Automatic Verification of Hoare Triples

Levente Sándor

Supervisors:
Arnold Meijster,
Gerard Renardel de Lavalette

July 2018

Abstract

This thesis presents an automatic proof generation system that can generate proofs for Hoare triples. The system works with an imperative, Pascal-like language similar to the one used in the course Program Correctness. In this language we can define assertions about the program state (boolean expressions) and commands (assignments). The system makes use of a knowledge base to avoid hard-coding arithmetic rules and operations. Given a precondition P, a series of assignments S, and a postcondition Q, the proof generator tries to find a proof for the Hoare triple $\{P\}S\{Q\}$. The possible outcomes can be:

- *valid: a resolution proof*
- *invalid: a counterexample*
- *undecided: the system did not find a proof nor a counter-example*

The output "undecided" is necessary since it is fundamentally impossible to make a system that generates a proof or counter-example for each possible input.

Contents

1 Introduction **2**
1.1 States . . . 2
1.2 Hoare triple . . . 2
1.3 Annotated proof of correctness . . . 3
1.4 Weakest precondition . . . 4
1.5 Related work . . . 5

2 Assignments, Expressions, and their representation **7**
2.1 Assignments . . . 7
2.2 Expressions . . . 7
2.2.1 Predicates . . . 8
2.2.2 Relations . . . 8
2.2.3 Arithmetic Expressions . . . 9
2.2.4 Assignments . . . 9
2.3 Computation of the weakest precondition . . . 10
2.4 Parsing input . . . 11
2.4.1 Parsing arithmetic expressions . . . 12

3 Proof Generation **15**
3.1 Resolution (propositional logic) . . . 15
3.1.1 Resolution algorithm . . . 17
3.1.2 Conjunctive Normal Form . . . 17
3.2 Unification . . . 18
3.2.1 Implementation . . . 18
3.3 Resolution and Unification for FOL . . . 20
3.4 Example: the swap program fragment . . . 22

4 Conclusion and future work **25**
4.1 Future work . . . 25
4.1.1 Extension of the programming language . . . 25
4.1.2 Improvements to proof generation . . . 25
4.1.3 Experimental improvements . . . 26

Chapter 1

Introduction

Before jumping into the design, a few key concepts should be understood by the reader. Provided below is an overview of some rules and notations the system relies on, as well as references to related work.

1.1 States

An *imperative program* modifies during its execution values stored in memory locations associated with the variables of the program. The collection of variables and their corresponding values is called the *state* of the program. Instead of thinking about bits, bytes, or registers, we can consider the state as a set of pairs (v, x), where v is an identifier (the name of a variable) and x a value of the type of v.

Example: Suppose we have a program fragment that modifies two integer variables x and y, and starts in a state in which $x = y = 0$. The program fragment increments x and decrements y. So, before execution we can represent the state as the set $\{(x, 0), (y, 0)\}$, while after execution the state is represented by $\{(x, 1), (y, -1)\}$.

1.2 Hoare triple

In [10, 4] the axiomatic basis for imperative programs is given. A Hoare triple $\{P\}\, S\, \{Q\}$ consists of

- a precondition P: a boolean expression describing the state before execution of S,
- a series of statements S: the statements that make up the program fragment,
- a postcondition Q: a boolean expression which describes the state after execution of S.

To show that the program fragment is correct, we need to show that if we start from a state in which P is valid, and execute the statements S, we end up in a state in which Q is valid.
While the formal definition specifies a series of statements S, this project is limited to assignments only. Other types of statements are regarded as future work.

Example: The following Hoare triple is of a program fragment that swaps two integer variables without using a temporary variable. Each line between pre- and postcondition is an assignment from the series S.

$\{\,\mathbf{P} : x = X \,\wedge\, y = Y\,\}$
$x := x + y;$
$y := x - y;$
$x := x - y;$
$\{\,\mathbf{Q} : x = Y \,\wedge\, y = X\,\}$

This fragment will be used as a running example throughout the rest of the thesis.

1.3 Annotated proof of correctness

An annotated correctness proof for the above Hoare triple might look like this:

$\{\,\mathbf{P} : x = X \wedge y = Y\,\}$
$(*\text{ prepare } x := x + y : x = X \wedge y = Y \Rightarrow x + y = X + Y *)$
$\{\,x + y = X + Y \wedge y = Y\,\}$
$x := x + y;$
$\{\,x = X + Y \wedge y = Y\,\}$
$(*\text{ prepare } y := x - y : x = X + Y \wedge y = Y \Rightarrow x - y = X + Y - Y = X *)$
$\{\,x = X + Y \wedge x - y = X\,\}$
$y := x - y;$
$\{\,x = X + Y \wedge y = X\,\}$
$(*\text{ prepare } x := x - y : x = X + Y \wedge y = X \Rightarrow x - y = X + Y - X = Y *)$
$\{\,x - y = Y \wedge y = X\,\}$
$x := x - y$
$\{\,\mathbf{Q} : x = Y \wedge y = X\,\}$

This kind of proof is typically produced manually and requires creativity and arithmetic skills from its creator. For example, in the first step the precondition is converted into the equivalent predicate $x + y = X + Y \wedge y = Y$. This is done as preparation for the assigngment $x := x + y$, which means that x will change so we need to take the assignment's precondition and rewrite each occurence of x to $x + y$. Note that some authors therefore almost mechanically convert the precondition into $x + y - y = X \wedge y = Y$.

The conversion of the precondition into the equivalent form $\{x + y = X + Y \wedge y = Y\}$ is needed because we have only one rule that specifies the semantics of assignments. This rule is called the *assignment axiom*:

$$\{[E/x]R\}\, x := E\, \{R\} \tag{1.1}$$

The notation $[E/x]R$ refers to the predicate that is obtained from predicate R by replacing each occurence of x by the expresssion E. So, for R being $\{x = X + Y \wedge y = Y\}$, we have $[x + y/x]R \equiv \{x + y = X + Y \wedge y = Y\}$, the rewritten version of P.
Note that we used a second axiom in the above proof. This axiom is the rule of *sequential composition*, which is given by:

$$(\{P\}S_0\{Q'\}) \wedge (\{Q'\}S_1\{Q\}) \Rightarrow (\{P\}S_0; S_1\{Q\}) \tag{1.2}$$

Using this rule, correctness proofs for a series of assignments can be made. Another important axiom that was implicitly used in the given proof is:

$$((R \Rightarrow T) \wedge (\{T\}S\{U\}) \Rightarrow \{R\}S\{U\} \tag{1.3}$$

This rule strengthens the precondition T to a stronger predicate R. Similarly we have an axiom to weaken the postcondition:

$$((\{R\}S\{U\}) \wedge \{U \Rightarrow T\}) \Rightarrow \{R\}S\{T\} \tag{1.4}$$

1.4 Weakest precondition

The weakest precondition of a series of assignments S and a postcondition Q is the weakest predicate that needs to hold before execution of S such that we end up in a state in which Q holds. This weakest predicate is denoted as $wp(S, Q)$. So, for a Hoare triple $\{P\}\, S\, \{Q\}$ to be true, we need the proof rule:

$$P \Rightarrow wp(S, Q) \tag{1.5}$$

The function wp is actually simple to compute as it relies on purely syntactic rewrites using the following recursive wp−rule:

$$wp(S, Q) = \begin{cases} Q, & \text{if } S = [\,] \text{ (i.e. empty)}, \\ [E/x]Q, & \text{if } S = [x := E] \text{ (i.e. singleton list)}, \\ wp(S_0, wp(S_1, Q)), & \text{if } S = [S_0; S_1], \text{ where } S_0 \text{ is a single assignment} \end{cases} \tag{1.6}$$

If we apply this rule to the swapping example we get:

$$\begin{aligned}
&\quad wp(x := x + y; y := x - y; x := x - y,\ \ x = Y \wedge y = X) \\
&\equiv \{\text{recursive case wp-rule}\} \\
&\quad wp(x := x + y; y := x - y,\ \ wp(x := x - y,\ \ x = Y \wedge y = X)) \\
&\equiv \{\text{base case wp-rule}\} \\
&\quad wp(x := x + y; y := x - y,\ \ x - y = Y \wedge y = X) \\
&\equiv \{\text{recursive case wp-rule}\} \\
&\quad wp(x := x + y,\ \ wp(y := x - y,\ \ x - y = Y \wedge y = X)) \\
&\equiv \{\text{base case wp-rule}\} \\
&\quad wp(x := x + y,\ \ x - (x - y) = Y \wedge x + y - y = X) \\
&\equiv \{\text{base case wp-rule}\} \\
&\quad x + y - (x + y - y) = Y \wedge x + y - y = X
\end{aligned}$$

Note that this process only performs substitutions and no arithmetic simplifications, this will be important later! However, by applying some arithmetics ourselves, we can easily conclude that

$$\begin{aligned}
&\quad x + y - (x + y - y) = Y \wedge x + y - y = X \\
&\equiv \\
&\quad P : y = Y \wedge x = X
\end{aligned}$$

So, in this case we have $P \equiv wp(S, Q)$ and therefore surely $P \Rightarrow wp(S, Q)$. Hence, the Hoare triple is correct.

The reader might wonder why we need the function wp in the first place, since we already gave an annotated correctness proof of the code fragment in section 1.3 that does not use the wp- rule. The answer to this question is two-fold. First, the annotated proof actually uses the rule in disguise. In each step of the proof that was annotated with "prepare ...", the assertions were actually massaged into the weakest precondition of the following assignment. Second, the presentation of annotated proof is top-down (line after line), but in doing that the author needs to use creativity and calculus to rewrite predicates in such a form that the assignment rule can be applied. This kind of creativity cannot be expected from an automated verification system, so

another approach is needed. By doing the calculuation of the weakest precondition in a bottom-up fashion (starting with the last assignment), the calculation of the weakest precondition is a purely syntactic process of substituting expressions that can easily be done by a computer program. The drawback is that the computed precondition might have an unnecessary complicated format (like $x + y - (x + y - y) = Y \wedge x + y - y = X$ instead of $x = X \wedge y = Y$). However, we will solve that issue later.

1.5 Related work

The foundation for *manually* producing formal proofs of the correctness of program fragments has a long history, and was initiated by Robert Floyd (see [7]). His contribution was later formalized into a set of formal rules for correctness proofs by Tony Hoare (see [10]). Later, Edsger Dijkstra introduced proofs based on the weakest precondition operator in [4, 5].

The interest for *automated* verification of the correctness of program fragments is much more recent. In fact, nowadays there is even a complete research institute devoted to the subject in the UK called the *Research Institute in Automated Program Analysis and Verification* (see URL: `https://verificationinstitute.org/`).

Automatic program correctness verification is a subfield of a more general domain called *automated reasoning*. Automated reasoning has been most commonly used to build automated theorem provers. The problem with these theorem provers is that they rarely are capable to produce complete proofs on their own, but require some human guidance instead. Therefore, it may be better to qualify these programs as *proof assistants*. Moreover, these proof assistants target the much more general problem of proving the correctness of a mathematical theorem that is reformulated in terms of logic. A few well-known proof assistants are:

- HOL: an interactive proof assistant for Higher-Order Logic (ref. [9], publicly available via `https://hol-theorem-prover.org`).
- ISABELLE: another proof assistant for higher order logic (ref. [14]). The heart of the system is a proof method called resolution with unification, a technique that we will use as the heart of our program as well. The system and its documentation is publicly available via the website `https://isabelle.in.tum.de/`. Moreover, the community using this proof assistant maintains a nice website with a collection of proofs that have been produced using ISABELLE (see `https://www.isa-afp.org/`).
- COQ: another interactive proof assistant (ref. [2]) that allows semi-interactive development of machine-checked proofs and programs. The software can be obtained via the website `https://coq.inria.fr/`. With COQ the correctness of the CompCert C compiler was proven (see [12]). Another highlight is a computer assisted proof for the famous "four color problem" (see [8]).
- NQTHM: This proof assistant is better known as the "Boyer-Moore" theorem prover (ref. [3]). It is one of the oldest proof assistants, and uses a LISP-style language to specify theorems.

A nice overview of the usage of these systems is given in [17].

All of these proof assistants require a solid mathematical background of its users. Moreover, it requires that the user is able to translate his problem into some formal specification language which is accepted by the proof assistant. This is quite a cumbersome task, and in our view limits very much the general acceptance of these tools by the general public. In the introduction

section of the HOL website we found the warning "*Starting from scratch, it takes on average about a month to become comfortable using HOL.*". An example that illustrates that it is quite cumbersome and complicated to specify a theorem in most proof assistants is the following HOL-example in which the user tries to prove that multiplication is a commutative operation (example taken from the HOL website):

```
(defn times (x y)
  (if (zero x)
      0
      (plus y (times (sub1 x) y))))

 (prove-lemma commutativity-of-times (rewrite)
   (equal (times x z) (times z x)))
```

The literature on (and availablility of) proof assistants for mathematical theorems expressed in some logic is abundant. However, we could hardly find proof assistants that were dedicated to the task of proving the correctness of Hoare triples. This does not mean that this cannot be done using the above mentioned assistants, but it requires manual labour to convert the proof of $P \Rightarrow wp(S, Q)$ into the formalism of the proof assistant. In our view, that is a price that most programmers are not willing to pay.

In [6] we found a proof assistant that is dedicated to correctness verification or program fragments. The heart of the system is based on automata theory. The approach is to first construct an automaton for a candidate proof manually, followed by an automated verification of validity using graph algorithms and automata theory. We do not discuss this approach any further, since it requires manual construction of a candidate proof and the corresponding automaton, while we aim at a fully automated process where the user does not need to specify more than a precondition, a postcondition and a series of statements.

Chapter 2

Assignments, Expressions, and their representation

2.1 Assignments

In this project we will restrict ourselves to program fragments that solely consist of a series of assignments of the form $x := E$, where x is an integer valued program variable and E an integer valued arithmetical expression. Assignments are separated by a semicolon. This means that our proof assistant can not deal with other data types, nor can it deal with loops or conditional statements. An extension of the project that allows these features is reserved for future research based upon this project. Note that this does not mean that the program is not useful in the analysis of programs with while-loops. For example, in a typical while-program of the form

$$\textbf{while } B \textbf{ do} S$$

where S consists of a series of assignments, one of the standard proof obligations that need to be proven is the Hoare triple (see [10, 4])

$$\{J \wedge B \wedge \mathrm{vf} = V\}\ S\ \{J \wedge \mathrm{vf} < V\}$$

In this Hoare triple the predicate J is called the invariant of the loop, and vf (variant function, a.k.a. the *bounded function*) is some positive integer valued expression in the program variables, and V a specification constant. If the satement S consists of a series of assignments, then the automated proof generator might be able to find a proof for this Hoare triple.

2.2 Expressions

Of course, we need to represent symbolical expressions. In the remainder of this chapter we will discuss how we represent logical expressions, how we parse them, and how we compute weakest preconditions. We chose Haskell as the implementation language for several reasons. First, the pattern matching facilities of Haskell make the implementation elegant, readable, and easily maintainable. Second, it is easy to create and destroy data structures without the burden of memory management. Moreover, the lazy evaluation of expressions in Haskell made the implementation of certain features (mainly resolution, see 3.1) easier.

2.2.1 Predicates

The system needs a representation for predicates in order to process Hoare triples. Predicates consist of one or more *relations* joined by the standard boolean connectives. We support the connectives $\wedge$ (logical and), $\vee$ (logical or), $\neg$ (logical negation), $\Rightarrow$ (implies), $\Leftarrow$ (follows from), and $\equiv$ (logical equivalence). An example of a predicate is $x = X + Y \vee x < 0$, which consists of a disjunction of two relational expressions, where a relation consists of two arithmetic expressions - a left hand side and a right hand side - and a relational operator.

The syntax for predicates (Boolean expressions) is defined as follows:

$$
\begin{array}{ll}
B \rightarrow B \wedge B & B \rightarrow \neg B \\
B \rightarrow B \vee B & B \rightarrow < \texttt{BoolConst} > \\
B \rightarrow B \Rightarrow B & B \rightarrow R \\
B \rightarrow B \Leftarrow B & B \rightarrow [B] \\
B \rightarrow B \equiv B &
\end{array}
$$

In the above grammar, the nonterminal R denotes a relation, for which the grammar will be given in subsection 2.2.2.

Note that the above grammar does not take into account the usual operator precedence (like $\wedge$ has higher priority than $\vee$) and is only presented for readability. The actual grammar used in the parser of the program is equivalent but much more complicated.

Moreover, note that we chose square parenthesis (i.e. '[' and ']') for Boolean expressions. We reserved the standard parentheses for arithmetic expressions (see subsection 2.2.3). In our view, choosing different parantheses for Boolean typed expressions and integer valued expressions makes complicated expressions easier to read, and it turns out that they are also easier to parse using a standard recursive descent parser.

In the Haskell program predicates are repsesented by the data type `BoolExpression`. It is defined as:

```
1  data BoolExpression = BoolExpression :&&: BoolExpression
2                     | BoolExpression :||: BoolExpression
3                     | BoolExpression :->: BoolExpression
4                     | BoolExpression :<-: BoolExpression
5                     | BoolExpression :==: BoolExpression
6                     | Not BoolExpression
7                     | BoolConst Bool
8                     | Compare Relation
```

Note that the smallest buildings blocks of boolean expressions are relations and `BoolConsts` (`True` and `False`).

2.2.2 Relations

The next building blocks in the expression hierarchy are relations. Their syntax is defined by the following grammar:

$$
\begin{array}{l}
R \rightarrow E < E \\
R \rightarrow E \leq E \\
R \rightarrow E = E \\
R \rightarrow E \geq E \\
R \rightarrow E > E \\
R \rightarrow E \neq E
\end{array}
$$

In the above grammar, the nonterminal E denotes an arithmetical integer valued expression, for which the grammar will be given in subsection 2.2.3. Implementation of relations as a Haskell data type is straightforward:

```
data Relation = ArithExpression :<: ArithExpression
              | ArithExpression :<=: ArithExpression
              | ArithExpression :=: ArithExpression
              | ArithExpression :>=: ArithExpression
              | ArithExpression :>: ArithExpression
              | ArithExpression :<>: ArithExpression
```

2.2.3 Arithmetic Expressions

Arithmetic expressions may include integer constants, variables, and arithmetic operators. Their syntax si defined by the following grammar:

$E \rightarrow E + E$	$E \rightarrow -E$
$E \rightarrow E - E$	$E \rightarrow <$ `Integer` $>$
$E \rightarrow E * E$	$E \rightarrow <$ `Const` $>$
$E \rightarrow E/E$	$E \rightarrow <$ `Variable` $>$
$E \rightarrow E\%E$	$E \rightarrow <$ `BoundVariable` $>$
$E \rightarrow E\hat{\ }E$	$E \rightarrow (E)$

Again, note that the above grammar does not take into account the usual operator precedence and is only presented for readability.

The Haskell data type `ArithExpression` is introduced to store arithmetic expressions. The `Const` data constructor is used for specification constants (capital letters in our examples), while `Variable` is used for variables (lower case letters). Note that there is also a data constructor for `BoundVariable`. Bound variables are variables like x and y in $\forall x, x < y \Rightarrow y > x$. Bound variables are not program variables, but are merely placeholders. They play a major role in the unification process that we will discuss later in chapter **??**. Our implementation of arithmetic expressions in Haskell:

```
data ArithExpression = ArithExpression :+: ArithExpression
                     | ArithExpression :-: ArithExpression
                     | ArithExpression :*: ArithExpression
                     | ArithExpression :/: ArithExpression
                     | ArithExpression :%: ArithExpression
                     | ArithExpression :^: ArithExpression
                     | UnaryMinus ArithExpression
                     | IntValue Integer
                     | Constant String
                     | Variable String
                     | BoundVariable String
```

2.2.4 Assignments

The last building block that we need to build Hoare triples are assignments. For this, we introduce the Haskell data type `Assignment`, consisting of a variable name (the assignee, i.e. a variable) and an arithmetic expression (the assigned value). The implementation looks like:

```
data Assignment = Assign String ArithExpression
```

2.3 Computation of the weakest precondition

Recall that the weakest precondition of a predicate and a series of assignments can be calculated using the recursive *wp*-rule:

$$wp(S, Q) = \begin{cases} Q, & \text{if } S = [\ ] \text{ (i.e. empty)}, \\ [E/x]Q, & \text{if } S = [x := E] \text{ (i.e. singleton list)}, \\ wp(S_0, wp(S_1, Q)), & \text{if } S = [S_0; S_1], \text{ where } S_0 \text{ is a single assignment} \end{cases} \tag{2.1}$$

The translation of this rule in Haskell is actually straightforward. We make a function `wps`[1] that takes two arguments: a series (list) of assignments and a postcondition. Of course, the returned value is the weakest precondition, hence a `BoolExpression`.

```
1 wps :: [Assignment] -> BoolExpression -> BoolExpression
2 wps [] e = e
3 wps (a:as) e = wp0 a (wps as e)
```

Here, the function `wp0` performs the second case of the *wp*-rule, i.e. the actual computation of the weakest precondition of a single assignment and a postcondition. Note that function *wp* satisfies the following rules:

$$\begin{aligned} wp(S, Q_0 \wedge Q_1) &\equiv wp(S, Q_0) \wedge wp(S, Q_1) \\ wp(S, Q_0 \vee Q_1) &\equiv wp(S, Q_0) \vee wp(S, Q_1) \\ wp(S, \neg Q) &\equiv \neg wp(S, Q) \\ wp(S, Q_0 \Rightarrow Q_1) &\equiv wp(S, Q_0) \Rightarrow wp(S, Q_1) \\ wp(S, Q_0 \Leftarrow Q_1) &\equiv wp(S, Q_0) \Leftarrow wp(S, Q_1) \\ wp(S, Q_0 \equiv Q_1) &\equiv wp(S, Q_0) \equiv wp(S, Q_1) \end{aligned}$$

The translation of these rules into Haskell is completely straightforward thanks to Haskell's pattern matching capabilities:

```
1 wp0 :: Assignment -> BoolExpression -> BoolExpression
2 wp0 ass (lhs :&&: rhs) = wp0 ass lhs :&&: wp0 ass rhs
3 wp0 ass (lhs :||: rhs) = wp0 ass lhs :||: wp0 ass rhs
4 ...
5 wp0 ass (Not exp) = Not (wp0 ass exp)
6 wp0 ass (Compare rel) = Compare (wpRel ass rel)
```

Note that the last line uses the function `wpRel` that computes the *wp* for relations (the atoms of the type `BoolExpression`). It has a very similar structure as the function `wp0`.

```
1 wpRel :: Assignment -> Relation -> Relation
2 wpRel ass (lhs :<: rhs)  = wpArithExp ass lhs :<:  wpArithExp ass rhs
3 wpRel ass (lhs :<=: rhs) = wpArithExp ass lhs :<=: wpArithExp ass rhs
4 ...
5 wpRel ass (lhs :<>: rhs) = wpArithExp ass lhs :<>: wpArithExp ass rhs
```

The function `wpArithExp` in the end performs substitutions. The function takes as its argument the assignment, and as its second argument an arithmetic expression. It returns the modified expression in which each occurence of the variable on the left hand side of the assignment has been replaced by the right hand side of the assignment (which is an expression).

[1]Note the plural naming `wps`, which is a Haskell style convention meaning that we apply the function to a list.

The structure of the function is again similar to the structure of `wp0`. The function recursively takes an arithmetic expression tree, finds all leaf nodes that contain the given variable and replaces them with the assigned value. Other nodes are left intact. Note that apart from Haskell's pattern matching capabilities it also allows 'copying' entire expressions (i.e. trees) in a single line. Note that this is in reality not a real copy. Under the hood this is just a reference (i.e. pointer) to an expression, which is valid implementation since Haskell has no side effects.

```
1  wpArithExp :: Assignment -> ArithExpression -> ArithExpression
2  wpArithExp ass (lhs :+: rhs) = wpArithExp ass lhs :+: wpArithExp ass rhs
3  wpArithExp ass (lhs :-: rhs) = wpArithExp ass lhs :-: wpArithExp ass rhs
4  ...
5  wpArithExp ass (UnaryMinus exp) = UnaryMinus (wpArithExp ass exp)
6  wpArithExp (Assign var e) (Variable str) =
7    if str == var
8    then e
9    else (Variable str)
10 wpArithExp ass exp = exp
```

The interesting part of this function are the lines 6–10. The case in line 6 is an assignment of the form `var:=e` and the symbolic expression (`str`) is a `Variable`. If the variable `var` equals the name in `str` then the expression is replaced by the expression `e`, otherwise it stays unmodified. Line 10 is a fall-through case, in which simply an unmodified expression is returned.

2.4 Parsing input

Now that we have a representation for predicates, relations, expressions, and the function `wps` we can compute the weakest precondition of a series of assignments and a post condition. As an example, we can compute

$$wp(x := x + y; y := x - y; x := x - y,\ x = Y \wedge y = X)$$

in Haskell as follows:

```
1  wps [Assign "x" ((Variable "x") :+: (Variable "y")),
2       Assign "y" ((Variable "x") :-: (Variable "y")),
3       Assign "x" ((Variable "x") :-: (Variable "y"))
4      ]
5      (Compare (Variable "x" :=: Constant "Y") :&&:
6       Compare (Variable "y" :=: Constant "X")
7      )
```

Clearly, this is a very clumsy notation. The system would be unnecessarily difficult to use if users have to specify Hoare triples in this style. What we need is a function `wp` that takes two strings as arguments: a series of assignments and a predicate. It parses the strings, converts them into the corresponding data types, and returns the weakest precondition in the system's representation.

```
1  wp :: String -> String -> BoolExpression
```

Using this function we get compute the precondition of the swap-example as follows (log from an interactive session):

```
1  *Main> wp "x:=x+y; y:=x-y; x:=x-y" "x=Y & y=X"
2  [(x+y)-((x+y)-y)=Y & (x+y)-y=X]
```

Note that the nicely formatted output is the result of making all the previously discussed data structures members of the class `Show` (see Haskell report, [11]). We implemented our own `show` routines which pretty print expressions.

The result of `wp` is semantically equivalent to, but syntactically quite different from the preffered expression `[x=X & y=Y]`. As mentioned before, the weakest precondition algorithm does syntactic substitution and nothing else, which makes it impossible to directly compare $wp(S, Q)$ with P (the precondition). There is no standard way of simplifying arithmetic expressions and we want to avoid hard-coding simplification rules in the system, so we came up with an algorithm that combines first-order unification and resolution to get around this problem (see chapter 3).

The implementation of the function `wp` that takes two strings as its input is actually quite simple. What we need are two parsers (see e.g. [1]). One is called `parseAssignments`, which converts a string into a series of assignments, i.e the list type `[Assignment]`. The other parser, named `parseBoolExpression` is used to convert a predicate into a `BoolExpression`. Once these conversion have taken place, we can simply apply the function `wps` to return the required result.

```
1 wp :: String -> String -> BoolExpression
2 wp assStr expStr =
3   let ass = parseAssignments assStr
4       exp = parseBoolExpression expStr
5   in wps ass exp
```

2.4.1 Parsing arithmetic expressions

In this section we discuss the implementation of the parser for arithmetic expression. Since parsing is not the (main) topic of this thesis, it will be discussed only briefly.

As demonstrated in section 2.2 every layer of the expression hierarchy is similar in composition to the previous layer, so taking a look at how arithmetic expressions are parsed should give the reader an idea of the parser as a whole. Hopefully this example will also clarify how operator precedence increases parser complexity.

The top level function for parsing arithmetic expressions is defined as follows:

```
1 parseArithExpression :: String -> ArithExpression
2 parseArithExpression str =
3   let (exp, (tok : tokens)) = parseE (lexer str)
4   in
5     case tok of
6       TokEnd -> exp
7       _ -> error ("Unused tokens: " ++ show (tok : tokens))
```

This function takes as its input a string containing an arithmetic expression, tokenizes it, and passes the resulting list of tokens to the function `parseE` which is responsible for the actual parsing of expressions. The function `lexer` converts the input string into a list of tokens, and will not be discussed further in this document. The function `parseE` expects on its input a list of `Token`s and returns an arithmetic expression and a list of remaining tokens (i.e. tokens that were not parsed). If the parsing of an expression was succesfull then the remaining list of tokens ought to be the singleton list `[TokEnd]` which contains the special token `TokEnd` which denotes the end of input.

Let us now have a look at the function `parseE`. It is the top level function for parsing arithmetic expressions. As noted before, the real grammar being used incorporates operator precedence and

differs from the grammar that we presented in section 2.2.3. The top level grammar rule that takes precedence into account looks like this:

$$E \to T\ E'$$
$$E' \to +\ T\ E'$$
$$E' \to -\ T\ E'$$
$$E' \to \varepsilon$$

Here T is a non-terminal that is used for accepting terms, and ε denotes the empty string. On its turn, the grammar rule for T contains a reference to the non-terminal F that is used for parsing factors, and so on. Not that the grammar is written in this format (using E', instead of $E \to T + T$) such that can be parsed with a recursive descent LL(1) parser (see [1]).

Apart from the toplevel parser function `parseE`, all other parsing functions have the type

```
parseXXX :: ArithExpression -> [Token] -> (ArithExpression, [Token])
```

The first argument is the arithmetic expression that was built up by the parsing process this far. For `parseE` this expression is non-existent, which explains why this argument is missing. The second argument is the list of input tokens that still need to be accepted. The return value is the (possibly) extended accepted expression, and a list of remaining tokens.

Using this format, the parsing routine `parseE` looks like

```
-- E -> T {E'}
parseE :: [Token] -> (ArithExpression, [Token])
parseE tokens =
  let (lhs, rest) = parseT tokens
  in parseE' lhs rest
```

Note that the function tries to parse a T (for term) before calling `parseE'`. This is needed for operator precedence: the expression returned by `parseT` is composed of nodes joined by higher precedence operators than that of E'.

For example, in tryting to parse the expression `"5*2+x"`, the expressions returned by `parseT` would be `((IntValue 5) :*: (IntValue 2))`, which would be taken as the left hand side of E' to make `(((IntValue 5) :*: (IntValue 2)) :+: (Variable "x"))`. Lower precedence operators plus and minus are handled by `parseE'`.

```
-- E' -> ("+" | "-") T {E'}
-- E' -> epsilon
parseE' :: ArithExpression -> [Token] -> (ArithExpression, [Token])
parseE' lhs (tok : tokens) =
  let (rhs, rest) = parseT tokens
  in
    case tok of
      TokPlus -> parseE' (lhs :+: rhs) rest
      TokMinus -> parseE' (lhs :-: rhs) rest
      _ -> (lhs, (tok : tokens))
```

The system deals with more operators than the ones shown here, therefore there are also more layers to the parser. However, the general idea is the same so there is little point in discussing these.

Eventually, arithmetic expression are built up from their atoms, which are variables and values, as well as expressions between brackets and negative terms (unary minus). These are parsed by the following function:

```
-- P -> <Var>
-- P -> <BoundVar>
-- P -> <Const>
-- P -> <Integer>
-- P -> "(" E ")"
-- P -> "-" T
parseP :: [Token] -> (ArithExpression, [Token])
parseP [] = error "Token expected"
parseP (tok : tokens) =
  case tok of
    (TokVariable str) -> (Variable str, tokens)
    (TokBoundVariable str) -> (BoundVariable str, tokens)
    (TokConstant str) -> (Constant str, tokens)
    (TokIntValue n) -> (IntValue n, tokens)
    TokLpar ->
      let (exp, (next : rest)) = parseE tokens
      in
        if next /= TokRpar
          then error "Missing right parenthesis"
          else (exp, rest)
    TokMinus ->
      let (exp, rest) = parseT tokens
      in (UnaryMinus exp, rest)
    _ -> error ("Syntax Error: " ++ show tok)
```

Chapter 3

Proof Generation

In this chapter we will discuss how the program tries to generate proofs for

$$P \Rightarrow wp(S, Q)$$

Our proof assistant makes use of a technique called *resolution* (see e.g. [16]), which is a technique that is very suited for automated proof generation. It does result in proofs, however, that may be counterintuitive, since it relies on proofs by contradiction. Instead of generating a direct proof (which is very hard to do automatically), the system tries to show that $P \wedge \neg wp(S, Q)$ is **false**.

Next to resolution, we also need a technique called *unification* (again, see e.g. [16]). For reasons of readability, the layout of this chapter is as follows. In section 3.1 we first discuss the resolution technique for propositional logic, even though we actually need it for first order logic. Later, in section 3.2 we discuss the unification technique. In section 3.3 we merge the technique into a proof system for first order logic (FOL).

3.1 Resolution (propositional logic)

Resolution is an inference technique that can be used to generate a proof by contradiction [16]. Before explaining the actual algorithm, it is helpful to first discuss an example that shows how it works.
Suppose we have te following *knowledge base*: the set of predicates that are known to be true:

$$KB = \{a \Rightarrow b,\ b \Rightarrow c \vee d,\ \neg c,\ \neg c \Rightarrow a\}$$

We want to prove the validity of d. A natural proof that shows that we can infer d from KB might follow these steps:

1. from $\neg c$ and $\neg c \Rightarrow a$ we can infer a
2. from a and $a \Rightarrow b$ we can infer b
3. from b and $b \Rightarrow c \vee d$ we can infer $c \vee d$
4. from $c \vee d$ and $\neg c$ we can infer d.

Automatic generation of natural proofs like these is very difficult, since it requires insight of the problem. Similar to simplification of arithmetic expressions, it proves difficult for a computer

since it does not know in which direction it should search (should it expanding or shrink expressions?). However, there is one key difference between sentences in propositional logic (KB in our example) and arithmetic expressions. Propositional sentences have a *normal form* which makes comparison possible: the *conjunctive normal form (CNF)*. A proposition is in CNF if it consists of a conjunction of disjunctions, where the literals are propositional symbols and/or negation of propositional symbols. We can convert KB in an equivalent form that is in CNF:

$$KB = \{\neg a \vee b,\ \neg b \vee c \vee d,\ \neg c,\ c \vee a\}$$

The elements of a knowledge base in CNF format are called *clauses*.

The main reason for converting the knowledge base in a set of clauses is that we can apply the so-called *resolution rule*, which simply states that from the clauses $a \vee b$ and $\neg a \vee c$, we can conclude the clause $b \vee c$.

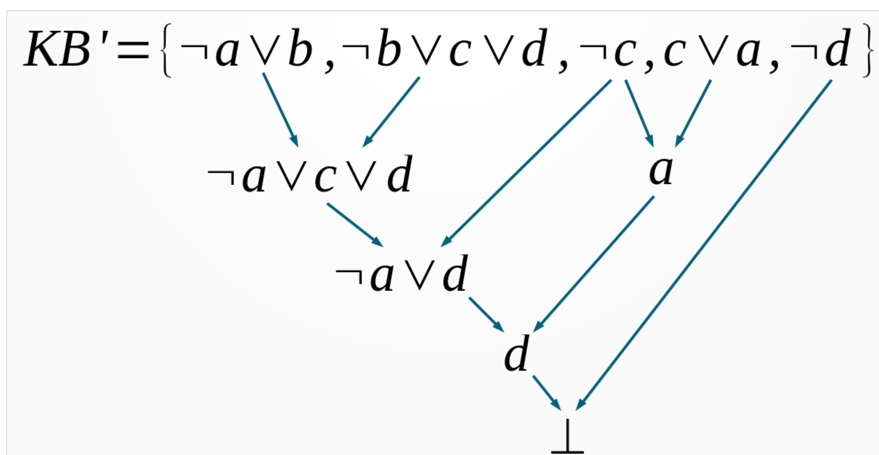
The rule is clearly valid because if a is false then b must be true, and if a is true then c must be true. Either way, one of b or c is true, regardless of a. So, the resolution rule can be formulated as:

$$\frac{\alpha \vee p \qquad \beta \vee \neg p}{\alpha \vee \beta} \tag{3.1}$$

By iterative application of this rule, we can generate automatically the required proof. We generate a proof by contradiction by first constructing a new knowledge base KB' which is constructed by augmenting the orginial KB with the negation of the goal in CNF (in this case $\neg d$), and try to infer **false** (i.e. the empty clause) using only the resolution rule. A sketch of this proof is given in the following steps:

1. from the clauses $\neg c$ and $c \vee a$ we infer the new clause a.
2. from the clauses a and $\neg a \vee b$ we infer the new clause b.
3. from the clauses b and $\neg b \vee c \vee d$ we infer the new clause $c \vee d$.
4. from the clauses $c \vee d$ and $\neg c$ we infer the new clause d.
5. from the clauses d and $\neg d$ we infer the empty clause (i.e. **false**).

This process is shown in the proof tree below. Notice how the clauses acquired are not exactly the same as in the steps above but the end result is. There are often many paths leading to the same conclusion.



3.1.1 Resolution algorithm

The proof sketched above can be generated automatically using a simple algorithm, which is called the *resolution algorithm* (see [16]). Its input is the knowledge base KB in CNF augmented with negation of the goal in CNF format. Its output is a simple boolean verdict. The algorithm returns **true** if and only if a proof can be constructed, i.e. the augmented KB is not satisfiable.

The algorithm iteratively takes pairs of clauses (α, β) and computes, using the function call $resolvents(\alpha, \beta)$, the set of resolvents from α and β using only the resolution rule. Note that there could be more than a single resolvent resulting from a pair of clauses, therefore *resolvents* returns a set. Note that this set is empty if α and β do not have literals which are each other's negation. Also note that this set may contain the empty clause (denoted as $\bot$), which means that we completed the proof. Newly inferred clauses are added to the knowlegde base on the fly. If no more new inferences can be made, i.e. the algorithm performed an exhaustive search, then the algorithm stops with the conclusion that a proof cannot be constructed. The following imperative pseudo-code fragment sketches the resolution algorithm:

input: KB in CNF, contains the negation of the goal.
output: the boolean variable *proven* is **true** iff KB is not satisfiable

$clauses := KB$;
$exhaust :=$ **false**;
$proven :=$ **false**;
while $\neg proven \wedge \neg exhaust$ **do**
  $inferences := \emptyset$;
  forall $\alpha \in clauses$ **do**
    forall $\beta \in (clauses \backslash \{\alpha\})$ **do**
      $inferences := inferences \cup resolvents(\alpha, \beta)$;
    end forall;
  end forall;
  $exhaust := inferences \subseteq clauses$;
  $clauses := clauses \cup inferences$;
  $proven := \emptyset \in inferences$;
end while;

Note that the above algorithm is coded in a *breadth first search (BFS)* fashion: first we compute all direct inferences by pairing all clauses from KB. Next, we compute 'second level' inferences, and so on. It is also possible to use a *depth first search* implementation, but we prefer the BFS style of computation since it always finds the shortest proof (if it exists).

3.1.2 Conjunctive Normal Form

Of course, for the resolution algorithm to be applicable, we need an algorithm to convert the knowledge base into an equivalent knowledge base in CNF. Fortunately, any sentence in propositional logic can be converted to CNF using a four-step process. Let us have a look at the steps of this process using the knowledge base $KB = \{a \Rightarrow (b \wedge \neg(c \vee d))\}$:

1. eliminate $\equiv$, replacing $a \equiv b$ with $(a \Rightarrow b) \wedge (b \Rightarrow a)$
   $KB = \{a \Rightarrow (b \wedge \neg(c \vee d))\}$ (no change)

2. eliminate $\Rightarrow$, replacing $a \Rightarrow b$ with $\neg a \vee b$
   $KB = \{\neg a \vee (b \wedge \neg(c \vee d))\}$

3. move $\neg$ inwards, eliminating double negations and applying De Morgan's laws
   $KB = \{\neg a \vee (b \wedge \neg c \wedge \neg d)\}$

4. apply distributivity law, distributing $\vee$ over $\wedge$ wherever possible
   $KB = \{\neg a \vee b, \neg a \vee \neg c, \neg a \vee \neg d\}$

3.2 Unification

Unfortunately, the basic resolution algorithm cannot be used in the proof assistant, since we deal with *First-Order Logic (FOL)* instead of propositional logic. FOL (or *predicate logic*) uses quantified variables and allows the use of sentences that contain variables. In the proof assistant we only allow bound variables for $\forall$-predicates. For example, the predicate $(\forall x :: x > 0 \Rightarrow x \geq 1)$ would be expressed in our system by the knowledge base rule `#x>0 -> #x>=1`. Here the symbol `#` denotes that the identifier directly following it is a $\forall$-quantified variable.

Clearly, as an example, we should be able to infer that `x+y>=1` if the knowledge base contains the rule `#x>0 -> #x>=1` and we know that `x+y>0`. However, resolution cannot infer this without an algorithm that matches the quantified expression `#x>0` with the expression `x+y>0`.

The algorithm that does exactly this matching is called the *unification algorithm*. Unification deals with the problem of finding a substitution for quantified variables that makes two expressions equal. This substitution is called a *unifier*. So, a unifier is a mapping that assigns an expression to some (or all) bound variables such that the two expressions match. The *most general unifier (MGU)* is the unifier that performs the least number of substitutions, and is the one we are interested in since it gives the proof assistant maximal freedom to find suitable unifiers later on in the proofing process.

As an example, let us try to unify the expressions `#a+#b=#b+#a` (commutativity of $+$) and `2*y+#x=#x+2*y`. Clearly, these expressions have the most general unifier `[#a/2*y, #b/#x]`, where the notation `#a/2*y` denotes that the bound variable `#a` must be replaced by the expression `2*y`. Note that the general unifier is unique, apart from renaming of bound variables. So, `[#a/2*y, #x/#b]` is also a valid most general unifier, which is perfectly fine since the name of a bound variable is irrelevant. Note that `[#a/2*y, #b/p, #x/p]` would also be a unifier, but it is (deliberately) not the most general one. It introduces the variable `p` without any reason.

Of course, it is not always possible to unify two given expressions. For example, `#a+#b=0` cannot be unified with `x*y=0`, simple because the operators mismatch. In that case the unifier is said to be empty.

3.2.1 Implementation

We implemented unifiers in Haskell as the data type `Unifier` which is a list (actually a set implemented as a list) of pairs. The first element of the pair is a `String`, denoting the name of the bound variable. The second is an `ArithExpression`, which is the expression to be substituted.

```
1 type Substitution = (String, ArithExpression)
2 type Unifier =  [Substitution]
```

Our implementation of unification is based on the version presented in [16] (page 278), which on its turn is based on the references [15] and [13]. Translating the algorithm in Haskell is actually not very difficult because the algorithm is fully recursive and does not use loops.

The base step of the recursion is to find a unifier for a single bound variable and an arithmetic expression. This is done by the function `unifyVar`. This function expects three arguments, and produces a unifier if one exists. The first argument (a `String`) is the name of the bound variable

to be unified. The second argument is the arithmetic expression. The third argument is the unifier built so far during the recursive process. The type of this argument is a `Maybe Unifier`. Note that we made use of Haskell's `Maybe t` construct, which returns either the value `Nothing` or `Just a` where `a` is of type `t`. This is very handy, since we can model failure (i.e. no unifier found) by `Nothing`, while an actual unifier `theta` can be passed as `Just theta`. The function `unifyVar` returns a `Maybe Unifier`.

At first sight, unifiying a variable `#a` with some expression `e` is simple. However, we need to be careful since `#a/e` is not always the right unifier. The reason is that we are in the base case of a larger recursion and that we might have found already a substitution for `#a` earlier in the process. So, first we need to check whether `#a` is in the unifier built-up this far. This is perfomed by a function named `findUnification`. If this function finds a substitution `e'` for `#a`, then we should actually go back to the top level of the recursion (which is the function `mguAexp`) which and try to unify `e'` and `e` instead.

But even if `#a` is not in the built-up unifier, then we still need to be careful. The expression `e` itself may consist of only a bound variable (so we try to unify `#a` with something like `#b`), which on its turn might again be bound via the built-up unifier. If that is the case, we also need to jump back to the main unification function `mguAexp` with `e` replaced by its unifiying expression.

Last but not least, if none of the above situations are the case at hand, we still need to perform the function `occurCheck` which checks whether a bound variable occurs in an arithmetic expression or not in order to avoid infinite recursion. For example, `#a` cannot be unified with the expression `#a+#a`, simple because `#a` occurs in the expression itself. In that case, the returned unifier should be `Nothing` (i.e. failure). If `occurCheck` returns **false**, the we extend the built-up unifier with the expected binding `#a/e` and return it.
The Haskell implementation of `unifyVar` is given below.

```
1  unifyVar :: String -> ArithExpression -> Maybe Unifier -> Maybe Unifier
2  unifyVar var exp (Just theta) =
3   if findUnification var theta
4   then mguAexp (getUnification var theta) exp (Just theta)
5   else
6     if (isBoundVar exp) && (findUnification (getBoundVarName exp) theta)
7     then mguAexp (BoundVariable var)
8                  (getUnification (getBoundVarName exp) theta) (Just theta)
9     else
10      if occurCheck var exp
11      then Nothing
12          else Just ((var, exp):theta)
```

The top level recursion function is the function `mguAexp`. It has three arguments, of which the first two are arithmetic expressions that it tries to unify. The third parameter is, like in `unifyVar`, the unifier built up this far, which we call `theta` in the remaining discussion.

If `theta=Nothing`, then we simply return `Nothing` since the unification failed already earlier on in the recursive process. If we try to unify a variable with an expression, then we use the function `unifyVar` to compute the unifier, and return it.

If we try to unify two expressions that both consist of a single specification constant, then they are only unifiable if they are the same specification constant, otherwise we return `Nothing` (i.e. failure). The same holds for the unification of two constant integer values, and program variables.

For compound expressions involving binary arithmetic operators, the operators must match, of course. Moreover, we recursively need to unify the corresponding left and right hand sides.
This analysis leads to the following Haskell code fragment:

```
mguAexp :: ArithExpression -> ArithExpression -> Maybe Unifier
           -> Maybe Unifier
mguAexp e0 e1 Nothing = Nothing
mguAexp (BoundVariable x) (BoundVariable y) theta =
  if (x == y) then theta else unifyVar x (BoundVariable y) theta
mguAexp (BoundVariable x) y theta = unifyVar x y theta
mguAexp x (BoundVariable y) theta = unifyVar y x theta
mguAexp (Constant x) (Constant y) theta = if x==y then theta else Nothing
mguAexp (IntValue x) (IntValue y) theta = if x==y then theta else Nothing
mguAexp (Variable x) (Variable y) theta = if x==y then theta else Nothing
mguAexp (UnaryMinus e0) (UnaryMinus e1) theta = mguAexp e0 e1 theta
mguAexp (l0 :+: r0) (l1 :+: r1) theta=mguAexp r0 r1 (mguAexp l0 l1 theta)
mguAexp (l0 :-: r0) (l1 :-: r1) theta=mguAexp r0 r1 (mguAexp l0 l1 theta)
mguAexp (l0 :*: r0) (l1 :*: r1) theta=mguAexp r0 r1 (mguAexp l0 l1 theta)
mguAexp (l0 :/: r0) (l1 :/: r1) theta=mguAexp r0 r1 (mguAexp l0 l1 theta)
mguAexp (l0 :%: r0) (l1 :%: r1) theta=mguAexp r0 r1 (mguAexp l0 l1 theta)
mguAexp (l0 :^: r0) (l1 :^: r1) theta=mguAexp r0 r1 (mguAexp l0 l1 theta)
mguAexp e0 e1 theta = Nothing
```

On top of the function `mguAexp` we build unification for `Relation`s and `BoolExpression`s. The code for these is straightforward.

```
mguRel :: Relation -> Relation -> Maybe Unifier -> Maybe Unifier
mguRel e0 e1 Nothing = Nothing
mguRel (l0 :<: l1) (r0 :<: r1) theta  =mguAexp l1 r1 (mguAexp l0 r0 theta)
mguRel (l0 :<=: l1) (r0 :<=: r1) theta=mguAexp l1 r1 (mguAexp l0 r0 theta)
mguRel (l0 :=: l1) (r0 :=: r1) theta  =mguAexp l1 r1 (mguAexp l0 r0 theta)
mguRel (l0 :>=: l1) (r0 :>=: r1) theta=mguAexp l1 r1 (mguAexp l0 r0 theta)
mguRel (l0 :>: l1) (r0 :>: r1) theta  =mguAexp l1 r1 (mguAexp l0 r0 theta)
mguRel (l0 :<>: l1) (r0 :<>: r1) theta=mguAexp l1 r1 (mguAexp l0 r0 theta)
mguRel _ _ _ = Nothing


mguBexp :: BoolExpression -> BoolExpression -> Maybe Unifier
           -> Maybe Unifier
mguBexp e0 e1 Nothing = Nothing
mguBexp (l0 :&&: l1) (r0 :&&: r1) theta=mguBexp l1 r1 (mguBexp l0 r0 theta)
mguBexp (l0 :||: l1) (r0 :||: r1) theta=mguBexp l1 r1 (mguBexp l0 r0 theta)
mguBexp (l0 :->: l1) (r0 :->: r1) theta=mguBexp l1 r1 (mguBexp l0 r0 theta)
mguBexp (l0 :<-: l1) (r0 :<-: r1) theta=mguBexp l1 r1 (mguBexp l0 r0 theta)
mguBexp (l0 :==: l1) (r0 :==: r1) theta=mguBexp l1 r1 (mguBexp l0 r0 theta)
mguBexp (Not e0) (Not e1) theta = mguBexp e0 e1 theta
mguBexp (BoolConst e0) (BoolConst e1) theta =
   if e0==e1 then theta else Nothing
mguBexp (Compare e0) (Compare e1) theta = mguRel e0 e1 theta
mguBexp _ _ _ = Nothing
```

3.3 Resolution and Unification for FOL

Now that we have building blocks that can perform unification, we can now build a resolution algorithm that is suitable for predicate calculus (i.e. first order logic, FOL). Just like in section

3.1 we need to convert the augmented knowledge base in CNF using the procedure described in subsection 3.1.2. However, since we deal with predicate calculus, the clauses consist of reduced `BoolExpressions`s. The expressions are reduced in the sense that they can only be a `Relation` or the negation of a `Relation`. The clauses are represented using the type `Clauses`, which is a list of boolean expressions.

```
type Clauses = [BoolExpression]
```

As an example, here are three rules from the knowledge base of the system.

`#a=#b -> #b=#a,  #a*#b=#c -> #b*#a=#c,  #a*#b=#a*#c & #a<>0 -> #b=#c`

After conversion of the knowledgebase into CNF and stored as a list of `Clauses`, we get

```
[[[~[#a=#b],#b=#a]],
 [[~[#a*#b=#c],#b*#a=#c]],
 [[~[#a*#b=#a*#c],~[#a<>0],#b=#c]]
]
```

Now that we have an augmented FOL knowledge base (i.e. a list of `Clauses`), we can perform resolution using the following function which has a similar structure as the pseudocode given in subsection 3.1.1.

```
resolution :: [Clauses] -> Bool
resolution [] = False
resolution cs =
  let resolvents = nub (resolveClauses cs)
  in
    if [] `elem` resolvents then True
    else if subsetOf resolvents cs then False
    else resolution (resolvents ++ cs)
```

The implementation is recursive and makes use of the helper function `resolveClauses` which generates all possible new clauses that can be generated from the knowledge base using the resolution rule and unification. If an empty clause is found then we found the required contradiction, and hence the proof is completed. If no new clauses are generated (i.e. the newly generated clauses are a subset of the set of clauses from a previous iteration), the search process is exhausted and the function returns `False`. Otherwise it will call itself recursively with an extended knowledge base as its argument.

We now have a look at the helper function `resolveClauses`. It takes the `Clauses` of the knowledge base one by one and tries to unify them with the remaining clauses using the helper function `resolveClauses'`.

```
resolveClauses :: [Clauses] -> [Clauses]
resolveClauses [] = []
resolveClauses (c:cs) = resolveClauses' c cs ++ resolveClauses cs
```

The function `resolveClauses'` that takes as its first argument a clause and as its second argument a set (actually a list) of clauses. It tries to match the first argument with all the clauses from the second using the resolution rule and unification.

In the following discussion let `a` be the first argument of this function (i.e. the clause that we pair with all others), and let `b` be one of the other clauses. The function first concatenates the two clauses to produce the list `a++b` and drops from it items which are duplicates. Duplicates can occur if `a` and `b` have disjuncts in common. Let `c` be the result of this filtered concatenation.

Next, using a helper function `findOppositeMGUs` it tries to find in `c` two literals that are complements using unification. If a unifier is found, then the complemented lietarls are dropped and the function `applyMGU` is applied to the remaining part of `c`.
The code of this function is given in the following fragment.

```
resolveClauses' :: Clauses -> [Clauses] -> [Clauses]
resolveClauses' _ [] = []
resolveClauses' a (b:bs) =
  let new  = dropIdenticals [a ++ b]
      new' = applyMGUs (findOppositeMGUs new) new
      res  = dropOpposites new'
  in
    if res == new
    then resolveClauses' a bs
    else res ++ resolveClauses' a bs
```

3.4 Example: the swap program fragment

Let us use the swap fragment to demonstrate the complete proof generation process. Recall that a Hoare triple's correctness can be verified by the proof rule 1.5. In this case the Hoare triple is

$$\{x = X \wedge y = Y\}\, x := x + y; y := x - y; x := x - y\, \{x = Y \wedge y = X\}$$

So, the automated proof assistant needs to prove

$$(x = X \wedge y = Y) \Rightarrow wp(x := x + y; y := x - y; x := x - y,\ x = Y \wedge y = X)$$

expanding wp as mentioned in section 1.4:

$$\{x = X \wedge y = Y\} \Rightarrow \{(x + y) - ((x + y) - y) = Y \wedge (x + y) - y = X\}$$

As explained in section 3.1, the resolution algorithm works by merging the negation of its input clauses with the knowledge base and looking for contradictions in the resulting extended knowledge base. Moreover, every predicate that goes into the knowledge base will be converted to CNF. The clauses after negation and in CNF look like this:

$$x = X \wedge y = Y \wedge ((x + y) - ((x + y) - y) \neq Y \vee (x + y) - y \neq X)$$

Let us first consider what happens when the original knowledge base is empty, so $KB = \{\}$. Adding the clauses acquired from the proof rule, we get

$$KB' = \{x = X,\ y = Y,\ (x + y) - ((x + y) - y) \neq Y \vee (x + y) - y \neq X\}$$

It is easy for the reader to see that this leads to a contradiction, but unfortunately the system cannot do arithmetic simplification so it will never find a proof for this knowledge base using only resolution and unification. Somehow we have to make it "understand" that $(x+y)-((x+y)-y) \neq Y$ is equivalent to $y \neq Y$. As a general rule, we can add $\#a - (\#a - \#b) \neq \#c \Rightarrow \#b \neq \#c$ to our knowledge base. If we do the same for $(x + y) - y \neq X$ and convert these clauses to CNF we end up with

$$KB = \{\#a - (\#a - \#b) = \#c \vee \#b \neq \#c,\ (\#a + \#b) - \#b = \#c \vee \#a \neq \#c\}$$

Now, we construct the augmented knowledge base

$$
\begin{aligned}
KB' = \{x &= X, && (3.2)\\
y &= Y, && (3.3)\\
&(x+y) - ((x+y) - y) \neq Y \vee (x+y) - y \neq X, && (3.4)\\
&\#a - (\#a - \#b) = \#c \vee \#b \neq \#c, && (3.5)\\
&(\#a + \#b) - \#b = \#c \vee \#a \neq \#c\} && (3.6)
\end{aligned}
$$

One possible path to a proof using clauses from this KB' is:

1. apply unifier $\theta = \{\#a/x,\ \#b/y,\ \#c/X\}$ to 3.6 to get

$$(x+y) - y = X \vee x \neq X \tag{3.7}$$

2. apply resolution rule to 3.2 and 3.7 to get

$$(x+y) - y = X \tag{3.8}$$

3. apply resolution rule to 3.4 and 3.8 to get

$$(x+y) - ((x+y) - y) \neq Y \tag{3.9}$$

4. apply unifier $\theta = \{\#a/x+y,\ \#b/y,\ \#c/Y\}$ to 3.5 to get

$$(x+y) - ((x+y) - y) = Y \vee y \neq Y \tag{3.10}$$

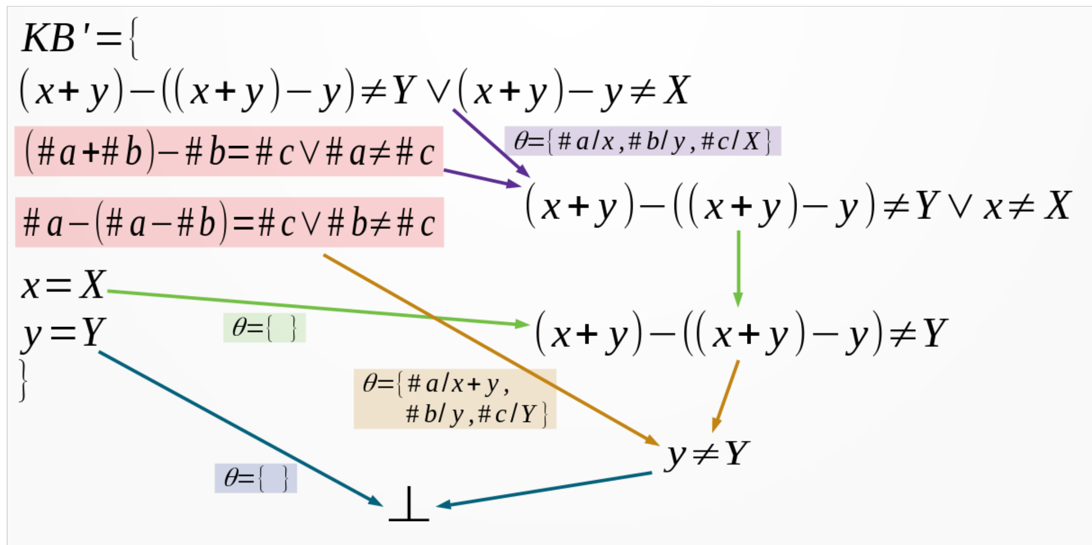
5. apply resolution rule to 3.9 and 3.10 to get

$$y \neq Y \tag{3.11}$$

6. apply resolution rule to 3.3 and 3.11 to get

$$\perp \tag{3.12}$$

It took only a few steps to infer the correctness of the swap fragment. The system numbers all clauses in the knowledge base similarly to the example above. This is important because it relies on recursion, and once it finds a contradiction it needs to backtrack the steps that took it there. A graphical representation of this process:



It is clear that the knowledge base is crucial in this process. Ideally the knowledge base would cover most (arithmetic) simplification rules so adding special clauses for a new problem would not be needed. In practice this means, among others, that we need many standard rules for the arithmetic operators like associativity, distributivity, unit elements, etc.

We do not discuss the construction of the knowledge base any further. In the construction of the knowledge base we take the attitude that it must be extended by the user of the proof assistant. Every time that the system is not able to find a proof, while it is clear to the user that a proof must exist, then we leave it up to the user to extend the knowledge base with valid rules that make a resolution proof possible. Note that it is never wrong to have rules in the knowledge bases that are not used by the proof assistant, so in practice this means that the knowledge base gets extended only (and never shrinks).

Chapter 4

Conclusion and future work

We have succeeded in our goal of creating a proof generation system that is easy to use with only a basic understanding of Hoare triples. Additionally, the use of a knowledge base allows the system to be extensible without modifications to its implementation. However, at this point the system might not be useful for real-world applications, mainly because our programming language lacks control structures and other basic data types.

4.1 Future work

4.1.1 Extension of the programming language

As mentioned above, the programming language used by the system could be closer to real languages in that it should support:

- other basic data types other than integers (booleans, floating point numbers, etc.)
- conditional statements
- loops
- arrays

The implementation of conditional statements should not cause significant difficulty: it is possible to generate a standalone proof of each branch of the conditional with the guard added to the precondition. An additional rule for proving loops was introduced in section 2.1, which makes proofs for loops fairly straightforward to implement as well.

4.1.2 Improvements to proof generation

First of all, our proof generation algorithms are far from perfectly optimized. In fact, the issues of memory consumption and computational complexity were completely disregarded. This does not mean that the system is horribly inefficient but that there are many opportunities to reduce its memory footprint and execution time.

A useful feature to implement would be substitution of subexpressions. Suppose we are trying to verify

$$x = x \times (X + Y) \wedge X + Y = 0 \Rightarrow x = x \times 0$$

A normal proof would simply use substitution of the value 0 for the subexpression $X + Y$ in the first conjunct of the premise. However, resolution does not perform substitutions like those. It only involves pattern matching. For this substitution to work we could add the following rule to the knowledge base:

$$\#a = \#b \times \#c \wedge \#c = \#d \Rightarrow \#a = \#b \times \#d$$

While this would work, as soon as we increase the depth of the expression tree without changing its result e.g.

$$x = x \times ((X - Z) + Y) \wedge X - Z = X \wedge X + Y = 0$$

we have to add yet another rule to the knowledge base and so on. Ideally this issue should be resolved in our proof generation algorithm instead, but we felt this lies outside the scope of this thesis.

4.1.3 Experimental improvements

And finally, a brief introduction to the idea of finding counterexamples. The system is capable of substituting (random) values for variables in a Hoare triple. In this way, the incorrectness of programs can be proven - trying a few hundred different permutations for all variables involved and evaluating the resulting arithmetic expressions might result in a contradiction at some point. If such a case is found it is considered a proof by counterexample, since finding even one set of values for which the program is not correct means it cannot be correct in general. This was one of the first methods we experimented with but decided to focus on resolution/unification eventually, leaving this halfway implemented.

Bibliography

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: principles, techniques, and tools.* Addison-Wesley, Reading, MA, 1986.

[2] Yves Bertot and Pierre Castran. *Interactive Theorem Proving and Program Development: Coq'Art The Calculus of Inductive Constructions.* Springer Publishing Company Incorporated, 1st edition, 2010.

[3] R.S. Boyer, M. Kaufmann, and J.S. Moore. The boyer-moore theorem prover and its interactive enhancement. *Computers & Mathematics with Applications*, 29(2):27 – 62, 1995.

[4] Edsger W. Dijkstra and W. H. J. Feijen. *A method of programming.* Addison-Wesley, 1988.

[5] Edsger W. Dijkstra and Carel S. Scholten. *Predicate Calculus and Program Semantics.* Springer-Verlag, Berlin, Heidelberg, 1990.

[6] Azadeh Farzan, Matthias Heizmann, Jochen Hoenicke, Zachary Kincaid, and Andreas Podelski. Automated program verification. In Adrian-Horia Dediu, Enrico Formenti, Carlos Martín-Vide, and Bianca Truthe, editors, *Language and Automata Theory and Applications*, pages 25–46, Cham, 2015. Springer International Publishing.

[7] Robert W Floyd. Assigning meanings to programs. *Mathematical aspects of Computer Science*, 19(19-32):1, 1967.

[8] Georges Gonthier. Formal proof—the four-color theorem. *Notices of the AMS*, 55(11), 2008.

[9] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic.* Cambridge University Press, New York, NY, USA, 1993.

[10] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, October 1969.

[11] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries – The Revised Report.* Cambridge University Press, Cambridge, England, 2003.

[12] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[13] Alberto Martelli and Ugo Montanari. An efficient unification algorithm. *ACM Trans. Program. Lang. Syst.*, 4(2):258–282, April 1982.

[14] Tobias Nipkow, Markus Wenzel, and Lawrence C. Paulson. *Isabelle/HOL: A Proof Assistant for Higher-order Logic.* Springer-Verlag, Berlin, Heidelberg, 2002.

[15] J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, January 1965.

[16] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach.* Series in Artificial Intelligence. Prentice Hall, Upper Saddle River, NJ, third edition, 2010.

[17] Freek Wiedijk. Formal proof - Getting started. *Notices Am. Math. Soc.*, 55(11):1408–1414, 2008.