dfrac{\dfrac{\dfrac{\checkmark}{\Gamma, \sigma : \neg b \vdash \sigma : \neg b, \Delta}\,\mathrm{Id} \quad \Gamma, \sigma : \neg b \vdash \sigma : \psi, \Delta}{\Gamma, \sigma : \neg b \vdash \sigma : \neg b \wedge \psi, \Delta}\,\wedge\mathrm{R}}{\Gamma, \sigma : \neg b \vdash \sigma : b \wedge \phi, \sigma : \neg b \wedge \psi, \Delta}\,\mathrm{W}-\mathrm{R}}{\Gamma \vdash \sigma : b \wedge \phi, \sigma : \neg b \wedge \psi, \Delta}\,\mathrm{Cut}$$

$$\dfrac{\Gamma \vdash \sigma : b \wedge \phi, \sigma : \neg b \wedge \psi, \Delta}{\Gamma \vdash \sigma : (b \wedge \phi) \vee (\neg b \wedge \psi), \Delta}\,\vee\mathrm{R}$$

is abbreviated as

$$\dfrac{\Gamma, \sigma : b \vdash \Delta, \sigma : \phi \quad \Gamma, \sigma : \neg b \vdash \Delta, \sigma : \psi}{\Gamma \vdash \Delta, \sigma : (b \wedge \phi) \vee (\neg b \wedge \psi)}\,\mathrm{CondCut}$$

Figure 3.1. The CondCut tactic.

$$\dfrac{\dfrac{\Gamma \vdash \Delta, \rho(\gamma_0) \quad \dots \quad \Gamma \vdash \Delta, \rho(\gamma_i) \qquad \Gamma, \rho(\delta_0) \vdash \Delta \quad \dots \quad \Gamma, \rho(\delta_j) \vdash \Delta \qquad \Gamma \vdash \Delta, \rho(\kappa) < \kappa \quad \overset{\checkmark}{\dfrac{}{\Gamma, \rho(\gamma_0), \dots, \rho(\gamma_i) \vdash \Delta, \rho(\delta_0), \dots, \rho(\delta_j), \sigma' : \nu^{\rho(\kappa)} Z.\phi}}\ \mathrm{DC}}{\Gamma \vdash \Delta, \sigma' : \nu^{\rho(\kappa)} Z.\phi}\ \mathrm{CUT}}{\begin{array}{c}\vdots \\ \gamma_0, \dots, \gamma_i \vdash \delta_0, \dots \delta_j, \sigma : \nu^{\kappa} Z.\phi\end{array}\ \nu^{\kappa}\mathrm{R}}$$

Figure 3.2. The DisCut tactic.

$$\dfrac{\dfrac{\dfrac{\checkmark}{\Gamma, \sigma : \chi \vdash \sigma : \phi[e/X], \sigma' : \psi, \Delta}\,\mathrm{Tr-R}}{\Gamma, \sigma : \chi, \sigma \xrightarrow{X:=e} \sigma' \vdash \sigma' : \phi, \sigma' : \psi, \Delta} \qquad \Gamma, \sigma' : \phi \vdash \sigma' : \psi, \Delta}{\dfrac{\Gamma, \sigma : \chi, \sigma \xrightarrow{X:=e} \sigma' \vdash \sigma' : \psi, \Delta}{\Gamma, \sigma : \chi \vdash \sigma : \{X := e\}\psi, \Delta}\,\mathrm{Up-R}}\,\mathrm{Cut}$$

is abbreviated as

$$\dfrac{\Gamma, \sigma' : \phi \vdash \sigma' : \psi, \Delta}{\Gamma, \sigma : \chi \vdash \sigma : \{X := e\}\psi, \Delta}\,\text{Forward},\ \ \chi \Rightarrow \phi[e/X]$$

Figure 3.3. The FORWARD tactic.

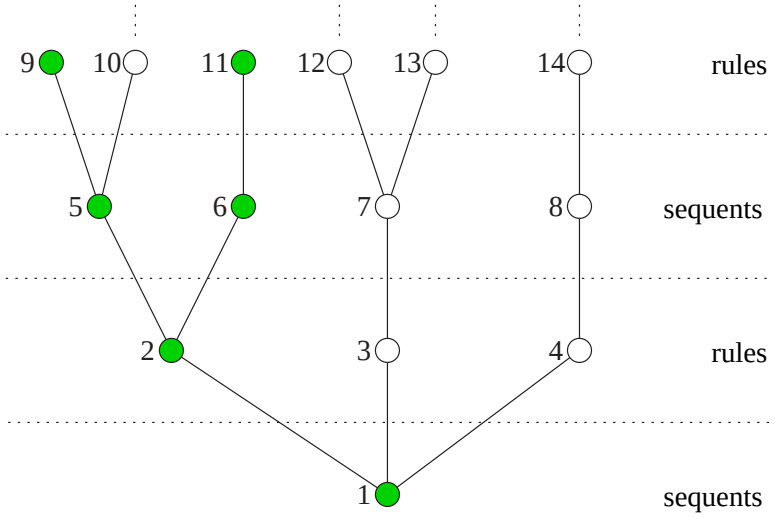


Figure 3.4. An example of a proof search tree. Nodes 1 and 5–8 represents sequents and 2–4 and 9–14 rule applications.

proving validity is required). A rule node, on the other hand, is flagged proven if *all* of its children are flagged proven (all subgoals need to be valid). Since axiom rules generate no subgoals they are always flagged proven. Nodes that are flagged proven in figure 3.4 are colored. Goal sequent is flagged proven since one rule application is flagged proven, since all its subgoals are flagged proven, ...

A node could also be *flagged unproven* which means that it has not been proven and no more attempts will be made to prove it. A sequent node is flagged unproven if all of its children are flagged unproven and a rule node is flagged unproven if any of its children is flagged unproven.

If a proof has been discovered it is extracted by removing all nodes not flagged proven. In figure 3.4 the goal is proven by applying the rule used in the node 2. The subgoals of this rule, node 5 and 6, are proven by applying the rules used in node 9 and 11 respectively.

3.2.2 Proof Search

To find a proof, our current implementation basically performs a breadth first search on the proof search tree. No intelligent proof planning has been developed. Instead some effort has been put into discharge detection and tree pruning.

Discharge detection

If the prover is about to generate rule applications for a sequent node containing a fixpoint approximation formula, it traverses all sequent nodes from that node back

to the root. If it runs into any sequent with a similar formula it attempts to apply the DisCut-tactic.

This companion search could be refined by searching for companions in other parts of the search tree as well. The problem is that the companion node has to be a part of the final proof, and the proper discharge conditions has to be met. For details, we refer to [16].

Search Tree Pruning

We here present some of the pruning tricks that we have found useful.

- Canceling expansion of proven and unprovable subtrees.

  If a node n has been flagged as proven or unprovable no more work needs to be put on trying to prove it. Thus expansion for all unexpanded leafs in the subtree with n as root is cancelled.

- No expansion of **Assn** satisfaction assertions.

  There is no point in trying to "help" the well-developed external theorem prover by expanding FOL expressions.

- Using tactics

  Since tactics are rule sequence abbreviations they help keeping the node count down.

- Subtree reusage

  If a subgoal is about to be created, that already exist in the search tree one can refer to the existing node instead. This is efficiently implemented by storing the subgoals in a hash set.

Chapter 4

Results

All programs in the test have been successfully proven in our proof system by hand. Unfortunately automatic theorem provers are not as good as humans and cannot do as smart applications of the CUT rule. As a result, our tool could not verify some programs automatically. Below the results from some tests are presented. The Hoare triples ht_0 - ht_9 are defined below the table.

Hoare Triple	Successfully verified	Generated nodes
ht_0	yes	38
ht_1	yes	18
ht_2	no	1256*
ht_3	no	1323*
ht_4	yes	63
ht_5	yes	68
ht_6	no	1323*
ht_7	yes	279
ht_8	no	13
ht_9	no	1335*

* Search was terminated before finishing.

$ht_0 \equiv \{\mathbf{tt}\}$
$\quad$ **while** $\neg(X = 0)$ **do**
$\quad\quad X := X - 1$
$\quad \{X = 0\}$

$ht_1 \equiv \{\mathbf{tt}\}$
$\quad Y := X;$
$\quad X := 2 \cdot X;$
$\quad X := (X + (0 - X + X) + X) \cdot 2;$
$\quad X := X - 8 \cdot Y$
$\quad \{X = 0\}$

$ht_2 \equiv \{Y = n \wedge Y > 0\}$
$X := 0;$
while $Y > 0$ **do**
$X := X + 1;$
$Y := Y - 1$
$\{X = n\}$

$ht_3 \equiv \{X < 0\}$
while $\neg(X = 0)$ **do**
$X := 0 - X;$
$\{\mathbf{ff}\}$

$ht_4 \equiv \{\neg(X = 0)\}$
while $\neg(X = 0)$ **do**
$X := 0 - X;$
$\{\mathbf{ff}\}$

$ht_5 \equiv \{X \geq 0 \wedge Y \geq 0 \wedge (X = 0 \Rightarrow Y = 0)\}$
while $X > 0$ **do**
while $Y >= X$ **do**
$Y := Y - X;$
$X := X - 1$
$\{X = 0 \wedge Y = 0\}$

$ht_6 \equiv \{X < 0\}$
while $\neg(X = 0)$ **do**
$X := 0 - X;$
$\{\mathbf{ff}\}$

$ht_7 \equiv \{I \geq 0 \wedge C \geq 1\}$
while $I > 0$ **do**
if $I \geq C$ **then**
$I := I - C$
else
$I := I - 1$
$\{I = 0\}$

$ht_8 \equiv \{X = i \wedge i \geq 0\}$
while $X > 1$ **do**
$X := X - 2$
$\{(X = 1 \wedge \neg\exists j.2j = i) \vee$
$(X = 0 \wedge \exists j.2j = i)\}$

$ht_9 \equiv \{n \geq 0 \wedge N = n\}$
$P := 0;$
$C := 0;$
while $C < N$ **do**
$P := P + M;$
$C := C + 1$
$\{P = M \cdot N\}$

All tests that terminated normally ran for less than two seconds.

The attentive reader has probably noted that those Hoare triples that were successfully proven had a precondition that was invariant during execution. This will be further discussed in the next section.

4.1 Conclusions

By using the presented approach one gets rid of the program text in an early stage. The information is now expressed in a well-developed mathematics with fixpoints. This context is in many ways better and in many ways worse.

Despite the fact that the logic is expressive enough to specify the weakest precondition of loops directly, all failed discharge attempts point to that the proofs still requires the precondition of a loop to be invariant. The reason is that the missing

assertions that the DisCut tactic tries to derive, include the precondition of the loop and if it is not invariant after one iteration it is not derivable.

It is easy to separate the proofs for the formulas in **EAssn** from those in **Assn**, which is good since much work has already been put into **Assn** proving. The general line of action to create a proof has been:

1. Approximate the fixpoint formula.
2. Unfold the formula once.
3. Prove loop base case.
4. Generate transitions by processing the updates.
5. Process transitions by forward or backward reasoning.
6. Apply a discharge rule.
7. Prove discharge side conditions.

Item 1, 2, 4, 5 and 6 concern fixpoints (**EAssn**) while item 3 and 7 concern FOL/arithmetic formulas (**Assn**).

As seen in example 3 this approach naturally extends to nested loops. The generated fixpoint expressions will be nested but the inner expression will not mention the bound propositional variable of the outer expression. The proof of example 3 (see figure 2.3) is straightforward and was actually generated with our prototype tool.

4.1.1 Future Work

Possible future work includes:

- Do the failed discharge attempts possess any valuable information?

  One question we have not answered is whether the failed discharge attempts may contribute with information when it comes to creating a widening. These two assertions for example: $\sigma : X = i \wedge i \geq 0$ and $\sigma : X - 2 = i \wedge i \geq 0$ happen to be equivalent in mod 2. Does this give us a hint about the loop invariant, which in this case was $even(X) \leftrightarrow even(i)$?

- Implementation approaches

  What external theorem prover fits this approach? Could the entire proof system be efficiently implemented in, say PVS? Are the existing proof plan facilities in these tools helpful in this case?

- Forward reasoning

  Would the approach benefit from utilizing state of the art forward reasoning strategies?

- Smarter expansion

  Our prototype theorem prover simply expands the proof search tree breadth first. Is there any better way? Could heuristics guide the prover in some better direction?

Bibliography

[1] What is Formal Methods?, `http://shemesh.larc.nasa.gov/fm/fm-what.html`, Date of access: 2006-12-8.

[2] J. Bradfield and C. Stirling, Modal logics and mu-calculi: an introduction, 2001.

[3] O. Olsson and A. Wallenburg, in *Software Engineering and Formal Methods. 3rd IEEE International Conference, SEFM 2005, Koblenz, Germany, September 7–9, 2005, Proceedings*, edited by B. Aichernig and B. Beckert (IEEE Computer Society Press, ADDRESS, 2005), pp. 180–189.

[4] G. Winskel, *The Formal Semantics of Programming Languages* (The MIT Press, Cambridge, Massachusetts, London, England, 1993).

[5] C. A. R. Hoare, Commun. ACM 12, 576 (1969).

[6] A. Blass and Y. Gurevich, ACM Trans. Comput. Logic 2, 1 (2001).

[7] N. Suzuki and K. Ishihata, in *POPL '77: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages* (ACM Press, New York, NY, USA, 1977), pp. 132–143.

[8] Z. Xu, B. P. Miller, and T. Reps, in *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation* (ACM Press, New York, NY, USA, 2000), pp. 70–82.

[9] P. Cousot and R. Cousot, in *POPL '77: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages* (ACM Press, New York, NY, USA, 1977), pp. 238–252.

[10] M. Sintzoff, SIGPLAN Not. 7, 203 (1972).

[11] K. R. M. Leino and F. Logozzo, in *Programming Languages and Systems, Third Asian Symposium, APLAS 2005* (Springer, ADDRESS, 2005), Vol. 3780 of Lecture Notes in Computer Science, pp. 119–134.

[12] S. Sankaranarayanan, H. B. Sipma, and Z. Manna, in *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (ACM Press, New York, NY, USA, 2004), pp. 318–329.

[13] A. Ireland and J. Stark, Fourth Nasa Langley Formal Methods Workshop (1997).

[14] M. Dam and D. Gurov, Journal of Logic and Computation 12, 43 (2002).

[15] E. C. R. Hehner, Acta Inf. 11, 287 (1979).

[16] C. Sprenger and M. Dam, Theoretical Informatics and Applications 37 365 (2003), special issue on FICS'02.

[17] Isabelle Home, `http://isabelle.in.tum.de`, Date of access: 2007-01-15.

Appendix A

Definitions of the semantical functions

A.1 Arithmetic expressions, $\mathcal{A}$

$$\mathcal{A}[\![n]\!] = \{(\sigma, n) \mid \sigma \in \mathbf{\Sigma}\}$$

$$\mathcal{A}[\![X]\!] = \{(\sigma, \sigma(X)) \mid \sigma \in \mathbf{\Sigma}\}$$

$$\mathcal{A}[\![a_0 + a_1]\!] = \{(\sigma, n_0 + n_1) \mid (\sigma, n_0) \in \mathcal{A}[\![a_0]\!] \wedge (\sigma, n_1) \in \mathcal{A}[\![a_1]\!]\}$$

$$\mathcal{A}[\![a_0 - a_1]\!] = \{(\sigma, n_0 - n_1) \mid (\sigma, n_0) \in \mathcal{A}[\![a_0]\!] \wedge (\sigma, n_1) \in \mathcal{A}[\![a_1]\!]\}$$

$$\mathcal{A}[\![a_0 \times a_1]\!] = \{(\sigma, n_0 \times n_1) \mid (\sigma, n_0) \in \mathcal{A}[\![a_0]\!] \wedge (\sigma, n_1) \in \mathcal{A}[\![a_1]\!]\}$$

A.2 Boolean expressions, $\mathcal{B}$

$$\mathcal{B}[\![\mathbf{tt}]\!] = \{(\sigma, \mathbf{tt}) \mid \sigma \in \mathbf{\Sigma}\}$$

$$\mathcal{B}[\![\mathbf{ff}]\!] = \{(\sigma, \mathbf{ff}) \mid \sigma \in \mathbf{\Sigma}\}$$

$$\begin{aligned}\mathcal{B}[\![a_0 = a_1]\!] = \quad & \{(\sigma, \mathbf{tt}) \mid \sigma \in \mathbf{\Sigma} \wedge \mathcal{A}[\![a_0]\!]\sigma = \mathcal{A}[\![a_1]\!]\sigma\} \cup \\ & \{(\sigma, \mathbf{ff}) \mid \sigma \in \mathbf{\Sigma} \wedge \mathcal{A}[\![a_0]\!]\sigma \neq \mathcal{A}[\![a_1]\!]\sigma\}\end{aligned}$$

$$\begin{aligned}\mathcal{B}[\![a_0 \leq a_1]\!] = \quad & \{(\sigma, \mathbf{tt}) \mid \sigma \in \mathbf{\Sigma} \wedge \mathcal{A}[\![a_0]\!]\sigma \leq \mathcal{A}[\![a_1]\!]\sigma\} \cup \\ & \{(\sigma, \mathbf{ff}) \mid \sigma \in \mathbf{\Sigma} \wedge \mathcal{A}[\![a_0]\!]\sigma \not\leq \mathcal{A}[\![a_1]\!]\sigma\}\end{aligned}$$

$$\mathcal{B}[\![\neg b]\!] = \{(\sigma, \neg t) \mid \sigma \in \mathbf{\Sigma} \wedge (\sigma, t) \in \mathcal{B}[\![b]\!]\}$$

$$\mathcal{B}[\![b_0 \wedge b_1]\!] = \{(\sigma, t_0 \wedge t_1) \mid \sigma \in \mathbf{\Sigma} \wedge (\sigma, t_0) \in \mathcal{B}[\![b_0]\!] \wedge (\sigma, t_1) \in \mathcal{B}[\![b_1]\!]\}$$

$$\mathcal{B}[\![b_0 \vee b_1]\!] = \{(\sigma, t_0 \vee t_1) \mid \sigma \in \mathbf{\Sigma} \vee (\sigma, t_0) \in \mathcal{B}[\![b_0]\!] \vee (\sigma, t_1) \in \mathcal{B}[\![b_1]\!]\}$$

A.3 Command expressions, $\mathcal{C}$

$$\mathcal{C}[\![\mathbf{skip}]\!] = \{(\sigma, \sigma) \mid \sigma \in \mathbf{\Sigma}\}$$

$$\mathcal{C}[\![X := a]\!] = \{(\sigma, \sigma[\mathcal{A}[\![a]\!]\sigma / X]) \mid \sigma \in \mathbf{\Sigma}\}$$

$$\mathcal{C}[\![c_0; c_1]\!] = \mathcal{C}[\![c_1]\!] \circ \mathcal{C}[\![c_0]\!]$$

$$\mathcal{C}[\![\mathbf{if}\ b\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1]\!] = \{(\sigma, \sigma') \mid \mathcal{B}[\![b]\!]\sigma = \mathbf{tt} \wedge (\sigma, \sigma') \in \mathcal{C}[\![c_0]\!]\} \cup \{(\sigma, \sigma') \mid \mathcal{B}[\![b]\!]\sigma = \mathbf{ff} \wedge (\sigma, \sigma') \in \mathcal{C}[\![c_1]\!]\}$$

$$\mathcal{C}[\![\mathbf{while}\ b\ \mathbf{do}\ c]\!] = \{(\sigma, \sigma') \mid \mathcal{B}[\![b]\!](\sigma) = \mathbf{tt} \wedge (\sigma, \sigma') \in \mathcal{C}[\![\mathbf{while}\ b\ \mathbf{do}\ c]\!] \circ \mathcal{C}[\![c]\!]\} \cup \{(\sigma, \sigma) \mid \mathcal{B}[\![b]\!](\sigma) = \mathbf{ff}\}$$

A.4 Extended arithmetic expressions, $\mathcal{A}v$

$$\mathcal{A}v[\![n]\!]_I = \{(\sigma, n) \mid \sigma \in \mathbf{\Sigma}\}$$

$$\mathcal{A}v[\![X]\!]_I = \{(\sigma, \sigma(X)) \mid \sigma \in \mathbf{\Sigma}\}$$

$$\mathcal{A}v[\![i]\!]_I = \{(\sigma, I(i)) \mid \sigma \in \mathbf{\Sigma}\}$$

$$\mathcal{A}v[\![a_0 + a_1]\!]_I = \{(\sigma, n_0 + n_1) \mid (\sigma, n_0) \in \mathcal{A}v[\![a_0]\!]_I \wedge (\sigma, n_1) \in \mathcal{A}v[\![a_1]\!]_I\}$$

$$\mathcal{A}v[\![a_0 - a_1]\!]_I = \{(\sigma, n_0 - n_1) \mid (\sigma, n_0) \in \mathcal{A}v[\![a_0]\!]_I \wedge (\sigma, n_1) \in \mathcal{A}v[\![a_1]\!]_I\}$$

$$\mathcal{A}v[\![a_0 \times a_1]\!]_I = \{(\sigma, n_0 \times n_1) \mid (\sigma, n_0) \in \mathcal{A}v[\![a_0]\!]_I \wedge (\sigma, n_1) \in \mathcal{A}v[\![a_1]\!]_I\}$$

A.5 Assertion expressions, $\mathcal{A}n$

Let I be the integer valuation function and let P denote all states in the transition system.

$$\mathcal{A}n[\![\mathbf{tt}]\!]^I = P$$

$$\mathcal{A}n[\![\mathbf{ff}]\!]^I = \emptyset$$

$$\mathcal{A}n[\![a_0 = a_1]\!]^I = \{\sigma \mid \mathcal{A}v[\![a_0]\!]_I\sigma = \mathcal{A}v[\![a_1]\!]_I\sigma\}$$

$$\mathcal{A}n[\![a_0 \leq a_1]\!]^I = \{\sigma \mid \mathcal{A}v[\![a_0]\!]_I\sigma \leq \mathcal{A}v[\![a_1]\!]_I\sigma\}$$

$$\mathcal{A}n[\![\phi_0 \wedge \phi_1]\!]^I = \mathcal{A}n[\![\phi_0]\!]^I \bigcap \mathcal{A}n[\![\phi_1]\!]^I$$

$$\mathcal{A}n[\![\phi_0 \vee \phi_1]\!]^I = \mathcal{A}n[\![\phi_0]\!]^I \bigcup \mathcal{A}n[\![\phi_1]\!]^I$$

$$\mathcal{A}n[\![\neg\phi]\!]^I = P - \mathcal{A}n[\![\phi]\!]^I$$

$$\mathcal{A}n[\![\forall i.\phi]\!]^I = \{\sigma \mid \sigma \in \mathcal{A}n[\![\phi]\!]^{I[n/i]}\} \text{ for every } n$$

$$\mathcal{A}n[\![\exists i.\phi]\!]^I = \{\sigma \mid \sigma \in \mathcal{A}n[\![\phi]\!]^{I[n/i]}\} \text{ for some } n$$

A.6 Extended assertion expressions, $\mathcal{A}x$

If V denotes our propositional variable valuation function and I our integer variable valuation function the $\mathcal{A}x$ is defined in the following way:

$$\mathcal{A}x[\![\psi]\!]^I_V = \mathcal{A}n[\![\psi]\!]^I \quad \text{if } \psi \in \mathbf{Assn}$$

$$\mathcal{A}x[\![Z]\!]^I_V = V(Z)$$

$$\mathcal{A}x[\![\psi_0 \wedge \psi_1]\!]^I_V = \mathcal{A}x[\![\psi_0]\!]^I_V \bigcap \mathcal{A}x[\![\psi_1]\!]^I_V$$

$$\mathcal{A}x[\![\psi_0 \vee \psi_1]\!]^I_V = \mathcal{A}x[\![\psi_0]\!]^I_V \bigcup \mathcal{A}x[\![\psi_1]\!]^I_V$$

$$\mathcal{A}x[\![\{X := e\}\psi]\!]^I_V = \{\sigma \mid \exists\sigma'.(\sigma \xrightarrow{X:=e} \sigma' \wedge \sigma' \in \mathcal{A}x[\![\psi]\!]^I_V)\}$$

$$\mathcal{A}x[\![\mu Z.\psi]\!]^I_V = \bigcap\{F \subseteq 2^{\boldsymbol{\Sigma}} \mid \mathcal{A}x[\![\psi]\!]^I_{V[F/Z]} \subseteq F\}$$

$$\mathcal{A}x[\![\nu Z.\psi]\!]^I_V = \bigcup\{F \subseteq 2^{\boldsymbol{\Sigma}} \mid \mathcal{A}x[\![\psi]\!]^I_{V[F/Z]} \supseteq F\}$$

$$\mathcal{A}x[\![\mu^\kappa Z.\psi]\!]^I_V = \begin{cases} \emptyset & \text{if } V(\kappa) = 0 \\ \mathcal{A}x[\![\psi]\!]^I_{V[\mathcal{A}x[\![\mu^\kappa Z.\psi]\!]^I_{V[\beta/\kappa]}/Z]} & \text{if } V(\kappa) = \beta + 1 \\ \bigcup\{\mathcal{A}x[\![\mu^\kappa Z.\psi]\!]^I_{V[\beta/\kappa]} \mid \beta < V(\kappa)\} & \text{if } V(\kappa) \text{ is a limit ordinal} \end{cases}$$

$$\mathcal{A}x[\![\nu^\kappa Z.\psi]\!]^I_V = \begin{cases} \boldsymbol{\Sigma} & \text{if } V(\kappa) = 0 \\ \mathcal{A}x[\![\psi]\!]^I_{V[\mathcal{A}x[\![\nu^\kappa Z.\psi]\!]^I_{V[\beta/\kappa]}/Z]} & \text{if } V(\kappa) = \beta + 1 \\ \bigcap\{\mathcal{A}x[\![\nu^\kappa Z.\psi]\!]^I_{V[\beta/\kappa]} \mid \beta < V(\kappa)\} & \text{if } V(\kappa) \text{ is a limit ordinal} \end{cases}$$

Appendix B

Complete list of derivation rules

B.1 Structural rules

$$\frac{\cdot}{\Gamma, A \vdash A, \Delta}\ \mathrm{Id} \qquad \frac{\Gamma \vdash A, \Delta \quad \Gamma, A \vdash \Delta}{\Gamma \vdash \Delta}\ \mathrm{Cut}$$

$$\frac{\Gamma \vdash \Delta}{\Gamma, A \vdash \Delta}\ \mathrm{W-L} \qquad \frac{\Gamma \vdash \Delta}{\Gamma \vdash A, \Delta}\ \mathrm{W-R}$$

B.2 Logical rules

$$\frac{\Gamma \vdash \sigma : \phi, \Delta}{\Gamma, \sigma : \neg\phi \vdash \Delta}\ \neg\mathrm{L} \qquad \frac{\Gamma, \sigma : \phi \vdash \Delta}{\Gamma \vdash \sigma : \neg\phi, \Delta}\ \neg\mathrm{R}$$

$$\frac{\Gamma, \sigma : \phi \vdash \Delta \quad \Gamma, \sigma : \psi \vdash \Delta}{\Gamma, \sigma : \phi \vee \psi \vdash \Delta}\ \vee\mathrm{L} \qquad \frac{\Gamma \vdash \sigma : \phi, \sigma : \psi, \Delta}{\Gamma \vdash \sigma : \phi \vee \psi, \Delta}\ \vee\mathrm{R}$$

$$\frac{\Gamma, \sigma : \phi, \sigma : \psi \vdash \Delta}{\Gamma, \sigma : \phi \wedge \psi \vdash \Delta}\ \wedge\mathrm{L} \qquad \frac{\Gamma \vdash \sigma : \phi, \Delta \quad \Gamma \vdash \sigma : \psi, \Delta}{\Gamma \vdash \sigma : \phi \wedge \psi, \Delta}\ \wedge\mathrm{R}$$

$$\frac{\Gamma, \sigma : \phi[e/X] \vdash \Delta}{\Gamma, \sigma \xrightarrow{X:=e} \sigma', \sigma' : \phi \vdash \Delta}\ \mathrm{Tr-L} \qquad \frac{\Gamma \vdash \sigma : \phi[e/X], \Delta}{\Gamma, \sigma \xrightarrow{X:=e} \sigma' \vdash \sigma' : \phi, \Delta}\ \mathrm{Tr-R}$$

$$\frac{\Gamma, \sigma : \phi[j/i] \vdash \Delta}{\Gamma, \sigma : \forall i.\phi \vdash \Delta}\ \forall\mathrm{L} \qquad \frac{\Gamma \vdash \sigma : \phi[j/i], \Delta}{\Gamma \vdash \sigma : \forall i.\phi, \Delta}\ \forall\mathrm{R},\ j\mathrm{fresh}$$

$$\frac{\Gamma, \sigma : \phi[j/i] \vdash \Delta}{\Gamma, \sigma : \exists i.\phi \vdash \Delta}\ \exists\text{L},\ j\text{fresh} \qquad \frac{\Gamma \vdash \sigma : \phi[j/i], \Delta}{\Gamma \vdash \sigma : \exists i.\phi, \Delta}\ \exists\text{R}$$

$$\frac{\Gamma, \sigma \xrightarrow{X:=e} \sigma' \vdash \sigma' : \phi, \Delta}{\Gamma \vdash \sigma : \{X := e\}\phi, \Delta}\ \text{Up} - \text{R},\ \sigma'\text{fresh}$$

$$\frac{\Gamma \vdash \sigma : \nu^\kappa Z.\phi, \Delta}{\Gamma \vdash \sigma : \nu Z.\phi, \Delta}\ \nu\text{R},\ \kappa\text{fresh}$$

$$\frac{\Gamma, \kappa' < \kappa \vdash \sigma : \phi[\nu^{\kappa'} Z.\phi/Z], \Delta}{\Gamma \vdash \sigma : \nu^\kappa Z.\phi, \Delta}\ \nu^\kappa\text{R},\ \kappa'\text{fresh}$$

$$\frac{\Gamma \vdash \sigma : \phi[\mu Z.\phi/Z], \Delta}{\Gamma \vdash \sigma : \mu Z.\phi, \Delta}\ \mu\text{R}$$

$$\frac{\Gamma \vdash \kappa' < \kappa, \Delta \quad \Gamma \vdash \sigma : \phi[\mu^{\kappa'} Z.\phi/Z], \Delta}{\Gamma \vdash \sigma : \mu^\kappa Z.\phi, \Delta}\ \mu^\kappa\text{R},\ \kappa'\text{fresh}$$

B.3 Ordinal constraint rules

$$\frac{\Gamma, \kappa' < \kappa \vdash \kappa'' < \kappa', \Delta}{\Gamma, \kappa' < \kappa \vdash \kappa'' < \kappa, \Delta}\ \text{OrdTr}$$