

UNIVERSITY OF GRONINGEN

BACHELOR'S PROJECT

Automatic Verification of Annotated Sequential Imperative Programs

Author

Alinda Harmanny

Supervisors

dr. A. Meijster

Prof. dr. G.R. Renardel de

Lavalette

Contents

1	Introduction	5
1.1	Goal of the project	5
1.2	Literature Overview	5
1.3	Programming Language and its Semantics	6
1.3.1	Aladin's Programming Language	6
1.3.2	Program States	6
1.3.3	Hoare Triples, and Specification Constants	7
1.3.4	Annotations, and Weakest Preconditions	7
1.4	Resolution and Unification	9
1.4.1	Resolution in Propositional Logic	9
1.4.2	Conjunctive Normal Form (CNF)	11
1.4.3	Resolution in First Order Logic (FOL)	11
1.4.4	Unification	12
1.4.5	Semi-Decidability	15
1.5	Example proof	15
2	Parsing and Storing Expressions	17
2.1	Numerical Expressions	17
2.2	Boolean Expressions	18
2.3	Parsing Expressions	20
3	Finding Counterexamples	25
3.1	Parsing and Storing Programs	25
3.2	Reducing Expressions	26
3.2.1	Simplifying and Reducing Numeric Expressions	27
3.2.2	Simplifying and Reducing Boolean Expressions	28
3.3	Solving a CSP	30
3.4	Running Program Simulations	32
3.5	Example Runs	34
4	Unification	37
4.1	Unification Algorithm	37
4.2	Applying a Unifier	40
4.3	Substitution of Equalities	41
5	Resolution	45
5.1	Resolution Algorithm	45
5.2	Resolving Two Clauses	48
5.3	Filtering and Reducing Clauses	49
6	Verification of an Annotated Program	53
6.1	Verifying Correctness	53
6.2	The Implementation of <code>proofImplication</code>	56
6.3	Computing Weakest Preconditions of Assignments	57

6.4	Some Examples of Verification Runs	58
7	Conclusion and Future Work	61

Chapter 1

Introduction

Writing error-free programs is a daunting task, and many programs contain several bugs that were not detected by the programmer. These undetected errors may cause a program to behave unexpectedly, crash, or produce incorrect results. To prevent such errors, a programmer may decide to prove the *correctness* of a program. Here correctness means that the program satisfies its *specification*, and is error-free. However, producing these proofs is on itself an error-prone process.

This document is the report of a Bsc. project which has the aim to make a program that takes a correctness proof as its input, and verifies that no mistakes have been made in the proof.

This chapter presents the goal of the project, and some concepts and algorithms that should be understood by the reader in order to be able to understand the technical details of the implementation of the verification tool. Moreover, a literature overview is given.

1.1 Goal of the project

The goal of this project is to write a program that takes a correctness proof as its input, and verifies that no mistakes have been made in the proof. We named the program *Aladin*, which is an anagram of the letters of the first name of the author, and also a reference to the famous tale from '*The Book of One Thousand and One Nights*' in which a genie (ghost from the lamp) serves his master.

Aladin takes as its input a annotated correctness proof in the notation that we use in the course Program Correctness at the Rijksuniversiteit Groningen. This notation will be presented later in this chapter in the section *Programming Language and its Semantics*.

We defined a set of requirements that Aladin needs to meet:

1. Aladin should output "ACCEPTED" when no mistakes are detected in the input proof.
2. Aladin should output "ERROR" with a counterexample when an error is detected in the input proof.
3. Aladin should output "UNDECIDED" when it can not verify that the input proof is correct, nor could it find a counterexample.

1.2 Literature Overview

Program correctness has a long history, which goes back to Floyd, Hoare, Gries and Dijkstra. Robert W. Floyd introduced a basis for the formal definition of the meaning of a program [11]. After that, Tony Hoare introduced a new notation [13] which we now know as Hoare Triples. Edsger Dijkstra introduced proofs using the concept of a *weakest precondition* [10]. Hoare Triples and weakest preconditions are strongly related, and are explained in more detail later in this chapter.

Nowadays, several proof assistants are publicly available, of which the following ones are well-known [12]:

- Coq [1]. Coq is an interactive proof assistant. Coq implements a program specification and mathematical higher-level language called Gallina.
- HOL [2]. HOL is an interactive proof assistant for higher-order logic.
- Isabelle [3]. Isabelle is another interactive proof assistant for higher-order logic. Isabelle uses resolution and unification which we intend to use in Aladin too.
- NQTHM [8]. NQTHM, also known as the Boyer-Moore theorem prover is an interactive proof assistant. It uses first order logic for its proofs and the syntax of the logic resembles that of the LISP programming language.
- PVS [4]. PVS is an interactive theorem prover. PVS uses a specification language based on higher-order logic.
- Z3 [5]. Z3 is a theorem prover from Microsoft Research. Z3 uses predicate and propositional logic.

Although these proof assistants are already available, we will develop a new one. Most of these existing assistants are built to generate mathematical proofs in an interactive fashion with the user, which is quite different from detecting flaws in a given annotated program. Furthermore, each of these proof assistants utilize their own specification language. These specification languages are quite different from the annotated programs that we want to process with Aladin. In our view, it is quite user unfriendly to impose on the user the burden to translate his/her proof into one of those specification languages. Besides it takes a lot of time to learn such a language, and the task to translate a program with its annotation into one of those specification languages is quite error-prone. It is much more convenient to have a tool which accepts a program with its annotation similar to the way annotations are written in the Program Correctness course.

1.3 Programming Language and its Semantics

1.3.1 Aladin's Programming Language

The programming language that is accepted by Aladin is very simple, and resembles the notation of the Pascal programming language. It is not our intention to implement a full-blown programming language with multiple types, functions, and procedures. Aladin accepts only one variable type, being (infinite precision) integers. An Aladin program consists of declarations, annotations (predicates) and a few basic programming constructs: the skip command, an assignment command, an if-then-else construct, a while loop construct, and sequential composition of commands. The operational meaning of these constructs is as follows:

- **skip** is an empty command, it does no do anything.
- $x := E$ is an assignment command. Here x is a program variable and E is an expression in terms of the declared variables and constants. First E is evaluated and then this value is assigned to the variable x .
- $S; T$ is a sequential composition. First S is executed, followed by the execution of T .
- **if B then S else T end** is a conditional command. The Boolean expression B is called the *guard* (which is an expression in the program variables and constants) and the commands S and T are called the *branches*. If B evaluates to **true** then the branch S is executed, otherwise the branch T is executed.
- **while B do S end** is a repetition construct. Here B is called the *guard* (like in the if-then-else construct), and S is called the *loop body*. The body S is executed as long as the guard B evaluates to **true**. If the guard evaluates to **false** then the loop terminates.

1.3.2 Program States

During the executing of a program, the values stored in the variables associated with the program change. At any moment of execution time, the set of values assigned to the variables is called the *state* of the program at that given moment. More formally, the *state* of a program is a collection of pairs (x_i, v_i) , where v_i is a variable with its corresponding value v_i .

Example: Consider the following program fragment S , where x , y , and a are integer variables:

$$S : \quad x := x + a; \quad y := y - a;$$

Two example states of this program are $\{(x, 2), (y, 3), (a, 5)\}$ and $\{(x = -10), (y = 12), (a, 5)\}$. If we

execute the command S starting in any of these states, then after execution, the states have been transformed in the states $\{(x, 7), (y, -2), (a, 5)\}$ and $\{(x = -5), (y = 7), (a, 5)\}$ respectively. This command S will be used as a running example in the rest of this document.

1.3.3 Hoare Triples, and Specification Constants

Clearly, it is infeasible to track the states during the execution of a program for all possible starting states. However, *predicates* (expressions in first order logic) can be used to describe sets of states. For example, the infinite set of states $\{(x, 1)\}, \{(x, 2)\}, \{(x, 3), \dots\}$ is easily specified by the predicate $x > 0$. Using predicates, we can describe how execution of a program changes its state. For this we use a notation called *Hoare triples*. A Hoare triple is of the form:

$$\{P\} S \{Q\}$$

The operational meaning of this notation is:

If we execute command S in a state where precondition P holds, execution of S will terminate in a state where postcondition Q holds.

Since P is a predicate that describes the state before execution, it is called the *precondition* of S . Similarly, because Q describes the state after execution, it is called the *postcondition* of S .

Example: When we use again the program S that adds a to x and subtracts a from y and we know that initially $x + y = Z$ (for any value of Z), then it is clear that after executing $x + y = Z$ is still valid. As a result, the following is a valid Hoare triple for the program fragment S :

$$\begin{aligned} &\{P : x + y = Z\} \\ &x := x + a; \\ &y := y - a; \\ &\{Q : x + y = Z\} \end{aligned}$$

Note that in the above Hoare triple the value Z was introduced, though it is not a program variable. We call such a value a *specification constant*, which is used to implicitly quantify over all possible initial values for the expression $x + y$. Hence, the Hoare triple $\{x + y = Z\} S \{x + y = Z\}$ actually implicitly means

$$\forall(Z :: \{x + y = Z\} S \{x + y = Z\})$$

1.3.4 Annotations, and Weakest Preconditions

Now that it is clear what Hoare triples are, we need a formalism to check the validity of Hoare triples. For example, it is clear that the Hoare triple $\{x = Z\} x := x + 1; \{x = Z + 1\}$ is correct, while the Hoare triple $\{x = Z + 1\} x := x + 1; \{x = Z\}$ is not.

A systematic way of reasoning about Hoare triples uses the notion of *weakest preconditions*, which were introduced by Edsger W. Dijkstra (cf. [10]). For a given program statement S and some postcondition Q there is a set of program states such that if execution of S is initiated from any one of these states, then S ends up in a state in which Q holds. This set of initial states is called the *weakest precondition* of S , and is denoted by $wp(S, Q)$.

For some language constructs of the Aladin programming language, the function wp is easy to compute:

- $wp(\text{skip}, Q) \equiv Q$
- $wp(x := E, Q) \equiv [E/x]Q$
- $wp(S_0; S_1, Q) \equiv wp(S_0, wp(S_1, Q))$

Here, the notation $[E/x]Q$ denotes the predicate Q in which each occurrence of x has been replaced by the expression E (where E is an expression in the declared program variables and constants). For example, $wp(y := y - a, x + y = Z) \equiv x + (y - a) = Z$.

Using weakest preconditions, we can verify whether a Hoare triple is valid. The proof rule for the validity of the Hoare triple $\{P\} S \{Q\}$ is:

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

Example: When we use again the program S which adds a to x and subtracts a from y (and $P \equiv Q \equiv x + y = Z$), we can compute its weakest precondition as follows:

$$\begin{aligned}
 P' &: wp(S, Q) \\
 &\equiv wp(x := x + a, wp(y := y - a, x + y = Z)) \\
 &\equiv wp(x := x + a, x + (y - a) = Z) \\
 &\equiv (x + a) + (y - a) = Z
 \end{aligned}$$

For humans it is easy to see that $(x + a) + (y - a) = Z \equiv x + y = Z$, and thus trivially $P \Rightarrow P'$. Hence, the program fragment is correct.

Unfortunately, this is not easy at all for computers. Computing the weakest precondition with a computer program is (almost) trivial, since it is just syntactical manipulation of expressions. However, proving that an implication, in this case $P \Rightarrow P'$, is hard and is the main theme of this thesis.

We are rarely interested in proving the correctness of programs that consist of only or two assignments. Therefore, we introduce the concept of an *annotation* (also known as an *annotated program*). An annotation is a program fragment that has a precondition P , a postcondition Q , and *assertions* (predicates about the program state enclosed by curly braces) inserted in the program text. An annotation that has sufficiently fine-grained assertions inserted can be considered to be a proof of correctness of the program. In an annotated program, an assignment command can be enclosed between two assertions. The predicate before the assignment is the precondition of that command, while it is the postcondition of the command that precedes the assignment.

Example: Some possible valid annotations of the program fragment S are:

$$\begin{array}{lll}
 \{P : x + y = Z\} & \{P : x + y = Z\} & \{P : x + y = Z\} \\
 x := x + a; & x := x + a; & (* \text{calculus in preparation of } x := x + a *) \\
 y := y - a; & \{x + y = Z + a\} & \{(x + a - a) + y = Z\} \\
 \{Q : x + y = Z\} & y := y - a; & x := x + a; \\
 & \{Q : x + y = Z\} & \{(x - a) + y = Z\} \\
 & & (* \text{calculus in preparation of } y := y - a *) \\
 & & \{(x - a) + (y - a + a) = Z\} \\
 & & y := y - a; \\
 & & \{(x - a) + (y + a) = Z\} \\
 & & (* \text{calculus} *) \\
 & & \{Q : x + y = Z\}
 \end{array}$$

As can be seen from the left most case in the above example, multiple assertions may occur consecutively, possibly with some comment(s) in between. Comments are enclosed by the parentheses $(*$ and $*)$. For example, if we take the assertions $\{P\}\{Q\}$, possibly with some comment(s) in between them, then this is equivalent with the Hoare triple $\{P\} \text{ skip } \{Q\}$. Hence, in order to show that $\{P\} (*...*) \{Q\}$ is correct, it suffices to show that $P \Rightarrow Q$. Comments in between assertions should give a human readable argument why the implication holds, but is not part of the proof (nor is it checked by Aladin). Some examples of valid annotations are given above.

In the right most one, comments are included. First the assignment $x := x + a$ is prepared, by transforming the assertion $\{P : x + y = Z\}$ into the equivalent assertion $\{(x + a - a) + y = Z\}$, where $(*$ calculus in preparation of $x := x + a$ $*)$ argues why $x + y = Z \Rightarrow (x + a - a) + y = Z$ holds. The reason for this transformation is that $wp(x := x + a, (x - a) + y = Z)$ is $(x + a - a) + y = Z$. The same strategy is used to prepare for the assignment $y := y - a$. Finally, the assertion $(x - a) + (y + a) = Z$ clearly implies the postcondition $x + y = Z$ by applying some calculus.

1.4 Resolution and Unification

Producing a correct annotation requires smart inventive manipulation of predicates, and is typically a task that a computer program cannot perform. However, verifying the correctness of a given annotation by a computer program may be feasible. To verify that a given input annotation is correct, Aladin uses a proof technique called *resolution* combined with arithmetic, and a process that is called *unification*. Resolution and unification are explained in this section. Moreover, CNF (Conjunctive Normal Form) and semi-decidability are discussed, which are strongly related to reasoning by resolution.

1.4.1 Resolution in Propositional Logic

Resolution is a technique (or algorithm), based on *proof by contradiction*. Resolution corresponds exactly to the standard mathematical proof technique of *reductio ad absurdum*, which means *reduction to an absurd thing*. A resolution proof is based on producing a proof of the *unsatisfiability* of the logical negation of what we want to prove.

For example, proving a logical sentence α from a set KB of givens is done by checking the unsatisfiability of $KB \wedge \neg\alpha$. To do this, one assumes α to be false (by negating α) and shows that this leads to a contradiction with the set of givens KB . This contradiction is exactly what is meant by saying that the logical sentence $\alpha \wedge \neg\beta$ is unsatisfiable. Note the naming of the set of givens being KB . The reason for this naming is that KB is a set of logical sentences which is usually called the *Knowledge Base*. The set KB defines the context in which the sentence α is shown to be correct. Logicians denote this with the notation $KB \vdash \alpha$, which is pronounced as 'the KB entails α '. The knowledge base contains a set of rules which are assumed to be true. However, it is up to the supplier of the KB to make sure that the KB is consistent, otherwise $KB \wedge \neg\alpha$ is automatically unsatisfiable (and so is $KB \wedge \alpha$).

The resolution algorithm tries to prove that $KB \vdash \alpha$. To prove this, the algorithm tries to show that $KB \wedge \neg\alpha$ is unsatisfiable. It does this by repeated application of an inference rule, called the *resolution rule* which is applied to *clauses*.

A *clause* is a logical expression formed from a disjunction of a finite set of literals (atoms or their negations). A clause is true whenever at least one of the literals that form it is true, or false otherwise. An example of two clauses are $P \vee Q \vee R$ and $P \vee \neg Q \vee \neg S$. Since clauses use only one binary connective, being disjunction (\vee), we can write these clauses unambiguously as sets of literals: $C_0 = \{P, Q, R\}$ and $C_1 = \{P, \neg Q, \neg S\}$.

The *resolution rule* is an inference rule that produces a new clause that is implied by two *clauses* containing complementary literals. Two literals are said to be complementary if one is the negation of the other. The resulting clause contains all the literals that do not have complements. In formal logical notation, the resolution rule is described as:

$$\alpha \vee \beta, \neg\alpha \vee \gamma \vdash \beta \vee \gamma$$

The correctness of this rule is completely evident if we realize that $\neg\alpha \vee \gamma$ is equivalent to $\alpha \Rightarrow \gamma$, and $\alpha \vee \beta$ is equivalent to $\neg\beta \Rightarrow \alpha$. Hence, the resolution rule is basically a direct translation of the transitivity of implication:

$$\neg\beta \Rightarrow \alpha, \alpha \Rightarrow \gamma \vdash \neg\beta \Rightarrow \gamma$$

Note that $\neg\beta \Rightarrow \gamma$ is equivalent to $\neg\neg\beta \vee \gamma$, which reduces to $\beta \vee \gamma$ after removal of double negation.

If we apply the resolution rule to the clauses C_0 and C_1 that contain complementary literals (Q and $\neg Q$), then we find the clause $C_{0,1} = \{P, R, \neg S\}$. We call $C_{0,1}$ the *resolvent* of C_0 and C_1 . Note that syntactically we may conclude that $C_{0,1} = \{P, R, P, \neg S\}$, however sets do not contain duplicates which matches the logical rule that $P \vee P$ is equivalent to simply P . Also note that application of the resolution rule is useful only when the two clauses that are being resolved have only a single complimentary literal in common. If the clauses have at least two complimentary literals in common, then the resolution rule derives simply true, which is a useless conclusion. This is clear from the following example where resolution is applied to the complimentary literals P and $\neg P$:

$$P \vee Q \vee \alpha, \neg P \vee \neg Q \vee \beta \vdash Q \vee \alpha \vee \neg Q \vee \beta$$

Since $Q \vee \neg Q$ is always true, the resolvent is trivially true. Of course, the same will happen if we apply resolution to the complimentary literals Q and $\neg Q$.

```

function PL-RESOLUTION( $KB, \alpha$ ) returns true or false
  inputs:  $KB$ , the knowledge base, a sentence in propositional logic
            $\alpha$ , the query, a sentence in propositional logic

   $clauses \leftarrow$  the set of clauses in the CNF representation of  $KB \wedge \neg\alpha$ 
   $new \leftarrow \{ \}$ 
  loop do
    for each pair of clauses  $C_i, C_j$  in  $clauses$  do
       $resolvents \leftarrow$  PL-RESOLVE( $C_i, C_j$ )
      if  $resolvents$  contains the empty clause then return true
       $new \leftarrow new \cup resolvents$ 
    if  $new \subseteq clauses$  then return false
     $clauses \leftarrow clauses \cup new$ 

```

Figure 1.1: Resolution algorithm for Propositional Logic (PL)

The resolution algorithm for *propositional logic (PL)* is shown in figure 1.1. In reality, Aladin uses *First order logic (FOL)* instead, however we prefer to discuss the algorithm for propositional logic first and discuss an adapted version for FOL later in section 1.4.3.

To use the resolution rule, all sentences in KB and the query α have to be transformed into *Conjunctive Normal Form (CNF)* which will be explained in the next section. The result of this conversion is a set of clauses. On this set of clauses an iterative process is performed. In each step of this process, each pair of clauses that contains complementary literals is resolved to a new clause using the resolution rule. This process is repeated until one of the following conditions is met:

- In an iteration, no new clauses were inferred. In this case the algorithm could not infer anything new, and the algorithm stops. The conclusion is that it did not arrive at a contradiction and therefore the algorithm did not find a proof of $KB \vdash \alpha$.
- Two clauses resolved into an empty clause. Note that in the context of the algorithm the empty clause represents false. An empty clause is considered as a contradiction because it must have been produced by the resolution rule from two single literal clauses that are complementary (e.g. two clauses like $\{P\}$ and $\{\neg P\}$ resolve to the empty clause). In this case we have found a contradiction and we have proven $KB \vdash \alpha$.

Figure 1.2 gives an example of a proof using the resolution algorithm. We want to prove $[P \vee Q, P \Rightarrow Q, Q \Rightarrow P] \vdash P \wedge Q$. To prove this, we have to show that $[(P \vee Q) \wedge (P \Rightarrow Q) \wedge (Q \Rightarrow P)] \wedge \neg(P \wedge Q)$ is unsatisfiable by contradiction. We will do this by closely following the resolution algorithm. First, we have to transform $[(P \vee Q) \wedge (P \Rightarrow Q) \wedge (Q \Rightarrow P)] \wedge \neg(P \wedge Q)$ in CNF. The next section (1.4.2) explains how to do the conversion of an expression into CNF. For the time being, it is sufficient to know that $(P \vee Q) \wedge (P \vee \neg Q) \wedge (\neg P \vee Q) \wedge (\neg P \vee \neg Q)$ is a correct CNF representation of $KB \wedge \neg(P \wedge Q)$. The corresponding set of clauses is $\{P \vee Q, P \vee \neg Q, \neg P \vee Q, \neg P \vee \neg Q\}$.

Now it is time to pair clauses and infer their resolvents. To make things explicit, we list the steps of the first iteration of the resolution process:

1. The first two clauses resolve to $\{P\}$ because the literals Q and $\neg Q$ cancel due to the resolution rule.
2. The first and third clause resolve in the same way to $\{Q\}$ due to the cancellation of the complementary literals containing P .
3. The first and fourth clause do not resolve to anything useful, because they resolve to the clause that is trivially true (i.e. $Q \vee \neg Q$).
4. The second and the third clause also produce a useless trivial true clause.
5. The second and the fourth clause produce the clause $\{\neg Q\}$.
6. The third and the fourth clause produce again a useless trivial clause.

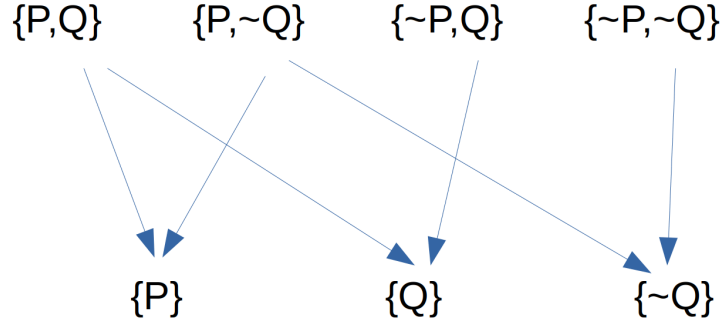


Figure 1.2: First iteration of the resolution proof of $[P \vee Q, P \Rightarrow Q, Q \Rightarrow P] \models P \wedge Q$

At the end of this first iteration, three new (singleton) clauses are found: $\{P\}$, $\{Q\}$, and $\{\neg Q\}$. The first iteration of this process is depicted in Fig. 1.2.

In a second iteration, we can resolve the two new clauses $\{Q\}$ and $\{\neg Q\}$ to obtain the empty clause, and hence the proof is completed. Note that the proof is not unique. Another valid proof, albeit using more steps, would be to pair the clause $\{Q\}$ with $\{\neg P \vee \neg Q\}$ to $\{\neg P\}$, and finally in a third iteration we can resolve $\{P\}$ and $\{\neg P\}$ to the empty clause $\{\}$. Note that the algorithm that is used in Aladin will always produce a proof with a minimal number of iterations. The reason is that the algorithm, as can be seen from the pseudocode in Fig. 1.1, is in fact a specific implementation of *breadth first search* (BFS) for a proof. It is well known that BFS will produce the solution with a minimal number of steps.

1.4.2 Conjunctive Normal Form (CNF)

The resolution rule can only be applied to clauses. Clauses are disjunctions of literals. Therefore, we need to rewrite logical sentences as a conjunction of clauses. This form is called *Conjunctive Normal Form*. To convert a sentence to CNF from propositional logic 4 steps are needed in the following order (see [17]):

1. Eliminate \Leftrightarrow : replace each occurrence of $\alpha \Leftrightarrow \beta$ by $(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)$
2. Eliminate \Rightarrow : replace each occurrence of $\alpha \Rightarrow \beta$ by $\neg\alpha \vee \beta$
3. Move negation (\neg) recursively inwards using the following equivalences:
 - $\neg(\neg\alpha) \equiv \alpha.$
 - $\neg(\alpha \wedge \beta) \equiv (\neg\alpha \vee \neg\beta).$
 - $\neg(\alpha \vee \beta) \equiv (\neg\alpha \wedge \neg\beta)$
4. Rewrite nested \wedge and \vee operators with the distributivity laws:
 - $(\alpha \wedge (\beta \vee \gamma)) \equiv ((\alpha \wedge \beta) \vee (\alpha \wedge \gamma))$ Distributivity of \wedge over \vee
 - $(\alpha \vee (\beta \wedge \gamma)) \equiv ((\alpha \vee \beta) \wedge (\alpha \vee \gamma))$ Distributivity of \vee over \wedge

After completion of these steps on some logical sentence, we obtained a logically equivalent sentence in CNF which is usually harder to read for a human person, but it can be used as input for the automated resolution algorithm.

1.4.3 Resolution in First Order Logic (FOL)

First-order logic (FOL) or predicate logic is an extension of propositional logic. The main difference lies in the introduction of the quantifiers \forall (for all) and \exists (there exists) and corresponding quantified variables (a.k.a. bound variables), and relational operators (like $=$, $<$, \leq , etc.). This makes it possible to write logical sentence like $\forall(x :: x < x + 1)$, where x is a quantified variable.

Resolution in First Order Logic is not that different from resolution in propositional logic. Just as in the case of propositional logic, a FOL sentence has to be converted into an equivalent CNF sentence. The first step of this conversion is to drop all \forall -quantifiers. This is simple, because it simply means that we drop the quantifier. However, this might introduce conflicts because a bound variable might suddenly interfere with a free (non-bound variable) with the same name. In Aladin this issue is solved by introducing a special

notation for bound variables. Those variables are prefixed by the symbol $\#$, such that the bound $\#x$ differs from the free variable x . As an example, the CNF conversion of the predicate $\forall(x :: x < x + 1)$ in Aladin yields the clause $\{\#x < \#x + 1\}$. Clearly, if needed, this clause can be renamed to $\{\#y < \#y + 1\}$ or any other name that does not conflict with names in other clauses. The second step in the conversion of FOL sentences in CNF is to remove the existential quantifier. This step is not implemented in Aladin, because Aladin does not use the existential quantifier. After removal of quantifiers, the process that is described in section 1.4.2 is followed to perform the remaining steps of the conversion.

In the remainder of this section, for readability reasons, we use Greek letters for bound variables, while we use Roman letters for free variables. We start by discussing an example with a knowledge base consisting of only one rule: $KB = \{(\alpha + \beta) + (\gamma - \beta) = \delta \Rightarrow \alpha + \gamma = \delta\}$. We want to prove $(x + a) + (y - a) = Z \Rightarrow x + y = Z$. The first step consists of the conversion of the knowledge base and the negation of the goal into CNF format. A CNF representation of the knowledge base is $\{\neg((\alpha + \beta) + (\gamma - \beta) = \delta) \vee \alpha + \gamma = \delta\}$ and a CNF representation of the negation of the goal is $\{(x + a) + (y - a) = Z \wedge \neg(x + y = Z)\}$. After that, we can use resolution to prove this example. Now we can use the resolution algorithm to proof the unsatisfiability of $\{(\neg((\alpha + \beta) + (\gamma - \beta) = \delta) \vee \alpha + \gamma = \delta) \wedge ((x + a) + (y - a) = Z) \wedge \neg(x + y = Z)\}$

Due to the introduction of relational operators, we have two different notations for the expression $x + y \neq Z$, being $x + y \neq Z$ itself and $\neg(x + y = Z)$. Even though these expressions are semantically equivalent, they are syntactically different. Since resolution is based on syntactically comparing clauses, it is awkward to deal with these dual notations. For this reason, we have made the following decision.

The only relational operators that are used in the resolution process are \neq , $<$, and \leq . Moreover, the negation operator (\neg) applied to a comparison is replaced by the complementary relational operator (e.g. $\neg(a < b)$ is replaced by $b \leq a$).

This decision significantly reduces the amount of cases that we have to consider in the resolution algorithm. For example, consider the literal $p > q$, then without this convention there are four possible complementary literals: $\neg(p > q)$, $\neg(q < p)$, $q \geq p$ and $p \leq q$. Clearly, this leads to a combinatorial explosion of cases, which is eliminated by adopting this convention.

If we apply this convention to the above example, then we end up with the following set of CNF clauses (knowledge base together with negation of the goal):

$$\{(\alpha + \beta) + (\gamma - \beta) \neq \delta, \alpha + \gamma = \delta\}, \{(x + a) + (y - a) = Z\}, \{x + y \neq Z\}$$

Now we start a resolution proving process. Clearly, the intention is that the literals $(\alpha + \beta) + (\gamma - \beta) \neq \delta$ and $(x + a) + (y - a) = Z$ are detected to be complementary. The problem however is that the variables have different names. To solve this, we need a way to replace bound variables (the Greek letters) by expressions. The process that detects which variables should be replaced by which expressions (if possible) is called *unification* and is explained in section 1.4.4. For now, it suffices to know that if we substitute x for α , a for β , y for γ and Z for δ , then we have two matching complementary literals, yielding the resolvent $\{\alpha + \gamma = \delta\}$ on which we have to apply the same substitution, i.e. the resolvent is the clause $\{x + y = Z\}$. In turn, this clause is resolved (without need for a substitution) with the complementary clause $\{x + y \neq Z\}$ resulting in the empty clause, and so the proof is completed.

An additional problem which may arise when converting First Order Logic sentences into equivalent CNF clauses is that the same variable name occurs more than once in different contexts. Consider for example the clauses $\{\alpha \cdot (\beta + \gamma) = \alpha \cdot \beta + \alpha \cdot \gamma\}$ and $\{\alpha + 0 = \alpha\}$. These clauses are the result of elimination of the quantifier \forall , and now have an overlapping name α , while clearly these α 's are not the same variable. To avoid confusion we have to rename one of the variables. The process that does this renaming is called *standardize-apart*. For example we can rename α in the second expression to δ without changing its meaning and we obtain $\{\delta + 0 = \delta\}$.

1.4.4 Unification

When humans produce formal proofs from a set of axioms (i.e. a knowledge base), they routinely match patterns, (implicitly) rename variables (to avoid name collisions), and perform substitution of expressions for bound variables. For example, consider a minimal knowledge base containing only the (familiar) rules $\alpha \cdot (\beta + \gamma) = \alpha \cdot \beta + \alpha \cdot \gamma$ and $\alpha + \beta = \beta + \gamma$. In both rules, we implicitly quantified over all possible values

of α , β , and γ . Clearly, these rules should suffice to prove that $x \cdot (y + z) = x \cdot z + x \cdot y$. However, in reality this observation consists of several implicit steps. The first consists of simply substituting $\alpha = x$, $\beta = y$ and $\gamma = z$ in the first rule to obtain $x \cdot (y + z) = x \cdot y + x \cdot z$. The reason is that, by pattern matching, we realized that we now have an expression on the left hand side of the goal that is equal to the left hand side of the expression that we just obtained. Hence, we can replace the left hand side of the goal by the right hand side of the equation that we just derived. More explicit, we used (probably without even realizing) the rule $E_0 = F_0 \wedge E_0 = F_1 \Rightarrow F_0 = F_1$. As a result, we obtained $x \cdot y + x \cdot z = x \cdot z + x \cdot y$. Next, we substitute $\alpha = x \cdot y$ and $\beta = y \cdot z$ in the second rule of the knowledge base to discover that the latter equation is true, and hence we completed the proof.

For a computer program, producing this proof is a lot harder. It requires pattern matching. For example, we had to match the left hand side of the goal (i.e. $x \cdot (y + z)$) with the left hand side of the first rule in the knowledge base (i.e. $\alpha \cdot (\beta + \gamma)$). For these expressions to match, we need the substitution $\theta = \{\alpha/x, \beta/y, \gamma/z\}$, where the notation v/e means that the quantified variable v must be replaced by the expression e . The set of substitution θ is called a *unifier* of the expressions $x \cdot (y + z)$ and $\alpha \cdot (\beta + \gamma)$.

Let $Subst(\theta, e)$ be a function that takes two arguments. Its first argument θ is a set of substitutions, and the second argument e is an expression. The function returns an expression that is obtained by applying all substitution from θ to the expression e . For example, $Subst(\{\alpha/x, \beta/y\}, \alpha + \beta + \gamma) = x + y + \gamma$. Using the function $Subst$ we can define what a *unifier* is.

A substitution θ is called a unifier of the expressions e_0 and e_1 if $Subst(\theta, e_0) = Subst(\theta, e_1)$.

Note that the empty set is a unifier for any expression with itself. Also note that a unifier is not unique. For example, the expressions $\alpha + \beta + \alpha$ and $x + \gamma + \delta$ have several unifiers. One obvious unifier is $\theta_0 = \{\alpha/x, \beta/x, \gamma/x, \delta/x\}$, but this one is very specific. A more general one is $\theta_1 = \{\alpha/x, \beta/\gamma, \delta/x\}$. Clearly we wish to use a unifier which is as general as possible, since it imposes fewer restrictions on the values of the variables. So, we want to use the *most general unifier (mgu)*, which is defined as follows:

A unifier θ_0 is called the most general unifier of the expressions e_0 and e_1 if for any other unifier θ_1 of e_0 and e_1 there exists a substitution θ_2 such that $Subst(\theta_2, Subst(\theta_0, e_0)) = Subst(\theta_1, e_0)$ and $Subst(\theta_2, Subst(\theta_0, e_1)) = Subst(\theta_1, e_1)$.

Indeed, the unifier $\theta_1 = \{\alpha/x, \beta/\gamma, \delta/x\}$ is an mgu of the expressions $\alpha + \beta + \alpha$ and $x + \gamma + \delta$, and θ_0 is less general because it can be composed of first applying θ_1 , followed by applying the substitution $\{\gamma/x\}$. Note that two expressions may have several most general unifiers, because most general unifiers are unique up to renaming of variables. The mgu $\theta_1 = \{\alpha/x, \beta/\gamma, \delta/x\}$ has a related mgu being $\theta'_1 = \{\alpha/x, \gamma/\beta, \delta/x\}$. Note that, if we have the choice between several mgu's, then it is irrelevant which one we choose.

To compute most general unifiers, we made a variant of the unification algorithm that was published by Martelli and Montanari in 1982 (see [16]). The variant is an extension of the original algorithm, because we incorporated arithmetic (numeric) expressions, while the original algorithm only addresses sentences in pure FOL. The original algorithm is used in the Prolog programming language. The pseudocode of this extended unification algorithm for unifying literals is given in Fig. 1.3. The main routine is the function `literalMGU` which takes two literals x and y , and returns a most general unifier or *failure* if no unifier exists. Note that a literal is an expressions of the type $lhs \preceq rhs$, where lhs and rhs are arithmetic expressions (involving numbers, constants, and the standard arithmetic operators). The comparison operator \preceq can be either $<$, \leq , $=$, or \neq (as discussed before). Note that in the pseudocode, the symbols \oplus and \otimes are used to denote any of the standard binary arithmetic operators ($+$, $-$, \times , **div**, **mod**).

The unification algorithm is simple. It recursively explores the two expressions simultaneously building up a unifier along the way (starting from an empty one). If at some point the (sub-)structures do not match, it returns *failure*. The algorithm contains one expensive step, which is the case in which it tries to unify a variable v with a (possibly complex) expression x (in the routine `unifyVar`). In that case we have to traverse the entire expression x to check whether the variable v occurs in x . If that is the case, no consistent unifier can be constructed and *failure* is returned.

function recursiveMGU(x, y, θ) **returns** a substitution to make x and y identical
input: x and y are arithmetic expressions, θ is the MGU so far
output: a most general unifier, or *fail* if such a unifier does not exist
if $\theta = \text{failure}$ **then return failure;**
if $x = y$ **then return** θ ;
if x is a variable **then return** unifyVar(x, y, θ);
if y is a variable **then return** unifyVar(y, x, θ);
if x is of the form $x_0 \oplus x_1$ and y is of the form $y_0 \otimes y_1$ and $\oplus = \otimes$
then return recursiveMGU($x_1, y_1, \text{recursiveMGU}(x_0, y_0)$);
return fail;

function unifyVar(v, x, θ) **returns** a substitution
input: v is a variable, x is an arithmetic expression, θ is the MGU so far
output: a most general unifier, or *fail* if such a unifier does not exist
if $v/e \in \theta$ **then return** recursiveMGU(e, x, θ);
if $x/e \in \theta$ **then return** recursiveMGU(v, e, θ);
if v occurs in x **then return failure;**
return $\theta \cup \{v/x\}$;

function arithMGU(x, y) **returns** a substitution to make arithmetic expressions x and y identical
input: x and y are arithmetic expressions
output: a most general unifier, or *fail* if such a unifier does not exist
return recursiveMGU(x, y, \emptyset);

function literalMGU(x, y) **returns** a substitution to make literals x and y identical
input: x and y are literals
output: a most general unifier, or *fail* if such a unifier does not exist
if $x = y$ **then return** \emptyset ;
if x is of the form $x_0 < x_1$ and y is of the form $y_0 < y_1$
then return recursiveMGU($x_1, y_1, \text{arithMGU}(x_0, y_0)$);
if x is of the form $x_0 \leq x_1$ and y is of the form $y_0 \leq y_1$
then return recursiveMGU($x_1, y_1, \text{arithMGU}(x_0, y_0)$);
if x is of the form $x_0 = x_1$ and y is of the form $y_0 = y_1$
then return recursiveMGU($x_1, y_1, \text{arithMGU}(x_0, y_0)$);
if x is of the form $x_0 \neq x_1$ and y is of the form $y_0 \neq y_1$
then return recursiveMGU($x_1, y_1, \text{arithMGU}(x_0, y_0)$);
return fail;

Figure 1.3: Unification algorithm for arithmetic expressions

1.4.5 Semi-Decidability

It is well-known that resolution is a *complete* process in the context of propositional logic. Here *complete* means that it is guaranteed to find a proof if a proof exists (i.e. the goal is entailed by the knowledge base).

Unfortunately, in the context of FOL this is no longer true. It is well-known that in this context the resolution technique is *semi-decidable*, which means that it will eventually (even though this may take a very long time) come up with a proof for entailed goals, but it may not terminate on goals which are not entailed by the knowledge base. The problem with sentences that are not entailed is that the resolution proof might be generating new clauses forever, and there is no way to detect that these new clauses will not eventually lead to a proof. In this sense, it is very similar to the famous Halting problem. Moreover, even sentences that are entailed by the knowledge base may not be provable in practice, because it simply requires too much computation time (or too much memory, or both). For example, if we want to generate a resolution proof for a goal α which is actually entailed by the knowledge base, but it takes like 20 iterations to finalize this proof (note again, that resolution is performing basically breadth first search) where each iteration on average triples the number of clause, then we need to generate in the order of $3^{20} \approx 3.5 \times 10^9$ clauses. Note that for many proofs, the number of clauses is not tripled but the number of clauses grows much faster than that. In fact, the worst case is that the number of clauses is squared in each iteration. Clearly, this means that even some entailed sentences can practically not be derived using the resolution technique.

As a consequence, if Aladin finds a proof for the correctness of annotation, then it will report that the annotation is correct and the user can rely on this verdict.

Aladin may also find that an annotation is incorrect by generating a counter example, in which case the counter example will be reported. Also, in this case the user can rely on this verdict (although he will be less happy).

But, since we cap the number of iterations that the resolution process may perform, Aladin may not be able to find a proof (even if it exists). In such a case, it will give "UNDECIDED" as output. So, when Aladin gives "UNDECIDED" as output it does not necessarily mean that the annotation is incorrect (but it might be). Moreover, it could also mean that the knowledge base does not have enough rules to complete the proof.

1.5 Example proof

Now that all necessary ingredients have been discussed, we conclude this chapter with an example of a complete verification of a simple annotation. For this example we use the running example:

$$\begin{aligned} &\{P : x + y = Z\} \\ &x := x + a; \\ &y := y - a; \\ &\{Q : x + y = Z\} \end{aligned}$$

First, we compute the weakest precondition P' of the two assignments, and after that we need to prove that $P \Rightarrow P'$. We obtain P' as follows:

$$\begin{aligned} P' &: wp(x := x + a, wp(y := y - a, x + y = Z)) \\ &\equiv wp(x := x + a, x + (y - a) = Z) \\ &\equiv (x + a) + (y - a) = Z \end{aligned}$$

Our goal α is to prove that $x + y = Z \Rightarrow (x + a) + (y - a) = Z$. Since we want to perform a resolution proof, we negate this goal, and transform it into CNF using the steps from section 1.4.2:

$$\begin{aligned} &\{\neg(x + y = Z \Rightarrow (x + a) + (y - a) = Z)\} \\ &\equiv (* \text{ Step 2 } *) \\ &\quad \{\neg(\neg(x + y = Z) \vee (x + a) + (y - a) = Z)\} \\ &\equiv (* \text{ Step 3 } *) \\ &\quad \{x + y = Z \wedge (x + a) + (y - a) \neq Z\} \end{aligned}$$

So, we find the set of clauses $\alpha = \{\{x + y = Z\}, \{(x + a) + (y - a) \neq Z\}\}$.

Now we assume we have a knowledge base KB which contains the following basic rules of arithmetic:

- $\alpha + \beta = \gamma \Leftrightarrow \alpha = \gamma - \beta$
- $\alpha = (\beta - \gamma) + \delta \Rightarrow \alpha = \beta - (\gamma - \delta)$
- $\alpha = \beta \Rightarrow \alpha + \gamma = \beta + \gamma$

Of course, the implications in the 2nd and 3rd rule can be replaced by equivalences, but to keep the example proof as short as possible, we decided to use only equivalences when they are needed in the proof. The identifiers ' α '..' δ ' are quantified variables, and are therefore unifiable. Note that the identifiers x , y , and a are program variables, and hence are not unifiable variables. Nor is Z , which is a specification constant. The set of clauses that we obtain after converting the knowledge base into CNF is:

$$KB = \{\{\alpha + \beta \neq \gamma, \alpha = \gamma - \beta\}, \{\alpha \neq \gamma - \beta, \alpha + \beta = \gamma\}, \{\alpha \neq (\beta - \gamma) + \delta, \alpha = \beta - (\gamma - \delta)\}, \{\alpha \neq \beta, \alpha + \gamma = \beta + \gamma\}\}$$

Now that the knowledge base and the goal are in clausal format, we can start the resolution process. In figure 1.4 we see the complete resolution proof tree for $KB \models \alpha$. The clauses that form the negation of the goal are marked in grey.

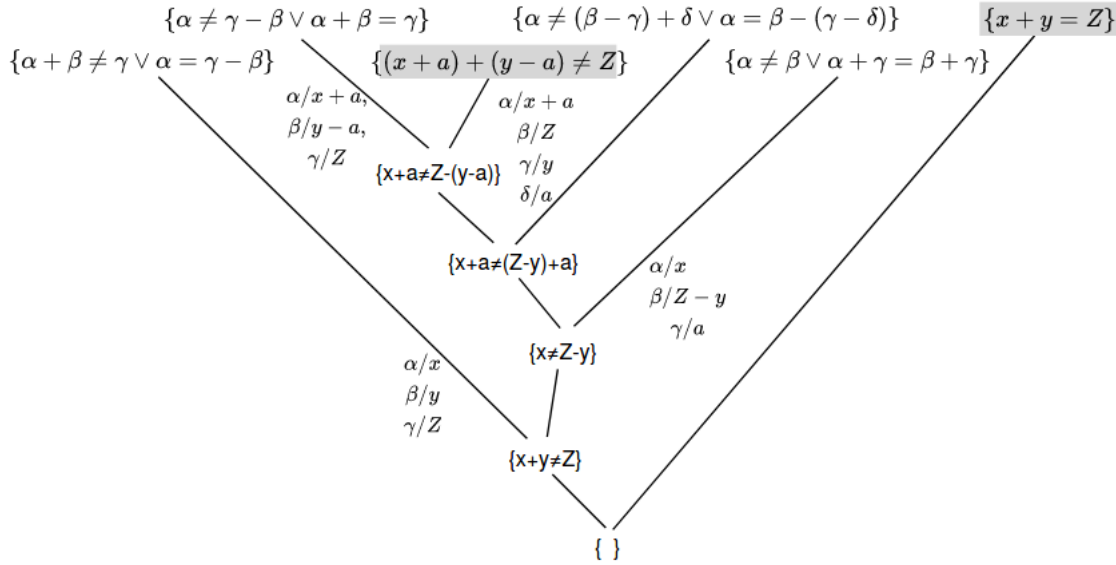


Figure 1.4: Proof of the running example

Note that the tree shows only resolvents that lead to completion of the proof. The resolution process generates more resolvents than the ones shown in the figure. For example $\{\alpha + \beta \neq \gamma, \alpha = \gamma - \beta\}$ and $\{x + y = Z\}$ would yield the resolvent $\{x = Z - y\}$, but it does not constitute to the proof. For reasons of readability we left out superfluous resolvents, but the reader should realize that the resolution process does generate them.

In the first step the clauses $\{\alpha \neq \gamma - \delta, \alpha + \beta = \gamma\}$ and $\{(x + a) + (y - a) \neq Z\}$ resolve, using the mgu $\{\alpha/(x + a), \beta/(y - a), \gamma/Z\}$, to the clause $\{x + a \neq Z - (y - a)\}$. On its turn, this clause resolves with $\{\alpha \neq (\beta - \gamma) + \delta, \alpha = \beta - (\gamma - \delta)\}$ to $\{x + a \neq (Z - y) + a\}$ using the mgu $\{\alpha/(x + a), \beta/Z, \gamma/y, \delta/a\}$. In the third step, this result resolves with $\{\alpha \neq \beta, \alpha + \gamma = \beta + \gamma\}$ to $\{x \neq Z - y\}$ using the mgu $\{\alpha/x, \beta/Z - y, \gamma/a\}$. This result on its turn resolves with the clause $\{\alpha + \beta \neq \gamma, \alpha = \gamma - \beta\}$ to $\{x + y \neq Z\}$ using the mgu $\{\alpha/x, \beta/y, \gamma/Z\}$. The last step is the observation that we now have the complementary clauses $\{x + y \neq Z\}$ and $\{x + y = Z\}$, so no mgu is needed to obtain the empty clause, and hence the proof is completed.

Chapter 2

Parsing and Storing Expressions


Clearly, we need data structures to represent expressions. Moreover, we need functions that convert a textual representation of an expression into the corresponding data structures. In this chapter, we will discuss how we store and parse numerical and Boolean expressions.

We have decided to implement Aladin in the functional programming language Haskell (cf. [15]). The reason for this decision is that it is easy in Haskell to syntactically manipulate expressions (or data structures in general) without the need of (deep-)copying data structures and doing memory management. Moreover, the pattern matching features of Haskell, together with its possibility to use wildcards, really helps in writing short and elegant implementations of many recursive algorithms.

2.1 Numerical Expressions

We start with the representation of numerical expressions. In Aladin, numeric expressions consist of integer constants, specification constants, program variables, and the standard arithmetic operators (+, −, *, **div**, and **mod**). Note that Aladin (currently) only supports integer expressions. A floating point data type, and a Boolean data type may be added in the future.

Numerical expressions are defined by the standard grammar which is shown in figure 2.1. The grammar can be parsed with a recursive descent parser (see section 2.3).



```

$$\begin{array}{ll} E & \rightarrow T E' \\ E' & \rightarrow + T E' \\ & \quad | - T E' \\ & \quad | \text{<empty string>} \\ T & \rightarrow F T' \\ T' & \rightarrow * F T' \\ & \quad | \text{div } F T' \\ & \quad | \text{mod } F T' \\ & \quad | \text{<empty string>} \\ F & \rightarrow ( E ) \\ & \quad | - F \\ & \quad | \text{<integer constant>} \\ & \quad | \text{<specification constant>} \\ & \quad | \text{<variables>} \\ & \quad | \text{<unifiable variables>} \end{array}$$

```

Figure 2.1: Grammar for numeric expressions. The start symbol is E .

This grammar is used for parsing numeric expressions in annotations as well as in the knowledge base. This explains why `<unifiable variables>` (i.e. bound/quantified variables) are incorporated in the grammar, even though they are not allowed to appear in an annotation (which is checked by the parser). The other way around, specification constants may appear in the annotation of a program, but not in the program code itself, nor may they appear in the knowledge base. These restrictions are checked by a semantic analysis phase that follows syntactic checking.

In this grammar we can see that addition (+) and subtraction (−) have a lower priority than multiplication (*), division (**div**), and computing a remainder (**mod**). In turn, we see that these operations have a lower priority than expressions enclosed by parentheses, or factors that are prefixed by a unary minus sign.

We use the following Haskell data structure `NumExpr` to store numerical expressions:

```

1 type Name = String
2 type UnifiableVar = Int
3
4 data NumOperator = Add | Sub | Mult | Div | Mod
5
6 data NumExpr = IntConst Integer
7               | SpecConst Name
8               | Var Name
9               | UniVar UnifiableVar
10              | UnaryMinus NumExpr
11              | Binary NumOperator NumExpr NumExpr

```

The constructor `IntConst` is used for integer constants. The constructor `SpecConst` is used for specification constants, which are strings of which the first letter must be a capital letter. The constructor `Var` is used for variables, which are strings of which the first letter must be a lowercase letter. The constructor `UniVar` is used for unifiable variables, followed by an integer. The reason for this to be an integer is that the actual name of a unifiable variable is not relevant. The names of unifiable variables in the knowledge base are prefixed by the character #, and are replaced by the parsing process by an integer. For example, a rule in the knowledge base like `#a*#b=#a+#a*(#b-1)` is replaced, after parsing, by `#0*#1=#0+#0*(#1-1)`, where `#a` is replaced by `#0` and `#b` is replaced by `#1`. Choosing integers, instead of names, is very helpful later in the resolution process, since it is easy this way to standardize apart unifiable variables by simply adding offsets to the integer names. The constructor `UnaryMinus` is used to store expressions that are prefixed by a unary minus sign (like `-5`, but also complex expressions like `-(x+y)`). We chose to make an extra data type called `NumOperator` to represent the binary arithmetical operations. The reason is that this yields significant less cases in a lot of functions in which we use pattern matching. For example, if we want to recursively compute a list of all variables in a numeric expression, then the recursive process is exactly the same for expressions of the type $e_0 \oplus e_1$, where \oplus denotes any of the binary operators. All these case can now be matched by a single line of code, thanks to pattern matching and the introduction of the data type `NumOperator`.

2.2 Boolean Expressions

Next, we consider the representation of Boolean expressions. In Aladin, Boolean expressions consist of atoms (literals) and the Boolean operators `==` (equivalence), `->` (implies), `<-` (follows), `and`, `or` and `not` (negation).

Boolean expressions are defined by the grammar which is shown in figure 2.2, which also can be parsed using a recursive descent parser (see section 2.3). In this grammar we see that the connectives `==` (equivalence), `<-` (follows) and `->` (implies) all have a lower priority than the `or` connective. In turn, this connective has a lower priority than the `and` connective. Finally, Boolean expressions enclosed by parentheses and the negation of a Boolean expression have the highest priority. Note that we deliberately chose the parentheses for Boolean expressions to be square brackets (i.e. `[` and `]`). This solves a recursive descent parsing conflict between the parentheses used for numeric expressions and those from Boolean

BE	\rightarrow	$BO\ BE'$
BE'	\rightarrow	$==\ BO\ BE'$
	$ $	$<-\ BO\ BE'$
	$ $	$->\ BO\ BE'$
	$ $	$<\text{empty string}>$
BO	\rightarrow	$BT\ BO'$
BO'	\rightarrow	$\text{or}\ BF\ BT'$
	$ $	$<\text{empty string}>$
BT	\rightarrow	$BF\ BT'$
BT'	\rightarrow	$\text{and}\ BF\ BT'$
	$ $	$<\text{empty string}>$
BF	\rightarrow	$[BE]$
	$ $	$\text{not}\ BF$
	$ $	$ATOM$
$ATOM$	\rightarrow	true
	$ $	false
	$ $	$E < E$
	$ $	$E <= E$
	$ $	$E = E$
	$ $	$E >= E$
	$ $	$E > E$
	$ $	$E \neq E$

Figure 2.2: Grammar for Boolean expressions. The start symbol is BE .

expressions. For example, if we try to parse an expression like $(x > 0 \text{ or } 2 * (x + y) < z)$ and $z < x + y$ using a recursive descent parser, then the parser cannot decide (at the moment that it detects the first open parenthesis), whether this would match (E) from the grammar for numeric expressions, or (BF) from the grammar for Boolean expressions. There are two ways out of this parsing conflict. Either we use another parsing strategy (a so-called LR(0) bottom up parser is able to solve this conflict), or we use different style parentheses for Boolean expressions. We chose to do the latter, since this project is not a project with its main focus on compiler construction techniques. So, the given example must be written as $[x > 0 \text{ or } 2 * (x + y) < z]$ and $z < x + y$. Moreover, we actually like this choice, because as a side-effect of this decision we have now typed parentheses, which actually makes expressions with many mixed parentheses easier to read.

We use the following Haskell data structure `BoolExpr` to store Boolean expressions:

```

1 data BinaryBoolOperator = And | Or | Implies | Follows | Equivalence
2
3 data BoolExpr = Atom BoolAtom
4               | BinOp BinaryBoolOperator BoolExpr BoolExpr
5               | Not BoolExpr

```

A Boolean expression may be an atom, a negation of a Boolean expression, or an expression using one of the other five connectives together with two Boolean expressions (a left-hand side and a right-hand side). Similarly to the `NumExpr` data structure, we used the extra data type `BinaryBoolOperator` and the constructor `BinOp` to limit the amount of cases to consider in pattern matching.

We use the following Haskell data structure to store Boolean atoms:

```

1 data BoolAtom = T | F | Compare CompOperator NumExpr NumExpr
2
3 data CompOperator = LessThan | LessEqual | Equal | NotEqual

```

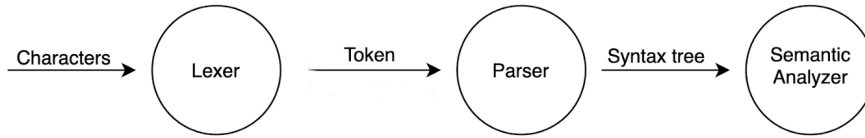


Figure 2.3: Processing pipeline for parsing.

Here again, we made an extra data type `CompOperator` and the constructor `Compare` to reduce the number of cases for pattern matching. As we can see in this code, a Boolean atom can be `T` (true), `F` (false) or a comparison operator together with two numerical expressions (the left-hand side and the right-hand side of the comparison). Note that we have omitted the comparison operators `>` and `≥`. As explained before, we decided to remove these operators for simplicity. For example, the atoms $p > q$ and $r \geq s$ can easily be rewritten as $q < p$ and $s \leq r$ respectively. This conversion is performed by Aladin directly in the parser for Boolean atoms. The main advantage of this decision is that we need to consider fewer cases in the resolution algorithm.

2.3 Parsing Expressions

Now that we have a way to store expressions, we discuss how to parse the expressions. The process that converts the input, which is basically a (large) string of characters, into `BoolExprs` and `NumExprs`, which are representations of syntax trees, is called *parsing*.

The parsing process is very similar to a standard front end of a compiler, and consists of a pipeline with three stages (see Fig. 2.3). The first stage is called the *lexer*, the second the *parser*, and the third the *semantic analyzer*.

The lexer is the first stage of the pipeline. Its task is to convert the input, which is a sequence of characters, into a sequence of tokens. The reason to use a lexer is that at the level of the parser we do not have to process the input at the low level granularity of characters, but at a higher level of tokens (keywords, arithmetic operators, etc.). Moreover, the lexer is able to skip superfluous white space (spaces, tabs, and newlines) without the parser even knowing.

We implemented the lexer as a standard Haskell function which has the prototype

```
lexer :: String -> [(LexToken, Int)]
```

Its input is simply a `String` of characters, while its output is a list of pairs of the type `(LexToken, Int)`. Each pair consists of a recognized token, and an integer value representing the line number of the input in which the token was found. This line number is irrelevant in the rest of the processing pipeline, but it is used for error reporting.

The type `LexToken` is an enumeration type representing all tokens:

```

1 data LexToken = LessThanTok | LessEqualTok | GreaterThanTok
2               | GreaterEqualTok | EqualTok | NotEqualTok | LparTok
3               | RparTok | LsqBrackTok | RsqBrackTok | LcurlBrackTok
4               | RcurlBrackTok | SemicolonTok | CommaTok | TrueTok
5               | FalseTok | AndTok | OrTok | NotTok | ImpliesTok | FollowsTok
6               | EquivalenceTok | IfTok | ThenTok | ElseTok | EndTok | WhileTok
7               | DoTok | AddTok | SubTok | MultTok | DivTok | ModTok
8               | AssignTok | SkipTok | VarDeclTok | ConstTok | LetTok
9               | CommentTok String | VarTok String | UniVarTok String
10              | SpecConstTok String | IntTok Integer

```

The data type `LexToken` is used for tokenizing annotated programs, and not only expressions. Note that the constructors in the lines 9-10 have an additional argument. The constructor `CommentTok` is used to store comment lines (i.e. text between `(*` and `*)` in a program) and thus has an additional `String` to store this text. Note that a lexer for a compiler would simply drop comment lines, but we do not want to lose these lines because we need it for pretty printing the output. The same holds for the constructor `VarTok`, which is used for storing the name of a variable (i.e. a `String`). The constructor `UniVarTok` is used for representing unifiable variable (i.e. identifiers that start with the character `#`). Note that, as discussed before, these names are replaced by integers, but that is done at the level of the parser. The lexer simply passes the name of the unifiable variable as a `String`. Similarly, the constructor `SpecConstTok` needs a `String` to store the name of a specification constant. If the lexer encounters a series of digits, then these digits are converted to an integer value and returned via the constructor `IntTok`. We will not discuss the exact implementation of the lexer, since its implementation details are not relevant in the rest of this document.

The next stage of the pipeline is the parser. The grammar for the input of Aladin has been constructed such that it can be parsed using a *recursive descent parser*. This type of parsers can readily be written by hand, by an (almost) mechanic translation from grammar rules to program code. Grammars of this type are called $LL(k)$ grammars (see [6]), and have the property that the input can be parsed from left to right, and that at any moment the parser can decide which grammar rule to apply based on (at most) the last k tokens. The grammar of Aladin is in the class $LL(1)$, so the parser can always decide which grammar rule to apply based on a single token. Note, as said before, we have decided to use different style parentheses for Boolean and numeric expressions. Without this decision the grammar of Aladin would not have been $LL(k)$ (for any k), requiring a much more complicated parsing technique.

We now discuss the parsing of Boolean expressions. The main Haskell function for parsing these expressions is called `parseBoolExpr`.

```

1 parseBoolExpr :: String -> (BoolExpr, [(LexToken, Int)])
2 parseBoolExpr input = parseBexpr (lexer input)
3
4 parseBexpr :: [(LexToken, Int)] -> (BoolExpr, [(LexToken, Int)])
5 parseBexpr tokens = (acc, rest)
6   where (acc, _, rest) = parseBE [] tokens

```

The function `parseBoolExpr` takes as its input a string of characters, which gets tokenized using the `lexer`. The output of the lexer is the input for the function `parseBexpr`, which is the actual starting function of the recursive descent parsing process. Note that the return type of the functions `parseBoolExpr` and `parseBexpr` is a pair of the type `(BoolExpr, [(LexToken, Int)])`. On a successful parse, the first element of this pair is the accepted expression, while the second element is the remaining tokenized input (if it exists). For example, the call `parseBoolExpr "2*x=y=y"` would return a pair of which the first element denotes the expression `2*x=y`, and the second element would be a list containing the tokenization of `"=y"`. From the code snippet we can see that the function `parseBoolExpr` is actually a wrapper function around the function `parseBexpr`. The reason for this wrapping is that we use the function `parseBexpr` also in the Aladin parser for complete programs (not only expressions), and in that case the input has already been tokenized.

The function `parseBexpr` calls the function `parseBE`, which is the function that is the Haskell translation of the starting rule from the grammar for Boolean expressions. Note that the function `parseBE` accepts two arguments. The first argument is a list of names of unification variables that is built up and passed around during the parsing of the expression. Initially this list is empty, hence `parseBexpr` passes the empty list to `parseBE`. The list is needed in the translation of the names of unification variables into unique integers. The second argument of the function is <https://www.overleaf.com/project/5ece62befd98920001654f5c> the list of input tokens that is produced by the lexer.

The consequence of this design is that many functions that are part of the parser for `BoolExprs` are typed as `[Name] -> [(LexToken, Int)] -> (BoolExpr, [Name], [(LexToken, Int)])`. The first two arguments are as explained. The return type is a triple, of which the first element is the accepted parsed expression, the second the (possibly extended) list of unification variables that was built up (so far), and the

third is the remaining tokenized input.

The function `parseBE` also has this type, and implements the grammar rule $BE \rightarrow BO BE'$.

```

1 parseBE :: [Name] -> [(LexToken, Int)] ->
2   (BoolExpr, [Name], [(LexToken, Int)])
3 parseBE uvars tokens = parseBE' acc uv rest
4   where (acc, uv, rest) = parseBO uvars tokens

```

As the grammar rule dictates, the parser first tries to accept an expression that can be produced by BO . Next, it tries to parse BE' . The function that accepts BO is called `parseBO`, and the function that accepts BE' is called `parseBE'`. The processing order is clear from the code. First `parseBO` is called in the `where` clause, and its output is input for the function `parseBE'`. In fact, this style of coding actually implements a form of function composition.

Recall the grammar rule for BE' :

$$\begin{array}{lcl}
 BE' & \rightarrow & == BO BE' \\
 & | & <- BO BE' \\
 & | & -> BO BE' \\
 & | & <\text{empty string}>
 \end{array}$$

The Haskell function `parseBE'` implements the parsing of this grammar rule.

```

1 parseBE' :: BoolExpr -> [Name] -> [(LexToken, Int)] ->
2   (BoolExpr, [Name], [(LexToken, Int)])
3 parseBE' accepted uvars ((EquivalenceTok, _):tokens) =
4   parseBE' (BinOp Equivalence accepted acc) uv rest
5   where (acc, uv, rest) = parseBO uvars tokens
6 parseBE' accepted uvars ((ImpliesTok, _):tokens) =
7   parseBE' (BinOp Implies accepted acc) uv rest
8   where (acc, uv, rest) = parseBO uvars tokens
9 parseBE' accepted uvars ((FollowsTok, _):tokens) =
10  parseBE' (BinOp Follows accepted acc) uv rest
11  where (acc, uv, rest) = parseBO uvars tokens
12 parseBE' accepted uvars tokens = (accepted, uvars, tokens)

```

Note that `parseBE'` has an additional (first) argument with the type `BoolExpr`. This argument contains the Boolean expression that was already built up before `parseBE'` was called (i.e. in `parseBO`), and this function will return it immediately in the case it accepts the empty string (line 12). Otherwise, it matches any of the connectives `==` (equivalence), `->` (implies), or `<-` (follows), and uses this extra argument in the left hand side of the returned binary Boolean expression. In these cases (lines 3-11), the structure is the same as in the function `parseBE`, i.e. first `parseBO` is called and its output is the input for another recursive call to `parseBE'`. This structure is typical for recursive descent parsers. In imperative languages, such parsers typically consist of huge nested if-then-else constructs. In Haskell we can make use of its (top-down) pattern matching facility, to prevent this nested coding style.

The grammar rules for BO , BO' , BT , BT' , and BF have the same structure. As noted before, the reason that the grammar is written in a layered fashion is that the priority of the connectives is now explicitly expressed in the structure of the grammar rules.

$$\begin{array}{ll}
BO & \rightarrow BT \ BO' \\
BO' & \rightarrow \text{or } BF \ BT' \\
& \quad | \text{ <empty string>} \\
BT & \rightarrow BF \ BT' \\
BT' & \rightarrow \text{and } BF \ BT' \\
& \quad | \text{ <empty string>} \\
BF & \rightarrow [\ BE \] \\
& \quad | \text{ not } BF \\
& \quad | \text{ ATOM}
\end{array}$$

The parsing functions for these rules have the same structure as the combination `parseBE` and `parseBE'` :

```

1  parseBO :: [Name] -> [(LexToken,Int)] ->
2      (BoolExpr, [Name], [(LexToken,Int)])
3  parseBO uvars tokens = parseBO' acc uv rest
4      where (acc,uv,rest) = parseBT uvars tokens
5
6  parseBO' :: BoolExpr -> [Name] -> [(LexToken,Int)] ->
7      (BoolExpr, [Name], [(LexToken,Int)])
8  parseBO' accepted uvars ((OrTok,_) : tokens) = parseBO'
9      (BinOp Or accepted acc) uv rest
10     where (acc,uv,rest) = parseBT uvars tokens
11 parseBO' accepted uvars tokens = (accepted, uvars, tokens)
12
13 parseBT :: [Name] -> [(LexToken,Int)] ->
14     (BoolExpr, [Name], [(LexToken,Int)])
15 parseBT uvars tokens = parseBT' acc uv rest
16     where (acc,uv,rest) = parseBF uvars tokens
17
18 parseBT' :: BoolExpr -> [Name] -> [(LexToken,Int)] ->
19     (BoolExpr, [Name], [(LexToken,Int)])
20 parseBT' accepted uvars ((AndTok,_) : tokens) = parseBT'
21     (BinOp And accepted acc) uv rest
22     where (acc,uv,rest) = parseBF uvars tokens
23 parseBT' accepted uvars tokens = (accepted, uvars, tokens)
24
25 parseBF :: [Name] -> [(LexToken,Int)] ->
26     (BoolExpr, [Name], [(LexToken,Int)])
27 parseBF uvars ((NotTok,_) : tokens) = ((Not acc), uv, rest)
28     where (acc,uv,rest) = parseBF uvars tokens
29 parseBF uvars ((LsqBrackTok,_) : tokens) = (expr, uv, tokens1)
30     where (expr, uv, ((RsqBrackTok,_) : tokens1)) = parseBE uvars tokens
31 parseBF uvars (tok : tokens) = ((Atom acc), uv, rest)
32     where (acc,uv,rest) = parseBA uvars (tok : tokens)

```

The last grammar rule of which we will discuss the parser is the grammar rule for *ATOM*s.

$$\text{ATOM} \rightarrow T \mid F \mid E < E \mid E \leq E \mid E = E \mid E \geq E \mid E > E \mid E / = E$$

The function `parseBA` implements the parser for this rule. The function makes use of the function `parseE` which is responsible for parsing numerical expressions. The first two cases, *T* (true) and *F* (false), are easy. The remaining cases are comparisons of two numeric expression. In these cases, first the function `parseE` is called (in line 7), which parses the left hand side (*lhs*) of the comparison. Next, a helper function

`parseBA'` is called. This helper function parses the remaining list of tokens, which should start with a comparison operator followed by a numeric expression. Note that, if the remaining list of tokens does not start with such an operator, then an error is reported (lines 18-19). Moreover, an error is also reported in the case of an unexpected end of input (line 7). If a successful parse yields a right hand side, then the function `makeAtom` (together with the comparison operator) produces in the end a valid `BoolAtom` expression. Note that the comparisons `lhs > rhs` and `lhs >= rhs` are converted into `rhs < lhs` and `rhs <= lhs` respectively (in lines 14-15).

```

1  parseBA :: [Name] -> [(LexToken, Int)] ->
2      (BoolAtom, [Name], [(LexToken, Int)])
3  parseBA uvars ((TrueTok, _) : tokens) = (T, uvars, tokens)
4  parseBA uvars ((FalseTok, _) : tokens) = (F, uvars, tokens)
5  parseBA uvars tokens = parseBA' rest
6      where
7      (lhs, uvrs, rest) = parseE uvars tokens
8      parseBA' [] = error "Error: unexpected end of input."
9      parseBA' (tok:tokens) = (makeAtom tok lhs rhs, uv, rest)
10         where (rhs, uv, rest) = parseE uvrs tokens
11  makeAtom :: (LexToken, Int) -> NumExpr -> NumExpr -> BoolAtom
12  makeAtom (LessThanTok, _) lhs rhs = Compare LessThan lhs rhs
13  makeAtom (LessEqualTok, _) lhs rhs = Compare LessEqual lhs rhs
14  makeAtom (GreaterThanTok, _) lhs rhs = Compare LessThan rhs lhs
15  makeAtom (GreaterEqualTok, _) lhs rhs = Compare LessEqual rhs lhs
16  makeAtom (EqualTok, _) lhs rhs = Compare Equal lhs rhs
17  makeAtom (NotEqualTok, _) lhs rhs = Compare NotEqual lhs rhs
18  makeAtom (_, n) _ _ = error ("Syntax error in line " ++ show(n)
19                               ++ ": expected relational operator")

```

Parsing numerical expressions is done with the function `parseE` and a set of other parsing functions. The type of the function `parseE` is similar to the type of `parseBE`.

```

1  parseE :: [Name] -> [(LexToken, Int)] ->
2      (NumExpr, [Name], [(LexToken, Int)])

```

The structure of the parser functions for numerical expressions is very similar to the parsing functions for Boolean expressions. In numerical expressions there is a hierarchy based on the precedence rules for the arithmetic operators, similar to the hierarchy of the Boolean connectives that was discussed in this section. We will not discuss the parsing of numerical expressions any further, because the discussion would be a repetition of the discussion of parsing Boolean expressions. Moreover, parsing is an essential part of Aladin, but it is not the main focus of this thesis.

Chapter 3

Finding Counterexamples

As described in chapter 1, for incorrect annotations Aladin *might* output "ERROR" together with a counterexample. If counterexamples exist, it is not guaranteed that Aladin actually succeeds in finding them. Aladin is only doing a best effort in trying to find them. The fact that Aladin did not find a counterexample is therefore not a proof of correctness, which is in line with the following famous quote of E.W. Dijkstra:

"Program testing can be used to show the presence of bugs, but never to show their absence!"

Finding those counter examples is the topic of this chapter. The method that Aladin uses to find them is based on a combination of finding solutions of a *constraint satisfaction problem* (CSP) and simulated executions of the program in which all assertions are checked on the fly. Both elements are discussed in this chapter. The constraint satisfaction problem consists of finding initial values of the specification constants and program variables that satisfy the precondition of the program. The solutions of these CSPs are used as the initial states for simulation runs.

3.1 Parsing and Storing Programs

Aladin accepts as its input an annotated program. Therefore we need to parse annotated programs and store them in a data type. The LL(1) grammar of Aladin is given in Figure 3.1. The non-terminals E and BE refer to the grammars for numeric and Boolean expressions that was discussed in chapter 2.

<i>Program</i>	→	<i>ConstDecl</i> ; <i>VarDecl</i> ; <i>Statements</i>
<i>ConstDecl</i>	→	const <i>IdentList</i> <empty string>
<i>VarDecl</i>	→	var <i>IdentList</i> <empty string>
<i>IdentList</i>	→	< identifier > <i>Idl'</i>
<i>Idl'</i>	→	, < identifier > <i>Idl'</i> <empty string>
<i>Statements</i>	→	<i>Command</i> ; <i>Statements</i> <empty string>
<i>Command</i>	→	skip <i>Commentline</i> <i>Assertion</i> <i>Assignment</i> <i>ConditionalCommand</i> <i>WhileCommand</i>
<i>Commentline</i>	→	(* <anything in text> *)
<i>Assertion</i>	→	{ <i>Assertion'</i> }
<i>Assertion'</i>	→	<i>BE</i> let <specification constant> = <i>E</i>
<i>Assignment</i>	→	<variable> := <i>E</i>
<i>ConditionalCommand</i>	→	if <i>BE</i> then <i>Commands</i> else <i>Commands</i> end
<i>WhileCommand</i>	→	while <i>BE</i> do <i>Commands</i> end

Figure 3.1: Grammar of Aladin

From the grammar it is clear that a program consists of an optional list of declared constants, followed by an optional list of declared variables, followed by a series of statements. This observation leads naturally to the following Haskell type to represent programs.

```
data Program = Program [Name] [Name] [(Statement, Int)]
```

A `Program` consists of a list of specification constants (the first `[Name]` list), a list of declared variables (the second `[Name]` list) and a list of tuples `(Statement, Int)`. The first element of such a tuple is of the type `Statement` which represents a program statement, while the second element is an integer denoting the corresponding line number in the source code. This line number is used for proper error reporting.

An Aladin statement can be a **skip** statement (which does not do anything), an assignment to a variable, a conditional statement, or a loop. Moreover, comment lines and assertions are also considered statements that have no effect on the program state.

We also introduced a **let** statement which is a special type of assertion. This 'statement' can be used to introduce a new local specification constant. Its primary use is in termination proofs of repetitions that use a bounded integer expression vf , called the *variant function*, that needs to decrease in each iteration of the repetition. If S is the body of such a loop, then this can be expressed as the proof rule $\{vf = V\} S \{vf < V\}$. However, the value V is not a normal specification constant, but actually a local constant that, before the execution of S , has the value of the expression vf . This can be expressed in Aladin using the **let** construct: $\{\text{let } V = vf\}$, where vf is an expression in the program variables and (normal) specification constants.

We use the following Haskell data structure `Statement` to store statements:

```
1 data Statement = Skip
2                 | Comment String
3                 | Assertion BoolExpr
4                 | Let Name NumExpr
5                 | Assignment Name NumExpr
6                 | Conditional BoolExpr [(Statement, Int)]
7                 | [(Statement, Int)]
8                 | WhileLoop BoolExpr [(Statement, Int)]
```

Parsing a program is done in the same way as parsing Boolean and numeric expressions using a recursive descent parser. We decided that each program consists of an assertion, the precondition, followed by a number of blocks. A block is either a series of assignments (possible mixed with comments and skip commands), a conditional statement, or a loop construct. Each block must be followed by an assertion. This enforces that each block has a 'local' post condition, and a 'local' precondition that is the post condition of the preceding block. The first block of the program has no preceding block, and its precondition is the precondition of the entire program. Similarly, the last block its 'local' postcondition is actually the postcondition of the entire program. Note that this block structure of an annotated program is not enforced by the grammar in figure 3.1, however after a program has been parsed Aladin checks whether the program satisfies these structural rules.

3.2 Reducing Expressions

In the simulation runs in which Aladin tries to find counterexamples, we clearly need a way to evaluate expressions given values for the variables involved to evaluate the right hand side of assignments. Moreover, we also need a way to reduce (simplify) expressions in the CSP solving process. In this section we explain how expressions are reduced maximally.

Note that we explicitly use the term *reduction* of expressions (instead of evaluation). The reason is that we want to reduce (or simplify) expressions, even if no value is available for some of the identifiers involved. For example, if we know that x equals 2, while we do not know the value of y , then the numeric expression $x + 2 * (5 + x) * y$ can still be reduced to the expression $2 + 14 * y$. A reduction becomes an evaluation if

values for all involved identifiers are known. We start this section with a discussion on the reduction of numeric expressions.

3.2.1 Simplifying and Reducing Numeric Expressions

We call an identifier together with its value a *valuation*, which is implemented as a tuple of two numerical expressions. The first element represents the identifier, and the second a numerical expression.

```
type Valuation = (NumExpr, NumExpr)
```

Note that the 'value' of an identifier is a numeric expression. For example, $x = 42$ is represented by the valuation (Var "x", IntConst 42), while (Var "y", Var "z") is a valuation for $y = z$.

We implemented a function `reduceNumExpr` which has the prototype

```
1 reduceNumExpr :: NumExpr -> [Valuation] -> NumExpr
```

It accepts a numerical expression and a list of valuations. If that list is empty, `reduceNumExpr` will return a numeric expression which is obtained by simplifying sub-expressions in which no variables are involved (e.g. $2*3*x+5*3$ will reduce to $6*x+15$). If the list of valuations is non-empty, then `reduceNumExpr` will substitute these valuations during the reduction of the numerical expression. The implementation of `reduceNumExpr` is straightforward, and we will discuss it only partially. We consider only a few cases of the call `reduceNumExpr e`.

In the case that e is a simple integer constant, it is returned unmodified. In the case that e is an identifier (a variable, specification constant, or a unifiable variable), the list of valuations is searched to find a corresponding valuating expression for the identifier. If such an expression is found, then the reduction of this expression is returned. If it is not found, then the identifier itself is returned unmodified. The following code snippet shows these cases.

```
1 reduceNumExpr :: NumExpr -> [Valuation] -> NumExpr
2 reduceNumExpr (IntConst c) _      = IntConst c
3 reduceNumExpr (Var x) vs          = reduceNumIdentExpr (Var x) vs
4 reduceNumExpr (UniVar x) vs       = reduceNumIdentExpr (UniVar x) vs
5 reduceNumExpr (SpecConst x) vs    = reduceNumIdentExpr (SpecConst x) vs
6
7 reduceNumIdentExpr :: NumExpr -> [Valuation] -> NumExpr
8 reduceNumIdentExpr idt vs = reduceIdent [z | (y,z) <- vs, y==idt]
9   where
10     reduceIdent [] = idt
11     reduceIdent xs = reduceNumExpr (head xs) vs
```

The remaining cases, in which e is not a numeric atom, are simply implementations of the arithmetic operators. Of these, we only show the implementation of the multiplication operation. The remaining operations have similar code. For all binary operators, first the operands are recursively reduced, after which the corresponding operator is implemented. Note that expressions like $x*0$ and $0*x$ are reduced to 0, and $1*x$ and $x*1$ are reduced to x . Several other optimisations are used for other operators as well. For example, the expression $e0-e0$ is simply reduced to zero, even if $e0$ is a complicated expression.

```
1 reduceNumExpr (Binary Mult p q) vs =
2   reduceMult (reduceNumExpr p vs) (reduceNumExpr q vs)
3   where
4     reduceMult (IntConst 0) rhs      = IntConst 0
5     reduceMult (IntConst 1) rhs      = reduceNumExpr rhs vs
6     reduceMult lhs (IntConst 0)      = IntConst 0
7     reduceMult lhs (IntConst 1)      = reduceNumExpr lhs vs
8     reduceMult (IntConst x) (IntConst y) = IntConst (x*y)
9     reduceMult (IntConst x) (Binary Mult (IntConst y) e0) =
```

```

10      Binary Mult (IntConst (x*y)) e0
11      reduceMult (Binary Mult (IntConst x) e0) e1 =
12          reduceNumExpr (Binary Mult
13              (IntConst x) (reduceNumExpr (Binary Mult e0 e1) vs)) vs
14      reduceMult lhs (IntConst y) =
15          reduceNumExpr (Binary Mult (IntConst y) lhs) vs
16      reduceMult lhs rhs = Binary Mult lhs rhs

```

3.2.2 Simplifying and Reducing Boolean Expressions

Now that we have a function to reduce numeric expressions, we consider the reduction of Boolean expression. We start with the function `reduceBoolExpr`, which (like numeric expressions in section 3.2.1) reduces a Boolean expression as much as possible given a list of valuations. However, the function does not infer valuations. As an example, if we reduce the Boolean expression $x=36-2$ and $y=2*4+x$ using an empty valuations list, then the returned reduced expression should be $x=34$ and $y=8 + x$, and not $x=40$ and $y=42$ because the latter requires the inference that $x=34$ can be substituted in the second conjunct. The reason for not allowing such inferences is simply that it is very hard to do this efficiently in a recursive one-pass algorithm. For example, if the algorithm does perform inferences and processes first the left conjunct, followed by the right conjunct, then reduction of the equivalent expression $y=2*4+x$ and $x=36-2$ would yield a different answer. However, if we do a reduction of the expression $x=36-2$ and $y=2*4+x$ with the valuation list `[(Var "x", IntConst 34)]` (meaning that the value of x is 34) then we obtain the expression $y=42$. The reason is that $x=34$ is now not an inference, but a valuation which can be substituted everywhere where x occurs. Hence $x=36-2$ reduces to $34=34$ (i.e. true), and the result is the remainder of the reduction of the second second conjunct. Similarly, if we try to reduce $x=36-2$ and $y=2*4+x$ with the valuation list `[(Var "x", IntConst 0)]` then the returned reduced expression is simply `false`, since the reduction of the first conjunct already yields `false`.

In the following code snippet we consider a fragment of the function `reduceBoolExpr`.

```

1  reduceBoolExpr :: BoolExpr -> [Valuation] -> BoolExpr
2  reduceBoolExpr (BinOp And e0 e1) vals =
3      reduceAndExpr (reduceBoolExpr e0 vals) (reduceBoolExpr e1 vals)
4  where
5      reduceAndExpr (Atom T) rhs = rhs
6      reduceAndExpr (Atom F) rhs = Atom F
7      reduceAndExpr lhs (Atom T) = lhs
8      reduceAndExpr lhs (Atom F) = Atom F
9      reduceAndExpr lhs rhs      = BinOp And lhs rhs

```

In the above code snippet the reduction of a Boolean expression which is the conjunction of two Boolean sub-expressions ($e0$ and $e1$) (given a list of valuations) is computed. The function recursively reduces the expressions $e0$ and $e1$, and combines the result using the helper function `reduceAndExpr`. From the code it is clear that the function performs *short-circuit reduction* thanks to Haskell's lazy evaluation strategy. For example, if the reduction of $e0$ yields `false` (line 6), then the returned reduced expression is automatically `false` (i.e. `Atom F`) without reducing the second conjunct. Similarly, if the first conjunct reduces to `true`, then the resulting reduced expression is the reduction of the second conjunct (line 5). The lines 7 and 8 are based on the same reasoning for the second argument, and are only reached when the first conjunct does not reduce to either true or false. Line 9 is reached if neither conjunct reduces to true or false, and the returned reduced expression is the conjunction of the reduced sub-expressions.

For the reduction of disjunctions, `reduceBoolExpr` has similar code that is tailored to the properties of disjunctions.

```

1  reduceBoolExpr (BinOp Or e0 e1) vals =
2      reduceOrExpr (reduceBoolExpr e0 vals) (reduceBoolExpr e1 vals)
3  where
4      reduceOrExpr (Atom T) rhs = Atom T

```

```

5   reduceOrExpr (Atom F) rhs = rhs
6   reduceOrExpr lhs (Atom T) = Atom T
7   reduceOrExpr lhs (Atom F) = lhs
8   reduceOrExpr lhs rhs      = BinOp Or lhs rhs

```

For the other three binary connectives (implies, follows and equivalence), the function `reduceBoolExpr` has similar code structure (which is not discussed further). An interesting remaining case is the reduction of a negation of the type `Not e`, where `e` is a `BoolExpr`. If `e = Not (Not e0)`, then the negations cancel and we recursively reduce `e0`. In the case that `e` is a conjunction of two expressions `e0` and `e1`, then we apply De Morgan's law, and reduce the disjunction of the negation of `e0` and the negation of `e1`. De Morgan's rule is also applied if `e` is a disjunction of two sub-expressions. In the case that `e` is an implication of the type `e0 implies e1`, then we recursively reduce the expression `e0` and `not e1`, which is a direct translation of $\neg(p \rightarrow q) \equiv \neg(\neg p \vee q) \equiv p \wedge \neg q$. The same rule is used if `e` is an implication in the opposite direction. In the case that `e` is an equivalence of two sub-expressions `e0` and `e1`, then we simply apply the rule $\neg(p \equiv q) \equiv (\neg p \equiv q)$. The remaining case is that `e` is an `Atom`, in which case a helper function `negateAtom` is used to implement the reduction of the negation of `e` (e.g. the comparison $a < b$ is replaced by $b \leq a$).

```

1   reduceBoolExpr (Not e) vals = reduceNotExpr e
2   where
3       reduceNotExpr (Not e0) = reduceBoolExpr e0 vals
4       reduceNotExpr (BinOp And e0 e1) =
5           reduceBoolExpr (BinOp Or (Not e0) (Not e1)) vals
6       reduceNotExpr (BinOp Or e0 e1) =
7           reduceBoolExpr (BinOp And (Not e0) (Not e1)) vals
8       reduceNotExpr (BinOp Implies e0 e1) =
9           reduceBoolExpr (BinOp And e0 (Not e1)) vals
10      reduceNotExpr (BinOp Follows e0 e1) =
11          reduceBoolExpr (BinOp And e1 (Not e0)) vals
12      reduceNotExpr (BinOp Equivalence e0 e1) =
13          reduceBoolExpr (BinOp Equivalence (Not e0) e1) vals
14      reduceNotExpr (Atom e0) =
15          reduceBoolExpr (Atom (negateAtom e0)) vals
16
17  negateAtom :: BoolAtom -> BoolAtom
18  negateAtom T           = F
19  negateAtom F           = T
20  negateAtom (Compare LessThan p q) = Compare LessEqual q p
21  negateAtom (Compare LessEqual p q) = Compare LessThan q p
22  negateAtom (Compare Equal p q)     = Compare NotEqual p q
23  negateAtom (Compare NotEqual p q)  = Compare Equal p q

```

There is only one case left for `reduceBoolExpr e`, and that is the case in which `e` is an `Atom`.

```

1   reduceBoolExpr (Atom e) vals = Atom (reduceBoolAtom e vals)
2
3   reduceBoolAtom :: BoolAtom -> [Valuation] -> BoolAtom
4   reduceBoolAtom T _           = T
5   reduceBoolAtom F _           = F
6   reduceBoolAtom (Compare op p q) vals =
7       reduceAtom op (reduceNumExpr p vals) (reduceNumExpr q vals)
8   where
9       reduceAtom LessThan (IntConst x) (IntConst y) =
10           if x < y then T else F
11       reduceAtom LessEqual (IntConst x) (IntConst y) =
12           if x <= y then T else F

```

```

13   reduceAtom Equal (IntConst x) (IntConst y)      =
14       if x == y then T else F
15   reduceAtom NotEqual (IntConst x) (IntConst y)   =
16       if x /= y then T else F
17   reduceAtom LessEqual e0 e1 = if e0==e1 then T else
18       (Compare LessEqual e0 e1)
19   reduceAtom Equal e0 e1 = if e0==e1 then T else
20       (Compare Equal e0 e1)
21   reduceAtom LessThan e0 e1 = if e0==e1 then F else
22       (Compare LessThan e0 e1)
23   reduceAtom op lhs rhs = Compare op lhs rhs

```

An `Atom` is either a trivial case (being true or false), in which case it simply reduces to itself, or a comparison of two numeric expressions. In the latter case, the two numeric expressions are reduced using the function `reduceNumExpr`. If both these reduction yield an integer constant, then the corresponding comparison is performed and the atom reduces to either `T` (true) or `F` (false). If two expressions `e0` and `e1` are not fully reduced, but are the same and have one of the compare operators `<`, `≤` or `=`, then we can perform an optimization to reduce a whole expression to `T` or `F`. If the expressions are not the same, we can simply return the comparison. Also, for all other cases a (possibly reduced) comparison is returned.

3.3 Solving a CSP

To find a counterexample for an annotation, simulation runs of the program are performed. To run such a simulation, we have to find a set of values for the specification constants in the precondition together with initial values for program variables such that the precondition holds. Given such a set of values, a simulation run is started, in which each assertion in the annotation is checked for validity. As soon as one of these assertions is false, the offending state (and line number in the program) is reported. Moreover, the initial state and the values of the specification constants are reported as a counter example of the annotation. In the case that a simulation run terminates without detecting false assertions, Aladin simply tries another simulation run starting from another initial set of values.

Finding initial values for the specification constants and variables that satisfy the precondition is actually a special case of solving a *constraint satisfaction problem* (CSP). Many methods (using clever heuristics) exist to solve constraint satisfaction problems, but the problem remains NP-complete which is obvious since many NP-complete problems can be phrased as a CSP. In 2015 a bachelor thesis by J. Bakker (see [7]) was published about this subject including the implementation of several heuristics to speed up the solving process. Unfortunately, the solvers from that thesis are not applicable for this project, since the solvers were implemented in C and cannot process the Haskell data structures that are used by Aladin.

An extra problem that we are facing is that in standard CSPs a finite domain of values is given for each variable of the CSP. In Aladin this is not true, since integers can take on an infinite number of values. We have decided to approach this problem in a pragmatic fashion by simply choosing the domain finite, being the range $[-100..100]$ for some variables. For example, if we have two variables x and y that need to satisfy $2 * x = y$ then the system iterates for x over this limited range, and computes for each x a corresponding value for y . Hence, the value of y may not be from the domain $[-100..100]$.

Since the main objective of this thesis is to check formal annotations, we did not spend much time on implementing an efficient CSP solver using heuristics. Instead, we use the naive method of generating values for some variables and trying to solve for remaining variables. Clearly, this approach is not very efficient and can be improved implementing the techniques from [7] in Haskell.

In order to try values for the identifiers (variables and/or specification constants) in the CSP (which is represented by a `BoolExpr`), we need a function that returns the list of variables in the CSP. The function `identifiersInBoolExpr` does exactly that. It recursively traverses a Boolean expression, with the help of the functions `identifiersInBoolAtom` and `identifierInNumExpr`, constructing a list of variables on the fly.

```

1 identifiersInBoolExpr :: BoolExpr -> [NumExpr]
2 identifiersInBoolExpr expr = nub (idents expr)
3   where
4     idents (Atom e)           = identifiersInBoolAtom e
5     idents (Not e)           = idents e
6     idents (BinOp op lhs rhs) = idents lhs ++ idents rhs
7
8 identifiersInBoolAtom :: BoolAtom -> [NumExpr]
9 identifiersInBoolAtom expr = nub (idents expr)
10    where
11      idents (Compare op lhs rhs) =
12        identifiersInNumExpr lhs ++ identifiersInNumExpr rhs
13      idents _                    = []
14
15 identifiersInNumExpr :: NumExpr -> [NumExpr]
16 identifiersInNumExpr expr = nub (idents expr)
17    where
18      idents (IntConst _)      = []
19      idents (UnaryMinus e)    = idents e
20      idents (Binary op lhs rhs) = idents lhs ++ idents rhs
21      idents ident             = [ident]

```

Note that `identifiersInBoolExpr` does not return a list of `Names`, but a list of `NumExprs`. The reason is that the returned list is a list that can contain variables and specification constants. As noted before in the discussion of the data type `NumExpr` we represent variables by numeric expressions of the type `Var Name`, and specification constants by numeric expressions of the type `SpecConst Name`. Also note that the recursive helper function `idents` yields a list that may contain duplicates. The standard Haskell function `nub` (in line 2) is used to remove duplicates from the list.

Now that we have a list of the relevant identifiers, we can try to assign values to them in search of a set of values that satisfy the CSP. A list of valuations in which each identifier of the CSP is given a value such that the CSP is satisfied, is a solution of the CSP. Using the function `reduceBoolExpr` (see section 3.2.2) it is easy to test whether a list of valuations is a solution, because it would simply mean that the reduction of the CSP (given the valuations) reduces to `T` (true).

The function `solveCSP` is the top-level function of the process that tries to solve a CSP. Its first argument is the CSP represented as a Boolean expression. It returns a list of lists of valuations. Each of these lists is a solution to the CSP. If the CSP has no solution, the empty list (of lists) is returned.

```

1 solveCSP :: BoolExpr -> [[Valuation]]
2 solveCSP csp = solve idents reducedCSP []
3   where
4     reducedCSP = reduceBoolExpr csp []
5     idents = identifiersInBoolExpr csp
6     solve _ (Atom T) vals = [vals]
7     solve _ (Atom F) vals = []
8     solve (var:vars) csp valuations = let val=searchValuation csp in
9       if val == Nothing then
10         concat[solve vars (reduceBoolExpr csp [v]) (v:vals) | v <- dom var]
11       else
12         solve vars (reduceBoolExpr csp (sure val:vals)) (sure val:vals)
13     dom var = [(var,IntConst n) | n <- 0:shake 100 31]
14     shake :: Int -> Integer -> [Integer]
15     shake 0 _ = []
16     shake n m = -m : m : shake (n-1) (99*m `mod` 101)

```

Before starting the recursive solving process, which is done by the helper function `solve`, the CSP is reduced with an empty valuation (i.e. constant sub-expressions are simplified). The function `solve` takes three arguments. The first is the list of identifiers which have not been assigned a value yet (i.e. they do not have a valuation). The second is a CSP which is reduced as much as possible using a list of valuations that is being built up during the solving process. This list of valuations is the third argument of the function. It returns a list of solutions.

The base cases of the recursive function `solve` are in the lines 6-7. If the CSP has been reduced to `T` (true) then a solution is found. This solution is returned as a singleton list (line 6). If the CSP has been reduced to `F` (false) then the valuation that has been built up is not a solution. Therefore, the empty list is returned (line 7). Note that there is no special case for the empty list of identifiers that have no valuation yet, because in that case the CSP will reduce to either `T` or `F` anyway.

This leaves one more case to consider, which is the case that the list of identifiers is not empty (i.e. `var:vars`), and `csp` is not fully reduced. In this case, we could in principal choose any variable that has not a valuation yet, and try to substitute all possible values for it (in a backtracking style). However, consider as an example the following case in which `csp` represents the expression $x=2*y$ and $y+7=28$, and `var` is the variable `x`. It would be inefficient to try all possible values for `x`, while it is clear that `y` equals 21, and hence by substitution `x` must be 42. For cases like these, we implemented the following optimisation. First (see line 8), a function `searchValuation` is called. This function returns values of the type `Maybe Valuation`. If `csp` contains a conjunct which is an equation containing a single identifier, then it tries to solve the equation for that identifier. If it succeeds in solving the equation, then it returns a valuation for that identifier. So, in the given example, it would return `Just (Var "y", IntConst 21)`. This means that the recursive `solve` process can simply use this valuation to reduce `csp`, add it to the list of valuations found this far, and continue to solve recursively (line 12). If `searchValuation` cannot find a solution to a single identifier conjunct, then it returns `Nothing` and the solve process has to resort in trying all possible valuations for `var` (see line 10). In this case, for `var` all values from the domain $[-100..100]$ are tried in a pseudo-random order. This pseudo random order is implemented by the function `shake`, that implements a simple *linear congruential pseudo random generator* (see e.g. [14]) that generates numbers from the domain $\pm[0..100]$ in pseudo random order using the recurrence $x_0 = 31$ and $x_{i+1} = (99 \cdot x_i) \bmod 101$. It is well known, because 101 is a prime number, that this recurrence will visit all numbers from the specified range. Note that there may be several solutions to the CSP, which explains why line 10 is a list comprehension, which is of the type `[[Valuation]]`. The function `concat` converts this result to the proper return type `[[Valuation]]`.

3.4 Running Program Simulations

Using the CSP solver, we can try to find initial program states that satisfy the precondition of a program. If we succeed in finding those, then we start simulating program runs starting from any of these initial states, verifying assertions on the fly. The top level function of this process is the function `runTests`.

```
runTests :: Int -> Program -> String
```

The first argument of `runTests` is an integer, which specifies the maximum number of attempts that Aladin should try to find a counterexample. The number is a maximum, since Aladin might not be able to find that many states that fit the precondition. In fact, due to the simplicity of the CSP solver, it might not perform any test at all, even though satisfying initial states exist. The function `runTests` returns a string with the possible outcomes "Passed X tests" (where X is the number of executed simulation runs) or "Assertion failed". In the latter case, Aladin also reports the initial state of the program, and the state at the moment that the assertion failed (and the corresponding line number). Note that, if Aladin does not find a counterexample within the maximal number of attempts, it may in the end still output "UNDECIDED" if it does not find a formal proof for the annotation.

```
1 runTests :: Int -> Program -> String
2 runTests trials program = result `echo` result
3   where
4     result = verdict (runProgram program initstates)
```



```

5   initstates = take trials (solveCSP (precondition program))
6   verdict Nothing = "Passed "++show(length initstates)+" tests."
7   verdict (Just (init, (state, assertion, line))) =
8       "Assertion failed on line " ++ (show line) ++ ": " ++
9       show assertion ++ "\n" ++
10      "Initial state: " ++ (valuationToString init) ++ "\n" ++
11      "Current state: " ++ (valuationToString state) ++ "\n"
12
13 precondition :: Program -> BoolExpr
14 precondition (Program constants variables code) = pre code
15   where pre ((Comment text, nr):rest) = pre rest
16         pre ((Assertion expr, nr):rest) = expr
17         pre _ = error ("No precondition found :(")

```

The function `runTests` first determines (using the helper function `precondition`) the precondition of the program which should be the first non-comment line of the code segment. Next, it computes a list `initstates` of states that match the precondition (see line 5). Note that at most `trial` states will be computed thanks to the function `take` and Haskell's lazy evaluation strategy. Once this list has been constructed, it is passed to the function `runProgram` that successively starts a test run for each initial state until it finds a counterexample or the list of states is exhausted. The return type of this function is `Maybe ([Valuation], ([Valuation], BoolExpr, Int))`. It returns `Nothing` if it did not find a counterexample, and `Just (init, (state, assertion, linenr))` if it did find a counterexample. In the latter case, the valuation `init` is the initial state that satisfies the precondition, `state` is the offending state that has been reached, `assertion` is the assertion that failed, and `linenr` is the source line in the program where `assertion` is found. In the end, the function `verdict` converts the return value of `runProgram` in a human readable string format, which is echoed to the screen (standard output) using the function `echo` in line 2. Note that the semantics of the expression `'echo' str` is that `a` is evaluated and returned, with the side effect that `str` is printed to the screen. Hence, `result 'echo' result` prints the string `result` and also returns it.

```

runProgram :: Program -> [[Valuation]] ->
    Maybe ([Valuation], ([Valuation], BoolExpr, Int))
runProgram program [] = Nothing
runProgram program (val:vals)
  | test == Nothing = runProgram program vals
  | otherwise      = Just (val, sure test)
  where
    test = testValuation program val
    sure (Just x) = x

```

The key function of `runProgram` is the function `testValuation` that, given a program and an initial state (valuation), performs a simulation run. It returns `Nothing` if the run did not encounter any errors, otherwise it returns `Just (state, assertion, linenr)` where `state` is the offending state that has been detected, `assertion` is the failed assertion, and `linenr` is the corresponding line in the program.

```

1   testValuation :: Program -> [Valuation] ->
2       Maybe ([Valuation], BoolExpr, Int)
3   testValuation (Program consts vars code) val = run code val
4   where
5       run [] val = Nothing      -- no counter example found
6       run (((Assertion expr), nr):rest) val
7         | reduceBoolExpr expr val == Atom T = run rest val
8         | otherwise                        = Just (val, expr, nr)
9       run (((Comment text), nr):rest) val = run rest val
10      run (((Skip), nr):rest) val = run rest val
11      run (((Conditional guard thenPart elsePart), nr):rest) val

```

```

12 |   | checkConditional == Atom T = run (thenPart++rest) val
13 |   | otherwise = run (elsePart++rest) val
14 |   where checkConditional = reduceBoolExpr guard val
15 |   run ((WhileLoop guard body),nr):rest) val
16 |   | reduceBoolExpr guard val == Atom T =
17 |       run (body++[(WhileLoop guard body),nr]++rest) val
18 |   | otherwise = run rest val
19 |   run ((Assignment name expr),nr):rest) val =
20 |       run rest (update (Var name) expr val)
21 |   run ((Let name expr),nr):rest) val =
22 |       run rest (update (SpecConst name) reduceNumExpr expr val)
23
24 | update :: NumExpr -> NumExpr -> [Valuation] -> [Valuation]
25 | update name expr val =
26 |     (name, (reduceNumExpr expr val)) : [(n,e) | (n,e) <- val, n/=name]

```

The function `testValuation` uses the recursive helper function `run` which accepts a list of statements (code) and a valuation. Basically, this function implements an interpreter for the programming language of Aladin. The valuation denotes the state of the program during the simulation. The base case is the case that code is the empty list, meaning that the entire program has been executed. In that case `Nothing` is returned (line 5).

If we reach a code line which is an assertion (lines 6-8), then `reduceBoolExpr` is used to reduce the assertion. If the assertion reduces to `T` (true), then the program execution simply continues. Otherwise, an offending state has been detected and it is returned. Note that a nice side-effect of reducing expressions (instead of evaluating them) is that the use of uninitialized variables in assertions will be detected as an error (since these assertions do not reduce to `T`).

Running **skip** statements and comment lines is trivial. these statements do not change the state, so they are simply skipped (lines 9-10). In lines 11-14 the **if-then-else** construct is implemented. The guard is reduced using `reduceBoolExpr`. If it reduces to `T` (true), then the `thenPart` of the code is placed ahead of the rest of the code, and it is (recursively) executed. Otherwise, the `elsePart` is placed in front of the rest of the code, followed by its execution. Note that, due to Haskell's lazy evaluation strategy, this concatenation actually does not take place (and therefore no excessive memory is consumed). Lines 15-18 implement a **while** loop in a similar way that the **if-then-else** construct is interpreted.

Lines 19-20 implement an assignment statement, which clearly changes the state of a program (and hence, the valuation of the state). This update is performed by the function `update`, which accepts three arguments. The first argument is a `NumExpr` which represents the left hand side of the assignment (hence, it is of the type `Var x`). The second argument is the right hand side of the assignment, and the third argument is a valuation describing the state just before the assignment. The function `update` replaces the valuation for the corresponding variable by a new one that is obtained by reducing the right hand side expression, and returns it.

Lines 21-22 implement an assertion contain the **let** construct. As described in section 3.1, this is an artificial construct that does not change the state of any of the program variables. However, it does introduce a fresh specification constant, so it is part of the valuation of the program. So, for the interpreter there is only an artificial distinction between assignment and the introduction of a such a specification constant. Hence, the code in these lines is very similar to the code for a 'normal' assignment.

3.5 Example Runs

We conclude this chapter with the demonstration of two example runs. The first is a clearly incorrect implementation of the swap of two variables:

```

1 | VAR x, y;
2 | {x=X and y=Y}

```

```

3  x := y;
4  y := x;
5  {x=Y and y=X}

```

If we run Aladin with this program as its input, we get the following output:

```

Assertion failed on line 5: x=Y and y=X
Initial state: Y=-31 y=-31 X=0 x=0
Current state: y=-31 x=-31 Y=-31 X=0

Counter example found.

```

The following code fragment is clearly a correct implementation.

```

1  VAR x, y, z;
2    {x=X and y=Y}
3  z := x;
4  x := y;
5  y := z;
6    {x=Y and y=X}

```

If we run Aladin with this program as its input, we get the following output:

```

Passed 10000 tests.
## Proving: [x=X and y=Y] -> [y=Y and x=X]
#### Proving: [x=X and y=Y] -> [y=Y]
#### Success
#### Proving: [x=X and y=Y] -> [x=X]
#### Success
## Success
Annotation is correct.

```

Clearly the program passed 10000 tests. Moreover, the annotation was checked by Aladin for correctness, and was approved. In the following chapters, we will discuss how Aladin is able to verify the formal correctness of the annotation.

Chapter 4

Unification

In order to formally verify an annotation, Aladin uses rules from a knowledge base. We described earlier that knowledge base rules contain bound variables, which we call *unifiable variables*. These unifiable variables are in fact place holders for which we can substitute expressions that contain program variables, specification constant, and other unifiable variables. As discussed in section 1.4.3 we distinguish unifiable variables from other identifiers by prefixing their names with the # symbol.

For example, if the knowledge base contains a rule like $\#a \leq \#b$ and $\#b \leq \#c \rightarrow \#a \leq \#c$ which expresses the transitivity of the \leq relation, then we may conclude $x+Y \leq 42$ from $x+Y \leq Z$ and $Z \leq 42$. This is the consequence of a pattern matching process that discovers that we can substitute the sub-expression $x+Y$ for the unification variable $\#a$, Z for $\#b$, and 42 for $\#c$. As described in section 1.4.4, we denote this substitution as $\{\#a/x+Y, \#b/Z, \#c/42\}$ which is the *most general unifier* of $\#a \leq \#b$ and $\#b \leq \#c$ (the premise of the knowledge base rule) and $x+Y \leq Z$ and $Z \leq 42$.

Note that the conclusion that $x+Y \leq 42$ consists actually of two steps. First we need to match the premise of the KB rule with $x+Y \leq Z$ and $Z \leq 42$. This process is called *unification*, and is the topic of this chapter. The second step involves inferring the result (i.e. applying the KB rule), which is done using resolution and will be discussed in chapter 5.

4.1 Unification Algorithm

In this section we will discuss the implementation of a unification algorithm. In fact, the algorithm always finds the most general unifier, if a unifier exists. From now on, if we use the term unifier, we in fact mean a most general unifier (unless explicitly stated otherwise). The implementation is based on the unification algorithm that was published by Martelli and Montanari in 1982 (see [16]). However, that algorithm was developed for the implementation of the Prolog ([9]) programming language, which is based on pure first-order logic (FOL). This means that numeric expressions, and their comparisons are not incorporated in that algorithm. On the other hand, the original version of the algorithm allowed unification of functions (and their arguments), which Aladin does not support (currently).

We start by introducing the type `Substitution`, which represents a substitution for a single unifiable variable by an expression. The type is a simple pair, where the first element is the unifiable variable, and the second is the corresponding expression.

```
1 type Substitution = (UnifiableVar, NumExpr)
```

We implemented a unification algorithm for `BoolAtoms` only. A more general unification algorithm for `BoolExprs` can be built on top of it, but we do not need that in the implementation of Aladin since the resolution algorithm needs to match atoms only. The aim of the algorithm is to return, given two `BoolAtoms`, a unifier (which is a list of substitutions) if it exists, or signal failure if a unifier does not exist. So the return type of the algorithm is similar to the `Maybe` data type. However, we preferred to make failure more explicit and therefore implemented explicitly our own variant of the `Maybe` data type. Note that the

empty list is a valid unifier, since it is a unifier for two identical expressions. Hence, the empty list can not be used to signal failure. We use the following data type `Unifier` to store unifiers.

```
1 data Unifier = Fail | Success [Substitution]
```

The function `mguAtom` is the top-level function of the unification process. It takes two `BoolAtoms`, and it returns a `Unifier`. The following haskell code snippet implements the function `literalMGU` of Figure 1.3:

```
1 mguAtom :: BoolAtom -> BoolAtom -> Unifier
2 mguAtom (Compare Equal p q) (Compare Equal r s) =
3   mguCommutativeNumExpr p q r s (Success [])
4 mguAtom (Compare NotEqual p q) (Compare NotEqual r s) =
5   mguCommutativeNumExpr p q r s (Success [])
6 mguAtom (Compare op1 p q) (Compare op2 r s)
7   | op1 == op2 = mguNumExpr q s (mguNumExpr p r (Success []))
8   | otherwise = Fail
9 mguAtom p q
10  | p == q      = Success []
11  | otherwise = Fail
```

We discuss now the processing of `mguAtom e0 e1`. If `e0` and `e1` are both comparisons, then they only match if they both use the same comparison operator (see lines 6-8). If they do not, then the returned unifier is `Failure`. If they do, then we have to unify the operands of the comparison operator, which are numeric expressions. The function `mguNumExpr` computes the unification of two `NumExprs`. It, in contrast to `mguAtom`, accepts three arguments. The first two are numeric expressions that need to be unified, while the third is an accumulating unifier (i.e. a unifier that gets built up during the unification process). Initially this unifier is empty (i.e. `Success []`). If we want to unify two atoms $\alpha \oplus \beta$ and $\gamma \oplus \delta$, where \oplus denotes the same binary operator, then we first unify the numeric expression α with the numeric expression γ . The returned unifier is the unifier built up this far, and therefore the third argument of `mguNumExpr` in the unification of β with δ . If either `e0` or `e1` is not a comparison (i.e. `T` or `F`) then `e0` and `e1` only unify if they are equal. In that case, no substitution is needed and the empty unifier is returned (line 10).

Note that we have implemented two special cases in the lines 2-5. The reason is that we would like to be able to match for example $\alpha + \beta = \gamma$ with $x = y + z$. The unification process in lines 6-8 will fail for such a case, because the left hand side of both expressions do not match with each other (nor do the right hand sides). However, we wish to find the unifier $\{\alpha/y, \beta/z, \gamma/x\}$. This can be solved by introducing rules like $\#a=\#b \rightarrow \#b=\#a$ in the knowledge base, however we prefer to keep the knowledge base as small as possible (in view of the combinatorial explosion that may happen later in the resolution process). So, we solve this problem by introducing a variant of `mguNumExpr` called `mguCommutativeNumExpr` that tries to unify $\alpha + \beta$ with x and γ with $y + z$. After failure, it tries to unify $\alpha + \beta$ with $y + z$ and γ with z , which will succeed. The implementation of `mguCommutativeNumExpr` is as follows:

```
1 mguCommutativeNumExpr :: NumExpr -> NumExpr -> NumExpr -> NumExpr
2                       -> Unifier -> Unifier
3 mguCommutativeNumExpr lhs0 rhs0 lhs1 rhs1 theta =
4   let unifier = mguNumExpr lhs0 lhs1 theta in
5   if unifier == Fail
6   then mguNumExpr rhs0 lhs1 (mguNumExpr lhs0 rhs1 theta)
7   else mguNumExpr rhs0 rhs1 unifier
```

The functions `mguAtom` and `mguCommutativeNumExpr` make use of the function `mguNumExpr`, which is the main function for unifying two numeric expressions. The following Haskell snippet shows its code, which implements the function `recursiveMGU` of Figure 1.3. As mentioned earlier, the third argument of `mguNumExpr` is a unifier that represents a unifier that was built up during the recursive unification process. Hence, if we want to find a unifier of two numeric expression `e0` and `e1`, we need to call it as `mguNumExpr e0 e1 (Success [])`, since at the top level of the recursion no unifier has been

built up yet.

```

1 mguNumExpr :: NumExpr -> NumExpr -> Unifier -> Unifier
2 mguNumExpr _ _ Fail = Fail
3 mguNumExpr (Binary Add p q) (Binary Add r s) theta =
4   mguCommutativeNumExpr p q r s theta
5 mguNumExpr (Binary Mult p q) (Binary Mult r s) theta =
6   mguCommutativeNumExpr p q r s theta
7 mguNumExpr (Binary binop1 p q) (Binary binop2 r s) theta
8   | binop1 == binop2 = mguNumExpr q s (mguNumExpr p r theta)
9   | otherwise       = Fail
10 mguNumExpr (UniVar name) e2 theta = unifyVar name e2 theta
11 mguNumExpr e1 (UniVar name) theta = unifyVar name e1 theta
12 mguNumExpr p q theta
13   | p == q = theta
14   | otherwise = Fail

```

Clearly, if during the recursive process the unification of sub-expressions has failed, then the overall unification fails (line 2). The lines 3-9 try to unify two expressions which both consist of binary operators applied to their arguments. The code is very similar to the unification process that is used in `mguAtom`. Note that the hard coded strategy for commutative operators that is applied in `mguAtom` is also applied here (lines 3-6). The lines 10-11 are the most interesting part of this code. In these lines, we try to unify an expression that consists of only a unifiable variable with another expression. In that case, the function `unifyVar` is called. This function adds substitutions to the unifier θ (the unifier built up this far) if it is not already in the unifier. The function `unifyVar` is implemented as follows and is an implementation of the function `unifyVar` of Figure 1.3.

```

1 unifyVar :: UnifiableVar -> NumExpr -> Unifier -> Unifier
2 unifyVar var x (Success theta)
3   | e == Nothing = unifyVar' var x
4   | otherwise    = mguNumExpr (sure e) x (Success theta)
5 where
6   v = findSubst var theta
7   unifyVar' var (UniVar x)
8     | e' == Nothing = unifyVar'' var (UniVar x)
9     | otherwise    = mguNumExpr (UniVar var) (sure e') (Success theta)
10   where e' = findSubst x theta
11   unifyVar' var x = unifyVar'' var x
12   unifyVar'' var x
13     | occurCheck var x = Fail
14     | otherwise        = Success ((var,x):theta)
15
16 occurCheck :: UnifiableVar -> NumExpr -> Bool
17 occurCheck x (UniVar y) = x==y
18 occurCheck x (UnaryMinus e) = occurCheck x e
19 occurCheck x (Binary op e0 e1) = occurCheck x e0 || occurCheck x e1
20 occurCheck x _ = False
21
22 findSubst :: UnifiableVar -> [Substitution] -> Maybe NumExpr
23 findSubst v theta = fsub theta
24 where
25   fsub [] = Nothing
26   fsub ((w,expr):theta)
27     | v==w = Just expr
28     | otherwise = fsub theta

```

That this Haskell code implements the function `unifyVar` from the pseudo-code of Figure 1.3 is maybe not obvious. The call `unifyVar var x (Success theta)` tries to unify the unifiable variable `var` with the expression `x`, given the unifier `theta` built up so far. In line 6, we look up `var` in `theta`. If a substitution `(var, e)` is found in `theta` (we use the helper function `findSubst` for that), then `unifyVar` simply returns the unifier of `e` and `x` (line 4). The lines 3-6 are a concrete implementation of the line “**if** $v/e \in \theta$ **then return** `recursiveMGU(e, x, \theta)`,” from the pseudo code in Figure 1.3.

If a substitution `(var, e)` does not exist in `theta`, then we reach line 3 which jumps to the helper function `unifyVar'` (lines 7-10). The code of this function is similar to the code just described, and it implements the line “**if** $x/e \in \theta$ **then return** `recursiveMGU(v, e, \theta)`,” from the pseudo-code for the special case that the expression `x` is actually itself a unifiable variable.

If this is also not the case, then we reach line 12 (either via line 8, or via line 11). Here, we check whether `var` occurs in the expression `x` using the function `occurCheck`. If this is the case, then a unifier does not exist, and `Fail` is returned. This implements the pseudo-code “**if** v occurs in x **then return failure**,”. This check prevents circular substitution (i.e. replace `var` by an expressions containing `var`).

In the end, if `var` does not occur in `x`, then we can simply add the substitution `(var, x)` to `theta` and return it as the overall unifier (line 14). This implements “**return** $\theta \cup \{v/x\}$,”.

4.2 Applying a Unifier

It is clear that, once we have found a unifier for two atoms, we want to apply the unifier to infer new conclusions. For example, the premise of the rule $\alpha * (\beta + \gamma) = \delta \Rightarrow \alpha * \beta + \alpha * \gamma = \delta$ unifies with $x * (y + z) = w$ using the unifier $\theta = \{\alpha/x, \beta/y, \gamma/z, \delta/w\}$. So, we can *apply* this unifier to the right hand of the rule to infer $x * y + x * z = w$.

As mentioned before, Aladin only needs to apply unifiers to the elements of clauses, i.e. Boolean atoms. The application of a unifier to a `BoolAtom` is implemented in the function `applyMGU`.

```

1  applyMGU :: BoolAtom -> Unifier -> BoolAtom
2  applyMGU expr (Success theta) = mguApply expr theta
3  where
4    mguApply expr [] = expr
5    mguApply expr (substitution:theta) =
6      mguApply (mguApplyAtom expr substitution) theta
7
8  mguApplyAtom :: BoolAtom -> Substitution -> BoolAtom
9  mguApplyAtom T _ = T
10 mguApplyAtom F _ = F
11 mguApplyAtom (Compare op p q) subst =
12   Compare op (applySubstitution p subst) (applySubstitution q subst)
13
14 applySubstitution :: NumExpr -> Substitution -> NumExpr
15 applySubstitution (UnaryMinus p) subst =
16   UnaryMinus (applySubstitution p subst)
17 applySubstitution (Binary op p q) subst =
18   Binary op (applySubstitution p subst) (applySubstitution q subst)
19 applySubstitution (UniVar x) (y, expr)
20   | x == y      = expr
21   | otherwise = UniVar x
22 applySubstitution expr _ = expr

```

The call `applyMGU expr (Success theta)` applies the unifier `theta` to the atom `expr`. The function is basically the implementation of the recurrence rule:

$$Subst(\emptyset, E) = E \quad \text{and} \quad Subst(\{x_0/e_0, x_1/e_1, \dots, x_n/e_n\}, E) = Subst(\{x_1/e_1, \dots, x_n/e_n\}, [e_0/x_0]E)$$

Here, $[e/x]E$ denotes the expression E in which each occurrence of x has been replaced by e . In fact, this is exactly what the function `mguApplyAtom` implements for `BoolAtoms`, and `applySubstitution` for `NumExprs`.

4.3 Substitution of Equalities

Besides finding unifiers for atomic expressions, we also implemented substitution of equalities using unification. For example, if we have the expression $x * (y + z) = w$ and the rule $\alpha * (\beta + \gamma) = \alpha * \beta + \alpha * \gamma$, then we can use the unifier $\theta = \{\alpha/x, \beta/y, \gamma/z\}$ to unify the left hand sides of both expressions. Next, we can replace the left hand side $x * (y + z)$ by the inference $x * y + x * z$ resulting in $x * y + x * z = w$.

There are two major advantages of this technique. First, we can write much more natural rules in the knowledge base, like for example $\alpha * (\beta + \gamma) = \alpha * \beta + \alpha * \gamma$. This rule is much more natural than the equivalent rule $\alpha * (\beta + \gamma) = \delta \Leftrightarrow \alpha * \beta + \alpha * \gamma = \delta$, which needs an auxiliary unifiable variable δ . Moreover, after conversion to CNF format, this rule yields two clauses, while the rule $\alpha * (\beta + \gamma) = \alpha * \beta + \alpha * \gamma$ (which is already in CNF) yields only one singleton clause.

Second, the technique can be used to deal with sub-expressions. For example, consider again the rule $\alpha * (\beta + \gamma) = \alpha * \beta + \alpha * \gamma$ and the atomic expression $x + 2 * (y * (z + w)) = v$. Here, the matching sub-expression $y * (z + w)$ is nested inside a larger expression. Standard unification will not be able to match the entire atomic expression with the knowledge base rule. However, it is clear that the unifier $\theta = \{\alpha/y, \beta/z, \gamma/w\}$ can be used to replace the sub-expression $y * (z + w)$ by $y * z + y * w$ yielding the expression $x + 2 * (y * z + y * w) = v$. A naive way to tackle this problem is to put a special rule for a sub-expression like this in the knowledge base, but this is a very poor solution for two reasons. Extra rules in the knowledge base will produce (spurious) clauses during the resolution proving process, yielding a (much) longer execution time to (in)validate an annotation. Moreover, the problem with this approach is that sub-expressions that are nested at a deeper level will still not match the extra rule.

The top-level Haskell function for performing substitutions on a list of clauses (which has the type `CNF`) is `makeSubstitution`.

```

1  makeSubstitution :: [BoolAtom] -> CNF -> CNF
2  makeSubstitution [] clauses = clauses
3  makeSubstitution equalities clauses =
4    (nub.concat) [substitution equality clauses | equality <- equalities]
5
6  substitution :: BoolAtom -> CNF -> CNF
7  substitution equality clauses =
8    concat [substitutionEquality equality clause | clause <- clauses]
```

The function `makeSubstitution` takes a list of atoms, which is a list of equalities, and a list of clauses. If the list with equalities is empty, i.e. no substitutions can be performed, then the list of clauses is simply returned (line 2). Otherwise, `makeSubstitution` tries to apply each equality (i.e. a possible substitution) to all clauses using a list comprehension (see line 4). In this list comprehension, the function `substitution` is used that takes a single equality and a list of clauses, and returns all possible substitutions of the equation in all (sub-)expressions in the list of clauses. Note that this function returns a list of clauses, hence the result of this list comprehension is a list of lists. Therefore, `concat` is used to flatten this structure to a standard list. Moreover, `nub` is used to remove any duplicate clauses.

The same flattening is used in the function `substitution` which, given an equality, is a list comprehension over all the clauses. In this comprehension we call the function `substitutionEquality`, which again is a list comprehension over all the atoms in a clause.

```

1  substitutionEquality :: BoolAtom -> Clause -> CNF
2  substitutionEquality equality atoms =
3    combinations [substitutionAtom equality atom | atom <- atoms]
```

Note that in line 3 the helper function `combinations` is called. This function produces, given a list of lists, a list of lists of all possible combinations. For example, if we apply this function to a list strings, then the output will be a list of strings: `combinations combinations ["abc", "de"]` will produce `["ad", "ae", "bd", "be", "cd", "ce"]`.

The reason why the function `combinations` is needed in `substitutionEquality` is quite subtle. We explain the need using a small example. The function `substitutionAtom` (in line 3) performs the actual substitutions. It accepts two atoms, of which the first is the equality that may be substituted in the second argument (which is an atom from a clause), and returns a list of all possible substitution including the original atom. Now, if we feed `substitutionEquality` with the rule $\alpha * (\beta + \gamma) = \alpha * \beta + \alpha * \gamma$ and the clause $a * (x + y) = z \vee a * (x + y) = w$ then it produces in the list comprehension in line 3 the result `[[a*(x+y) = z, a*x+a*y = z], [a*(x+y) = w, a*x+a*y = w]]`. If we apply on this result the function `combinations`, then we end up with the list `[[a*(x+y) = z, a*(x+y) = w], [a*(x+y) = z, a*x+a*y = w], [a*x+a*y = z, a*(x+y) = w], [a*x+a*y = z, a*x+a*y = w]]`. Each element of this list is a valid substitution clause for the original clause.

The implementation of the function `substitutionAtom` is relatively straightforward. Note that only substitutions are applied to atoms which are comparisons. Any non-comparison atom is either trivially `True` or `False`, and thus no substitution can be applied (line 5).

```

1 substitutionAtom :: BoolAtom -> BoolAtom -> [BoolAtom]
2 substitutionAtom equality (Compare op r s) =
3     [Compare op r' s' | r' <- (r:(substituteAllSubExpr equality r)),
4                           s' <- (s:(substituteAllSubExpr equality s))]
5 substitutionAtom equality trivialTF = [trivialTF]
```

In the end, the function `substitutionAtom` uses the function `substituteAllSubExpr`. This function performs the actual substitution on a numeric expression. This function is implemented as follows:

```

1 substituteAllSubExpr :: BoolAtom -> NumExpr -> [NumExpr]
2 substituteAllSubExpr (Compare Equal lhs rhs) expr =
3     substituteAllSE (Compare Equal lhs rhs) expr ++
4     substituteAllSE (Compare Equal rhs lhs) expr
5 where
6     substituteAllSE (Compare Equal lhs rhs) expr = toplevel ++
7         substAll lhs rhs expr
8     where
9         unifier = mguNumExpr lhs expr (Success [])
10        toplevel = if unifier == Fail then [] else
11            [applyMGUNumExpr rhs unifier]
12        substAll :: NumExpr -> NumExpr -> NumExpr -> [NumExpr]
13        substAll lhs rhs (Binary op e0 e1) =
14            [Binary op e0 r | r <- rs] ++ [Binary op l e1 | l <- ls]
15            ++ [Binary op l r | l <- ls, r <- rs]
16        where
17            ls = substituteAllSubExpr (Compare Equal lhs rhs) e0
18            rs = substituteAllSubExpr (Compare Equal lhs rhs) e1
19        substAll lhs rhs (UnaryMinus e) = map UnaryMinus
20            (substituteAllSubExpr (Compare Equal lhs rhs) e)
21        substAll lhs rhs _ = []
```

The function `substituteAllSubExpr` takes as input an equality (an atom) and a numerical expression and returns a list of numerical expressions. Because $\alpha = \beta \Leftrightarrow \beta = \alpha$, we call `substituteAllSE` for both combinations (lines 3-4). The helper function `substituteAllSE` tries to recursively match the left hand side of the equality with the expression. If this matching yields a unifier, the unifier is applied to the right hand side of the equality and added to the list of possible substitutions (line 10-11). Note that, once the expression has been processed, the process recursively descends to one level deeper in the expression to

see if it can match (and substitute) sub-expressions. This is implemented in the function `substAll`. As an example of this process, consider the expression $x * y * (w + z) = u$ and the rule $\alpha * (\beta + \gamma) = \alpha * \beta + \alpha * \gamma$. The algorithm will first find the unifier $\theta_0 = \{\alpha/x * y, \beta/w, \gamma/z\}$ which matches on the top level. Next, the function `substAll` is called. Now, assume that in line 13, `e0` is x and `e1` is $y * (w + z)$, then we call for the expression $y * (w + z)$ the recursively function `substituteAllSubExpr` again, making the expression $y * (w + z)$ the top level expression in the next recursion level. This level will find the unifier $\theta_1 = \{\alpha/y, \beta/w, \gamma/z\}$. The process will not find any other unifiers. Hence, the result is that `rs` in line 14 is the singleton list $[y * w + y * z]$ and we use this expression instead of $y * (w + z)$ when we complete the expression again. We use the same structure for the expression `e0` but x does not match either side of the equality. Also, we use the same approach for expressions which contain a unary minus.

Chapter 5

Resolution

Aladin, like many theorem provers, uses the resolution technique as its main inference tool. We discussed in section 1.4 the principals of the resolution method. For a more extensive overview of resolution theorem proving, the reader is referred to [18]. The topic of this chapter is the implementation of the resolution algorithm in Aladin. Although Aladin is based on First Order Logic (extended with integer arithmetic), we implemented the algorithm closely following the pseudo code from Figure 1.1 (which is the algorithm for propositional logic).

5.1 Resolution Algorithm

Recall that we want to show, given a knowledge base KB and a goal predicate α , that $KB \models \alpha$. However, the resolution technique is based on proof by refutation which means that it tries to show that $KB \models \neg\alpha$ is unsatisfiable.

The top-level function for performing resolution proofs is the function `resolutionProof`, which takes two arguments and returns a Boolean value. The first argument is the knowledge base KB in CNF format, while the second argument is the goal predicate α . The knowledge base contains a set of (arithmetic) rules containing unifiable variables. The function returns `True` if it was able to prove that $KB \models \alpha$, i.e. it was able to infer the empty clause (i.e. **false**). It returns `False` otherwise. As mentioned before, the return value `False` does not necessarily mean that the goal cannot be proven. It might also mean that inferring the empty clause needs a deeper breadth first search than the fixed horizon that is set in Aladin. Moreover, it can also mean that more rules are needed in the knowledge base. The implementation of `resolutionProof` is given below:

```
1 resolutionProof :: CNF -> BoolExpr -> Bool
2 resolutionProof knowledgebase goal = bfsProof 0 [] notGoal
3   where
4     notGoal = reduceCNF (toCNF (Not goal))
5     bfsProof n kb clauses = bfs n kb clauses
6     bfs 5 _ _ = False {- search horizon reached -}
7     bfs n kb clauses
8       | done          = True
9       | otherwise    = bfsProof (n+1) kb' inferred'
10    where
11      (done,inferred) = resProof kb (reduceCNF clauses)
12      inferred'      = filter (not.containsUniVarClause) inferred
13      kb' = resolveAllPairs knowledgebase inferred
```

The second argument `goal` is a `BoolExpr` and not in CNF format yet. The reason is that we first need to negate the goal, and convert it to CNF afterwards. This is done in line 4. Note that after this conversion, the function `reduceCNF` is applied to the resulting set of clauses and the result is given the name `notGoal`.

The function `reduceCNF` tries to reduce and simplify clauses as much as possible. It will be discussed in section 5.3. It implements optimisations that are not essential for `resolutionProof`, but it speeds up the proving process significantly.

In line 2, `notGoal` is passed to the function `bfsProof`, which is the function that actually performs the breadth first search (BFS) resolution process. The function takes 3 arguments. The first being the depth of the BFS process, which is initially zero. Aladin stops searching for a proof after 5 BFS levels. This horizon can be changed by simply changing it in line 6 of the code. For the tests that we performed, the horizon value 5 was sufficient. Setting this value higher will not affect the execution time for correct annotations that do not need a larger horizon (since BFS search finds a shortest proof if it can be found within the horizon), but it will drastically increase the execution time for annotations for which Aladin is not able to find a proof (due to the combinatorial explosion of the number of inferred clauses).

The second argument of `bfsProof` plays the role of the knowledge base, and the third argument is the negation of the goal in (reduced) CNF format. Note that at BFS level zero, the knowledge base is actually empty (see line 2). Hence, in a first BFS iteration, we try to prove the goal without using the knowledge base. At first sight, this may seem odd. But there is a good reason for this decision. Consider for example the following fragment from some annotation:

```
{ x = X and y = Y }
  (* drop conjunct *)
{ x = X }
```

Dropping one or more conjuncts, like in this example, is a very common structure in program annotations. For this example, the system needs to prove the goal $x = X \wedge y = Y \Rightarrow x = X$, which converts after negation and conversion to CNF to $[[x=X], [y=Y], [x \neq X]]$. Obviously, the clause $[x=X]$ resolves with the clause $[x \neq X]$ to the empty clause, without the need of a knowledge base. We use this strategy because a proof that does not make use of the knowledge base is usually much faster than a proof that does make use of it.

In line 11, a tuple `(done, inferred)` is computed using the helper function `resProof`, which implements the actual resolution algorithm. The Boolean value `done` is `True` if `resProof` was able to find a proof, otherwise it is `False`. During this search for a proof, new clauses are (possibly) inferred. These new clauses are returned in `inferred`. If `resProof` returns `False`, then the function `bfsProof` is called with a raised BFS level, a new knowledge base `kb'`, and the third argument `inferred'` is the set of all newly inferred clauses that do not contain unification variables. Note that the second argument `kb'` is the result of resolving all clauses from the knowledge base with the newly inferred clauses. The reason that the function `resProof` has two arguments, a knowledge base and a set of inferred clauses, is that we do not want to resolve the knowledge base with itself. We only want to resolve inferred clauses with clauses from the knowledge base.

The resolving process takes place in `resProof`, which resembles the pseudo code from Fig. 1.1.

```
1 resProof :: CNF -> CNF -> (Bool, CNF)
2 resProof clauses inferred
3   | [] `elem` inferred = (True, clauses')
4   | inferred == []      = (False, clauses)
5   | otherwise          = resolutionProof' clauses' inferred''
6   where
7     clauses' = clauses ++ inferred
8     inferred' = resolveAllPairs clauses inferred
9     clausesWithSubstitution = makeSubstitution
10      (equalities (clauses++inferred++inferred')) (inferred'++inferred)
11     inferred'' = filter (\x -> (not(x `elem` clauses) &&
12      not (x `elem` inferred) && not (x `elem` inferred')))
13      clausesWithSubstitution
```

The function `resProof` implements the loop of Figure 1.1 using recursion. In the loop of that algorithm, the resolvents of each pair of clauses from the set of currently known clauses is computed. Our implementation does not do that, because it is inefficient to do this for pairs of clauses that were already processed (paired) in a previous iteration. This is exactly the reason why `resProof` has two arguments, the first is `clauses` which is the set of known clauses thus far, and the second is `inferred` which is the set of newly inferred clauses from the previous iteration of the algorithm. In each iteration, we only compute from pairs of clauses which have not been paired before. This is done in the function `resolveAllPairs` (see line 8), which will be discussed hereafter.

The algorithm stops successfully if the empty list is in the set of newly inferred clauses (line 3), since the empty clause represents **false** (hence a contradiction). A proof fails (i.e. `False` is returned) if the set `inferred` of newly inferred clauses is empty (line 4). Note that this differs from the pseudo code in Figure 1.1 which stops if the set of newly inferred clauses is a subset of the already known clauses. Testing whether an unordered list of clauses is a subset of another unordered list of clauses is quite computationally expensive. For that reason, we maintain the invariant that the intersection of `clauses` and `inferred` (also `inferred'` and `inferred''`) is empty. In other words, we make sure that `inferred` (also `inferred'` and `inferred''`) only contains newly discovered clauses. Hence, we can replace the subset test by the simple (and cheap) test whether `inferred` is empty.

On those newly obtained clauses, we apply the function `makeSubstitution` which we discussed in Chapter 4, yielding an extended set of inferred clauses named `clausesWithSubstitution`. From this extended set, we construct the set `inferred''` which does not contain any clauses which were already known before (using a `filter` operation in line 11-13). If no new clauses are found, i.e. `inferred''` is empty, then `resProof` will terminate in the next recursive call of `resProof`, in which the set of known clauses (`clauses'` in lines 5 and 7) is the set union of `inferred` and `clauses` and the list of inferred clauses is `inferred''`.

What remains to discuss is the function `resolveAllPairs`.

```

1  resolveAllPairs :: CNF -> CNF -> CNF
2  resolveAllPairs clauses inferred
3  | ininf      == [[]] = [[]]
4  | clausesinf == [[]] = [[]]
5  | otherwise      = clausesinf ++ ininf
6  where
7    ininf      = resolvePairs (pairs inferred) clauses
8    known'     = ininf ++ clauses
9    clausesinf = resolvePairs [(c,d) | c<-inferred, d<-clauses] known'
10
11 resolvePairs :: [(Clause,Clause)] -> CNF -> CNF
12 resolvePairs pairs alreadyKnown = resPairs pairs []
13 where
14   resPairs [] resolvents = resolvents
15   resPairs ((c,d):pairs) resolvents
16     | new == [[]] = [[]]
17     | otherwise   = resPairs pairs (new ++ resolvents)
18   where
19     res = resolveTwoClauses c (standardizeApartFromClause c d)
20     new = discoveries (discoveries res resolvents) alreadyKnown
21     discoveries cs known = filter (not.('elem' known)) cs

```

The function `resolveAllPairs` takes two list of clauses (i.e. CNF format) and returns a list of resolvent clauses. It makes use of the helper function `resolvePairs` which accepts a list of pairs of clauses and a list of already known clauses, and it produces the actual newly discovered resolvents. In other words, the main task of `resolveAllPairs` is to produce the list of pairs of clauses that need to be resolved, and pass it to `resolvePairs`. These lists are produced in the lines 7 and 9. In line 7, a helper function `pairs` is used that takes a list and produces a list of all possible pairs that can be constructed from that list. In this line, a list of pairs of clauses is produced that are taken from newly obtained inferences from

the previous iteration of the resolving algorithm. This list is passed to `resolvePairs` and the resulting list of resolvents is named `infinf`. Note that, if any of the pairs that are passed to `resolvePairs` resolves to the empty clause, then `resolvePairs` returns a singleton list containing the empty clause (i.e. `[]`). This is a short-cut optimisation. Once an empty clause has been found, there is no need to compute resolvents any further.

So, if `infinf` equals `[]`, then `resolvePairs` itself returns immediately `[]` (line 3). Otherwise, each clause of `inferred` gets resolved with each clause from `clauses` using a list comprehension (line 9). The result is named `clausesinf`. If this list equals the singleton list `[]` (see line 4), then `resolvePairs` returns immediately `[]` and the proof is completed. When neither is true, we return the set of newly discovered resolvents, being `clausesinf++infinf` (line 5).

As mentioned before, the function `resolvePairs` takes a list of pairs of clauses, and resolves these pairs one by one. The function has an extra argument being a list of clauses that are already known. This extra list is necessary in maintaining the invariant that `inferred` only contains newly discovered clauses (lines 20-21). For each pair we call the function `resolveTwoClauses`, which we will discuss in section 5.2. Again, note that the processing stops as soon as an empty clause is found, and `[]` is returned (line 16). Otherwise it continues resolving the remaining clauses (line 17). Note that we call in this function the helper function `standardizeApartFromClause c d` (in line 19). This function returns a clause which is equivalent to the clause `d`, but all unifiable variables have been renamed such that they are disjoint with the set of unifiable variables in the clause `c`.

5.2 Resolving Two Clauses

In this section, we discuss the implementation of the function `resolveTwoClauses` which takes two clauses, and returns their resolvents. We implemented `resolveTwoClauses` as follows:

```

1  resolveTwoClauses :: Clause -> Clause -> [Clause]
2  resolveTwoClauses c0 c1
3    | [] `elem` resolvents = []
4    | otherwise           = resolvents
5    where
6      resolvents = nub (resolveTwoClauses' c0 c1)
7
8  resolveTwoClauses' :: Clause -> Clause -> [Clause]
9  resolveTwoClauses' c0 c1 = filterTrivialClauses (nub
10    (applyMGUOnResolvents c0 c1 (makeEachCombiUnifier c0 c1)))
11  where
12    makeEachCombiUnifier :: Clause -> Clause ->
13      [((BoolAtom, BoolAtom), Unifier)]
14    makeEachCombiUnifier cs ds = filter ((/= Fail).snd) [((c, d),
15      (complementaryMGU c d)) | c <- cs, d <- ds]
16    -----
17    applyMGUOnResolvents :: Clause -> Clause ->
18      [((BoolAtom, BoolAtom), Unifier)] -> [Clause]
19    applyMGUOnResolvents _ _ [] = []
20    applyMGUOnResolvents cs ds ((c, d), unifier):rest =
21      (applyMGUClauses ((filter (/=c) cs) ++ (filter (/=d) ds))
22        unifier):(applyMGUOnResolvents cs ds rest)
23

```

Note that the function `resolveTwoClauses` returns a list of clauses. We return a list of clauses and not a single clause. The reason is that it may be possible to resolve two clauses in multiple ways. For example, the clauses $x * (y + z) \neq w \vee r * (s + t) \neq q$ and $\alpha * (\beta + \gamma) = \delta \vee \alpha * \beta + \alpha * \gamma = \delta$ have two pairs of complementary atoms that can be used to apply the resolution rule. The atom $\alpha * (\beta + \gamma) = \delta$ matches with both the complementary atoms $x * (y + z) \neq w$ and $r * (s + t) \neq q$, both yielding different resolvents.

Again, note that `resolveTwoClauses` returns the singleton list containing only the empty list as soon as an empty clause is found. We use this shortcut because once we found `False` (the empty clause) we are not interested in the rest anymore and this saves a lot of time. If it does not find an empty list, the function returns the result of the function `resolveTwoClauses'`. The function `resolveTwoClauses'` uses the function `complementaryMGU` to find a unifier for two complementary clauses.

```

1  complementaryMGU :: BoolAtom -> BoolAtom -> Unifier
2  complementaryMGU T F = Success []
3  complementaryMGU F T = Success []
4  -- a/=b is complemented by a=b
5  complementaryMGU (Compare NotEqual a b) (Compare Equal a' b') =
6      mguAtom (Compare Equal a b) (Compare Equal a' b')
7  -- a <= b is complemented by b < a
8  complementaryMGU (Compare LessEqual a b) (Compare LessThan a' b') =
9      mguAtom (Compare LessThan a b) (Compare LessThan b' a')
10 -- a=b is complemented by any of a/=b, a<b, b<a
11 complementaryMGU (Compare Equal a b) (Compare NotEqual a' b') =
12     mguAtom (Compare Equal a b) (Compare Equal a' b')
13 complementaryMGU (Compare Equal a b) (Compare LessThan a' b') =
14     mguAtom (Compare Equal a b) (Compare Equal a' b')
15 -- a < b is complemented by any of a=b, b<=a, b<a
16 complementaryMGU (Compare LessThan a b) (Compare Equal a' b') =
17     mguAtom (Compare Equal a b) (Compare Equal a' b')
18 complementaryMGU (Compare LessThan a b) (Compare LessEqual b' a') =
19     mguAtom (Compare LessThan a b) (Compare LessThan a' b')
20 complementaryMGU (Compare LessThan a b) (Compare LessThan b' a') =
21     mguAtom (Compare LessThan a b) (Compare LessThan a' b')
22 -- everything else: Fail
23 complementaryMGU _ _ = Fail

```

Note that the function `complementaryMGU` is not strict regarding complements. For example, the strict complement of $a < b$ is $b \leq a$, however the function also accepts any of $a = b$, $b \leq a$, or $b < a$ as the complement of $a < b$.

5.3 Filtering and Reducing Clauses

Resolution may generate useless clauses. For example, clauses that have the structure $\alpha \vee \beta \vee \gamma$ and $\neg\alpha \vee \neg\beta \vee \delta$ will resolve to either $\alpha \vee \neg\alpha \vee \gamma \vee \delta$ or $\beta \vee \neg\beta \vee \gamma \vee \delta$. In both cases, the resolvent contains the structure $P \vee \neg P$ which is trivially true. Since we cannot deduce anything from true, we can simply omit resolvents like these. Moreover, due to the introduction of substitution of qualities, we can construct trivial clauses as well. For example, a knowledge base rule $\alpha * (\beta + \gamma) = \alpha * \beta + \alpha * \gamma$ together with the clause $x * (y + z) \neq w \vee x * y + x * z = w$ will produce (after unification and substitution) the clause $x * y + x * z \neq w \vee x * y + x * z = w$. Again, this clause has the form $P \vee \neg P$, and can be omitted.

To remove trivial clauses, we implemented the functions `reduceCNF` and `FilterTrivialClauses`. The function `reduceCNF` takes a set of clauses (i.e. of the type `CNF`) and returns an equivalent set of clauses in which numeric expressions have been simplified as much as possible (e.g. expressions like $3 * (7 + 7) * x$ are simplified to $42 * x$) and trivial clauses have been removed. The simplification of numeric expressions in a set of clauses is performed by the function `simplifyCNF`, which takes a set of clauses and traverses through the set while applying the numeric simplification functions that were discussed in section 3.2.1.

```

1  reduceCNF :: CNF -> CNF
2  reduceCNF clauses = filterTrivialClauses(simplifyCNF clauses)

```

After simplification of numeric expressions, the function `filterTrivialClauses` takes care of remov-

ing clauses that are trivially true. It takes a list of clauses and returns an equivalent (but possible shorter) list of clauses.

```

1 filterTrivialClauses :: CNF -> CNF
2 filterTrivialClauses clauses =
3     map removeSelfComparisons (filterTrueClauses clauses)

```

The function `filterTrivialClauses` first uses the function `filterTrueClauses` which removes clauses that are trivially true. The output is passed to the function `removeSelfComparisons` which we will discuss later. First we discuss the function `filterTrueClauses`:

```

1     filterTrueClauses :: CNF -> CNF
2     filterTrueClauses clauses =
3         filter (not.isTrueClause) clausesWithoutSelfComparisons
4     -----
5     clausesWithoutSelfComparisons =
6         filter (not.containsTrueSelfCompare) clauses
7     containsTrueSelfCompare :: Clause -> Bool
8     containsTrueSelfCompare clause = any isTrueSelfCompare clause
9     isTrueSelfCompare (Compare op p q) =
10         (op==Equal || op==LessEqual) && p==q
11     isTrueSelfCompare _ = False

```

In `filterTrueClauses` clauses are removed which are trivially true. First it removes clauses which contain atoms that are trivial self comparisons of the form $\alpha = \alpha$ or $\alpha \leq \alpha$ from the list. This is done using a filter that uses the Boolean functions `containsTrueSelfCompare`, which returns `True` if and only if a clause contains a trivial self comparison, and `isTrueSelfCompare` (lines 5-11).

After removal of trivial self comparisons, the function `filterTrueClauses` filters the result using the Boolean function `isTrueClause`. Its implementation is given in the following code snippet.

```

1 isTrueClause :: Clause -> Bool
2 isTrueClause clause = or [isComplement p q | (p,q) <- pairs clause]
3 -----
4 isComplement :: BoolAtom -> BoolAtom -> Bool
5 isComplement (Compare LessThan p q) (Compare LessEqual r s) =
6     p==s && q==r
7 isComplement (Compare LessEqual p q) (Compare LessThan r s) =
8     p==s && q==r
9 isComplement (Compare Equal p q) (Compare NotEqual r s) =
10     (Compare Equal p q) == (Compare Equal r s)
11 isComplement (Compare NotEqual p q) (Compare Equal r s) =
12     (Compare Equal p q) == (Compare Equal r s)
13 isComplement (Compare _ _ _) _ = False
14 isComplement T F = True
15 isComplement T _ = False
16 isComplement F F = True
17 isComplement F _ = True

```

The function `isTrueClause` accepts a clause and returns a Boolean value. It returns `True` if a clause contains complementary atoms. For example, the atoms $\alpha < \beta$ and $\beta \leq \alpha$ are complementary atoms. Also, $\alpha = \beta$ and $\alpha \neq \beta$ are complementary atoms. If two of these complementary atoms exist in a clause, then the clause is trivially true. Hence, clauses like these can be removed. Note that the trivial literals `True` and `False` are also considered in this function.

After the removal of trivial true clauses, the function `filterTrivialClauses` passes the filtered list of clauses to the function `removeSelfComparisons`.

```
1 removeSelfComparisons clause = filter (not.isSelfCompare) clause
2 isSelfCompare :: BoolAtom -> Bool
3 isSelfCompare (Compare op p q) = (op==LessThan||op==NotEqual)&&p==q
4 isSelfCompare _ = False
```

The function `removeSelfComparison` takes a clause, and removes from it atoms of the form $\alpha \neq \alpha$ and $\alpha < \alpha$. Atoms like these are clearly false. Recall that a clause is a disjunction of its elements, so false atoms can safely be removed from a clause.

Chapter 6

Verification of an Annotated Program

As described in Chapter 1, the goal of this project is to verify the correctness of an annotated program. We already discussed how Aladin parses (and stores) programs, and runs simulations in an attempt to generate counterexamples. We also discussed the unification algorithm and the resolution algorithm on which proofs generated by Aladin are based. However, we did not discuss yet how these ingredients are used to verify an annotated program. This is the topic of this chapter.

6.1 Verifying Correctness

The top-level function that is used to verify an annotated program is the function `proofProgram`:

```
1 proofProgram :: Program -> Bool
2 proofProgram program = proofAnnotation (removeSkipCommands program)
```

The function `proofProgram` takes a `Program` and returns a `Bool`. It returns `True` if it is able to verify the correctness of the program, otherwise it returns `False`. Recall that the return value `False` may mean that Aladin was not able to find a correctness proof even though it exists. This can be the result of a knowledge base containing insufficient rules to establish a proof, or a proof needs more breadth first search layers in the resolution proof tree than the horizon set by Aladin.

A `Program` may contain several comment lines and lines containing the **skip** command, which do not play any (formal) role in the annotation nor do they change the state of the program. Therefore, as an initializing step, `proofProgram` removes these lines using the function `removeSkipCommands`. The output is an equivalent program without comment lines and **skip** commands, which is the input for the actual proving process that is performed by the function `proofAnnotation`. Note that the function `removeSkipCommands` cannot be a simple filter applied to the list of statements that the program consists of, because it must recursively filter the bodies of conditional statements and loops.

```
1 removeSkipCommands (Program constants variables code) =
2   (Program constants variables (rmSkip code))
3   where
4     rmSkip [] = []
5     rmSkip ((Skip, _):code) = rmSkip code
6     rmSkip ((Comment _, _):code) = rmSkip code
7     rmSkip ((Conditional guard thenPart elsePart, n):code) =
8       (Conditional guard
9        (rmSkip thenPart) (rmSkip elsePart), n):rmSkip code
10    rmSkip ((WhileLoop guard body, n):code) =
11      (WhileLoop guard (rmSkip body), n):rmSkip code
12    rmSkip (statement:code) = statement:rmSkip code
```

After the removal of **skip** statements and comment lines, the actual verification of the annotation is performed by the function `proofAnnotation`.

```

1 proofAnnotation :: Program -> Bool
2 proofAnnotation (Program constants variables code) =
3   proof (findPrecondition code) (tail code)
4   where
5     findPrecondition :: [(Statement,Int)] -> BoolExpr
6     findPrecondition ((Assertion expr,nr):_) = expr
7     findPrecondition _ = error ("No precondition found :(")

```

Recall from the discussion of the structure of a program (in section 3.1) that we decided that `Programs` consist of blocks of code and that a program has to start with an assertion (its precondition). Hence, after removal of all comment lines and **skip** statements, the first statement of the `code` should be the precondition. If the first statement is not an assertion (i.e. not the precondition), then Aladin will output an error and abort. The precondition is found in the lines 5-7. When the first statement is the precondition, the actual proving process is started in the function `proof`.

```

1 proof :: BoolExpr -> [(Statement, Int)] -> Bool
2 proof precondition [] = True

```

The function `proof` takes two arguments. The second argument is a list of statements (and their line numbers) and the first is the precondition of this list of statements. The function is recursive, and processes on each recursive call a block of the list of statements. Hence, the first arguments is either the precondition of the entire program, or the precondition of a block, which is basically the postcondition of the preceding block. We start with the base case of this function, which is clearly the case in which the list of statements is empty. This means that the end of the program has been reached, and the function returns `True` since all preceding blocks have been proven correct (line 2).

```

1 proof precondition ((Assertion expr,nr):code)
2   | proofImplication precondition expr = proof expr code
3   | otherwise = False

```

The next case we consider is the case that the first statement of the list of statements is an assertion. Hence, we need to show that the precondition implies this assertion. The function `proofImplication` is used to prove implications. It accepts two `BoolExprs`, a premise p and a conclusion q , and it returns `True` if it is able to prove that $p \rightarrow q$, otherwise it returns `False`. If the proof fails, then the proving process stops (line 3), otherwise it continues using the assertion (the conclusion of the implication) as the new precondition (line 2). We will discuss the implementation of the function `proofImplication` after the complete discussion of the function `proof` in section 6.2.

```

1 proof precondition ((Assignment lhs rhs,nr):code)
2   | proofImplication precondition weakest = proof postcondition rest
3   | otherwise = False
4   where
5     (assignments,postcondition,rest) =
6       findAssignments ((Assignment lhs rhs,nr):code)
7     weakest = wps assignments postcondition

```

When the first statement of the list of statements is an assignment, then we deal with a block S containing one or more assignments, followed by the postcondition Q of the block. Aladin first computes the weakest precondition `weakest` of the block S with postcondition Q , i.e. $wp(S, Q)$ (see section 1.3.4), using the function `wps`. We postpone the discussion of the implementation of `wps` until the entire function `proof` has been discussed (see section 6.3). Next, it needs to prove that `precondition` implies `weakest` (in line 2). If this proof succeeds, the recursive proof process continues, with Q playing the role of the precondition of the rest of the statement list. Otherwise, `False` is returned and the proving process stops (line 3).

Note that we made use of the function `findAssignments` that returns a triple, of which the first element is the block of assignments S , the second element the postcondition Q , and the third element the rest of the code.

```

1  findAssignments :: [(Statement,Int)] ->
2    [(String,NumExpr)], BoolExpr, [(Statement,Int)]
3  findAssignments code
4    | [] == rest = error ("No postcondition found.")
5    | not(isAssertion (head rest)) = error ("In line "
6      ++ show (snd(head rest)) ++ " should be an assertion.")
7    | otherwise = (assignments, assertionToBoolExpr(head rest),
8      tail rest)
9  where
10    assignments = map toPair (takeWhile isAssignment code)
11    rest = dropWhile isAssignment code
12    toPair ((Assignment lhs rhs), _) = (lhs, rhs)

```

The assignments of the block S are returned as a list of pairs, of which the first element is a `String` which is the name of the variable on the left hand side of an assignment. The second element is a `NumExpr` which is the right hand side of the assignment. The assignments of the code are collected using the function `takeWhile`, and converted using `toPair` to the format `(String, NumExpr)`. When there is no assertion at the end of the block of assignments, the function will report an error. We distinguish two different error cases: if there is no code left, the error is "No postcondition found.", but if there is code left then the error "In line X there should be an assertion." is reported.

We continue with the discussion of the function `proof` for the case that we are dealing with a **let** statement. Recall that this statement is actually not a real statement, but a way to introduce locally a specification constant. A typical use of this feature is in the proof of termination of loops using a variant functions (see section 3.1). The processing of **let** statements is easy, because we can simply continue the proving process (i.e. call `proof`) with an extended precondition. The extended precondition is the conjunction of the current precondition with the extra condition of the **let** statement (line 2). For example, if we have at some point in the program the assertion $\{p : x = X \wedge y = Y\}$ and we come across the **let** statement $\{\text{let } V=E\}$ (where E is an expression) then we call `proof` with the extended precondition $\{x = X \wedge y = Y \wedge vf = E\}$ and the remaining code.

```

1  proof precondition ((Let specconst expr, _):code) =
2    proof (BinOp And precondition extension) code
3    where extension = Atom (Compare Equal (SpecConst name) expr)

```

The next case that we consider is an **if-then-else** construct, which is more complicated than the other cases that we have discussed thus far. We start with the prove rule for this construct. The Hoare triple

$$\{P\} \text{ if } B \text{ then } S_0 \{Q_0\} \text{ else } S_1 \{Q_1\} \text{ end; } \{Q\}$$

is correct if and only if all of the following requirements are met:

- $\{P \wedge B\} S_0 \{Q_0\}$
- $\{P \wedge \neg B\} S_1 \{Q_1\}$
- $Q_0 \Rightarrow Q$
- $Q_1 \Rightarrow Q$

Hence, instead of producing a single proof, Aladin has to produce four sub-proofs in the verification of the validity of an **if-then-else** construct. The implementation is shown in the following code snippet. The first requirement is checked in line 3, the second in line 4, the third in the lines 5-6, and the last requirement is checked in the lines 7-8. The function `last` that is used in the lines 5 and 7 is a helper function that returns the postcondition of a code block (i.e. the local postconditions Q_0 and Q_1 in the above proof rule). Note that the function `proof` returns the conjunction of the four proofs. However, as soon as any of these proofs

fails, then the remaining cases are not evaluated (due to lazy evaluation) and the proving process stops and `proof` returns `False`. If all these cases pass, then the proving process continues in line 9, which processes the rest of the code.

```

1 proof precondition ((Conditional guard thenPart elsePart,nr):
2                     (Assertion postcondition,_) :code) =
3     proof (BinOp And precondition guard) thenPart &&
4     proof (BinOp And precondition (Not guard)) elsePart &&
5     proofImplication (assertionToBoolExpr (last thenPart))
6     postcondition &&
7     proofImplication (assertionToBoolExpr (last elsePart))
8     postcondition &&
9     proof postcondition code

```

There is one more construct to consider in the function `proof`, which is the **while** construct. The Hoare triple

$$\{P\} \text{ while } B \text{ do } S \{Q_0\} \text{ end}; \{Q\}$$

is correct if the following requirements are met:

- $\{P \wedge B\} S \{Q_0\}$
- $Q_0 \Rightarrow P$
- $P \wedge \neg B \Rightarrow Q$

Usually the predicate Q_0 equals the predicate P , in which case P is called an *invariant* of the loop. The implementation is shown in the following code snippet. The first requirement is checked in line 3, the second requirement is checked in line 4, and the third in the lines 5-6. The structure of the code is similar to the structure of the **if-then-else** construct.

```

1 proof precondition ((WhileLoop guard body,n):
2                     (Assertion postcondition,_) :code) =
3     proof (BinOp And precondition guard) body &&
4     proofImplication (assertionToBoolExpr(last body)) precondition &&
5     proofImplication (BinOp And precondition (Not guard))
6     postcondition &&
7     proof postcondition code

```

Note that the given proof rule for a loop does not guarantee that the loop terminates. Termination can be shown by introducing the following extra proof rules. Let vf be an integer expression in terms of the program variables and specification constants. Then, the loop terminates if the following requirements hold.

- $P \wedge B \Rightarrow vf \geq 0$
- $\{P \wedge B \wedge vf = V\} S \{P \wedge vf < V\}$

The rationale behind these rules is that the expressions vf (called the variant function) decreases in each iteration, while P is kept invariant. Since the variant function is integer valued, it must at some point become negative. However, the first requirement says that the variant function is non-negative as long as $P \wedge B$ holds. Hence, the conclusion must be that at that moment $\neg B$ holds, and the loop stops. Termination detection has deliberately not been implemented in Aladin, since the user can easily ‘encode’ termination detection using the **let** construct. An example of this technique is given in section 6.4.

6.2 The Implementation of `proofImplication`

As the discussion in section 6.1 shows, the function `proofImplication` is a key ingredient of Aladin’s proving process. The function takes two Boolean expressions, and returns a Boolean value. If p and q are two predicates, then `proofImplication p q` returns `True` if Aladin is able to prove $KB \models p \Rightarrow q$, otherwise it returns `False`. Note that the function has a (small) side-effect, since it outputs on the standard

output which proof it is performing, and what the outcome is. This output allows the user to trace the state of a proof while it is running.

The function `proofImplication` makes use of a local helper function `proofImpl` that accepts an extra argument, being a string, that is used for indenting proofs on the standard output. This indentation is introduced because some proofs are split into sub-proofs. For example, an implication of the type $p \Rightarrow q_0 \wedge q_1$ is actually proved by a proof of $p \Rightarrow q_0$ followed by a proof of $p \Rightarrow q_1$. The reason for this split is that the sub-proofs are typically shorter than a proof of the entire implication. In practice this means that the number of clauses that is produced using sub-proofs is much smaller than the number of clauses that would have been produced if the complete implication is proven at once. The actual proof is performed by the function `proof'`, which also performs the splitting in sub-proofs (lines 9-10). Proofs that cannot be split further are in the end passed to the function `resolutionProof` (in line 11) together with the knowledge base in CNF format.

```

1 proofImplication :: BoolExpr -> BoolExpr -> Bool
2 proofImplication p q = proofImpl "##" p q
3   where
4     proofImpl indent p q
5       | proof indent p q = True `echo` (indent ++ " Success")
6       | otherwise       = False `echo` (indent ++ " Failed")
7     proof indent p q      = proof' indent p q `echo`
8       (indent++" Proving: [" ++ show p ++ "]" -> [" ++ show q ++ "])
9     proof' indent p (BinOp And q r) =
10       proofImpl ("##"++indent) p q && proofImpl ("##"++indent) p r
11     proof' indent p q = resolutionProof cnfKB (BinOp Implies p q)

```

6.3 Computing Weakest Preconditions of Assignments

Another key element of Aladin's proving process is the computation of weakest preconditions of assignment statements. As discussed in section 1.3.4, the weakest precondition of a series of assignments S and a postcondition Q is a predicate that describes a set of program states such that if execution of S is initiated from any one of these states, then S ends up in a state in which Q holds. Computing the weakest precondition, given S and Q , is a purely syntactical process as described in section 1.3.4.

We implement the series of assignments S as a list of the type `[(String, NumExpr)]`, where each tuple of this list consists of a `String` representing the name of a variable on the left hand side of an assignment, and a `NumExpr` which represents the right hand side of the assignment. The weakest precondition of a series of assignments with a given postcondition is computed recursively via the function `wps`. It computes the weakest precondition using the rule $wp(x := E; rest, Q) = wp(x := E, wp(rest, Q))$.

```

1 wps :: [(String, NumExpr)] -> BoolExpr -> BoolExpr
2 wps [] post                = post
3 wps ((lhs,rhs):rest) post = wp lhs rhs (wps rest post)

```

The function `wps` relies on the function `wp`, which computes the weakest precondition of a single assignment given a postcondition q .

```

1 wp :: String -> NumExpr -> BoolExpr -> BoolExpr
2 wp name rhs (BinOp op bel be2) =
3   BinOp op (wp name rhs bel) (wp name rhs be2)
4 wp name rhs (Not be)           = Not (wp name rhs be)
5 wp name rhs (Atom ba)          = Atom (replaceVarBA name rhs ba)

```

The function `wp` accepts three arguments. The first is the name of the variable on the left hand side of the assignment. The second argument is the expression `rhs` on the right hand of the assignment, and the third argument is the postcondition `q` of the assignment. It returns a Boolean expression which is the

weakest precondition. It is computed by replacing each occurrence of name in q by rhs . The function wp itself is basically only traversing the expression rhs , while the helper functions `replaceVarBA` and `replaceVariable` perform the actual replacement.

```

1  replaceVarBA :: String -> NumExpr -> BoolAtom -> BoolAtom
2  replaceVarBA name rhs (Compare op p q) =
3      Compare op (replaceVariable name rhs p) (replaceVariable name rhs q)
4  replaceVarBA _ _ ba                      = ba
5
6  replaceVariable :: String -> NumExpr -> NumExpr -> NumExpr
7  replaceVariable name rhs (Var x)          =
8      if x == name then rhs else (Var x)
9  replaceVariable name rhs (Binary op p q) =
10     Binary op (replaceVariable name rhs p) (replaceVariable name rhs q)
11  replaceVariable name rhs (UnaryMinus p)  =
12     UnaryMinus (replaceVariable name rhs p)
13  replaceVariable _ _ expr                  = expr

```

6.4 Some Examples of Verification Runs

In this section we show some example runs performed by Aladin. We start with the running example that consists solely of assignments statements.

```

const a;
var x, y;
  {x + y = Z}
  x := x + a;
  y := y - a;
  {x + y = Z}

```

We made a file with the file name `runningExample.p` containing the above code, and ran Aladin on it (as if it were a compiler). The following log of the running session is the result.

```

./Aladin runningExample.p
Passed 10000 tests.
## Proving: [(x + y)=Z] -> [((x + a) + (y - a))=Z]
## Success
Annotation is correct.

```

We see that Aladin needs to only prove that $x + y = Z \Rightarrow (x + a) + (y - a) = Z$, which is indeed correct since this is the direct translation of $x + y = Z \Rightarrow wp(x := x + a, wp(y := y - a, x + y = Z))$.

The next example program consists of an **if-then-else** construct. It computes the minimum of two variables x and y , and stores it in the variable x .

```

var x, y;
  {x = X ∧ y = Y}
if x < y then skip; {x = X ∨ x = Y} ∧ x < y}
else x := y; {x = X ∨ x = Y} ∧ x ≤ y}
end; {x = X ∨ x = Y} ∧ x ≤ y}

```

The output that is produced when we feed this program to Aladin is shown in the following log.

```

./Aladin min.p
Passed 10000 tests.
## Proving: [x=X and y=Y and x<y] -> [[x=X or x=Y] and x<y]
#### Proving: [x=X and y=Y and x<y] -> [x=X or x=Y]
#### Success
#### Proving: [x=X and y=Y and x<y] -> [x<y]
#### Success
## Success
## Proving: [x=X and y=Y and not x<y] -> [[y=X or y=Y] and y<=y]
#### Proving: [x=X and y=Y and not x<y] -> [y=X or y=Y]
#### Success
#### Proving: [x=X and y=Y and not x<y] -> [y<=y]
#### Success
## Success
## Proving: [[x=X or x=Y] and x<y] -> [[x=X or x=Y] and x<=y]
#### Proving: [[x=X or x=Y] and x<y] -> [x=X or x=Y]
#### Success
#### Proving: [[x=X or x=Y] and x<y] -> [x<=y]
#### Success
## Success
## Proving: [[x=X or x=Y] and x<=y] -> [[x=X or x=Y] and x<=y]
#### Proving: [[x=X or x=Y] and x<=y] -> [x=X or x=Y]
#### Success
#### Proving: [[x=X or x=Y] and x<=y] -> [x<=y]
#### Success
## Success
Annotation is correct.

```

We clearly see that the proof is split in four sub-proofs. In each of these sub-proofs, on their turn, proofs of the type $\alpha \Rightarrow \beta \wedge \gamma$ are split again into two sub-proofs $\alpha \Rightarrow \beta$ and $\alpha \Rightarrow \gamma$. The latter splits are done by the function `proofImplication`. The sub-proofs are indented using the symbol `##` for readability reasons. If at some point Aladin is not able to complete a sub-proof then it terminates, and the user can easily see which sub-proofs succeeded and which sub-proof failed.

The last example that we show is a simple implementation of the integer multiplication $a * b$ using solely addition and subtraction. Of course, we simply have the multiplication operator at our disposal in Aladin's programming language (so, there is little need to implement it), but this example is a standard (early) exercise in proving the correctness of loops in Bachelor courses that focus on Program Correctness.

```

const n;
var a, b, c;
  {P : a * b = n ∧ b ≥ 0}
c := 0;
  {c + a * b = n ∧ b ≥ 0}
while b ≠ 0 do
  {let V = b}
  {c + a * b = n ∧ b ≥ 0 ∧ b ≠ 0 ∧ b = V ∧ V ≥ 0}
  c := c + a;
  b := b - 1;
  {c + a * b = n ∧ b ≥ 0 ∧ b < V}
end;
  {Q : c = n}

```

Clearly, the invariant of this loop is $c + a * b = n \wedge 0 \leq b$. A termination proof is included using the `let`

construct, where the variant function $vf = b$ is introduced. The requirement that the function is positive, is encoded in the conjuncts $b = V \wedge V \geq 0$ in the line directly following the **let** statement. The decrement of the variant function is encoded by the conjunct $b < V$ in the last line of the body of the loop.

If we feed the program to Aladin, then we get the following output:

```
Passed 10000 tests.
## Proving: [(a*b)=n and 0<=b] -> [(0 + (a*b))=n and 0<=b]
#### Proving: [(a*b)=n and 0<=b] -> [(0 + (a*b))=n]
#### Success
#### Proving: [(a*b)=n and 0<=b] -> [0<=b]
#### Success
## Success
## Proving: [(c + (a*b))=n and 0<=b and b/=0 and V=b] ->
[(c + (a*b))=n and 0<=b and b/=0 and b=V and 0<=V]
#### Proving: [(c + (a*b))=n and 0<=b and b/=0 and V=b] ->
[(c + (a*b))=n and 0<=b and b/=0 and b=V]
##### Proving: [(c + (a*b))=n and 0<=b and b/=0 and V=b] ->
[(c + (a*b))=n and 0<=b and b/=0]
##### Proving: [(c + (a*b))=n and 0<=b and b/=0 and V=b] ->
[(c + (a*b))=n and 0<=b]
##### Proving: [(c + (a*b))=n and 0<=b and b/=0 and V=b] ->
[(c + (a*b))=n]
##### Success
##### Proving: [(c + (a*b))=n and 0<=b and b/=0 and V=b] -> [0<=b]
##### Success
##### Success
##### Proving: [(c + (a*b))=n and 0<=b and b/=0 and V=b] -> [b/=0]
##### Success
##### Success
##### Proving: [(c + (a*b))=n and 0<=b and b/=0 and V=b] -> [b=V]
##### Success
#### Success
#### Proving: [(c + (a*b))=n and 0<=b and b/=0 and V=b] -> [0<=V]
#### Success
## Success
## Proving: [(c + (a*b))=n and 0<=b and b/=0 and b=V and 0<=V] ->
[((c + a) + (a*(b - 1)))=n and 0<=(b - 1) and (b - 1)<V]
#### Proving: [(c + (a*b))=n and 0<=b and b/=0 and b=V and 0<=V] ->
[((c + a) + (a*(b - 1)))=n and 0<=(b - 1)]
##### Proving: [(c + (a*b))=n and 0<=b and b/=0 and b=V and 0<=V] ->
[((c + a) + (a*(b - 1)))=n]
##### Success
##### Proving: [(c + (a*b))=n and 0<=b and b/=0 and b=V and 0<=V] ->
[0<=(b - 1)]
##### Success
#### Success
#### Proving: [(c + (a*b))=n and 0<=b and b/=0 and b=V and 0<=V] ->
[(b - 1)<V]
#### Success
## Success
## Proving: [(c + (a*b))=n and 0<=b and b<V] -> [(c + (a*b))=n and 0<=b]
#### Proving: [(c + (a*b))=n and 0<=b and b<V] -> [(c + (a*b))=n]
#### Success
#### Proving: [(c + (a*b))=n and 0<=b and b<V] -> [0<=b]
#### Success
## Success
## Proving: [(c + (a*b))=n and 0<=b and not b/=0] -> [c=n]
## Success
Annotation is correct.
```

Chapter 7

Conclusion and Future Work

In this last chapter, we will discuss the conclusions of this project. Moreover we will discuss some limitations of Aladin and some future work to improve Aladin.

The most important conclusion is that Aladin works, and that it is possible to verify annotations using a proving strategy that is based on resolution, unification, substitutions, and arithmetic. As shown in the examples at the end of the chapters 3 and 6, Aladin is able to find counterexamples for incorrect annotations, and is able to verify correct annotations. When no errors are found in the annotation, Aladin outputs:

```
Annotation is correct.
```

If Aladin finds a counterexample, it outputs:

```
Assertion failed on line X: ...  
Initial state: ...  
Current state: ...  
Verification failed.
```

Although we successfully reached the goal of this project, Aladin has quite some limitations. Currently, Aladin supports only the integer data type. Most examples of annotations that are presented in the bachelor course ‘*Program Correctness*’ at the university of Groningen involve the integer data type. Indeed, Aladin gets quite far in verifying the correctness of annotations from that course. However, if we want to verify annotations that involve other data types (like Booleans or floating point numbers), then Aladin is not able to process them. Especially, the addition for the support of arrays would greatly improve the applicability of Aladin. Aladin also lacks support for functions, both in its programming language and in the language for expressing Boolean expressions (i.e. assertions). Introducing functions would greatly enhance Aladin’s applicability to annotations of real life programs. Adding these features to Aladin however would result in a large project, which is well beyond the size of a Bsc project (in fact, it could easily qualify as a Phd project).

As mentioned in Chapter 1, the resolution algorithm is in fact an implementation of a *breath first search*. A great advantage of this proving strategy is that it will produce proofs with a minimal length (i.e. number of proving steps). However, a huge disadvantage of this strategy is that in every layer of the proving process all possible inferences are generated, including ones that do not constitute to the proof. The combinatorial explosion of generated clauses can be significant, and we have seen several examples where memory consumption exceeds the size of available memory, and/or execution times are not realistic.

Therefore, it might be worth considering an alternative proving strategy that is based on a technique called *backward chaining*. In fact, that is the technique that is at the heart of the programming language Prolog (see [9], [17]). The main idea of this proving strategy is to perform directed searches for a proof. For example, if we want to proof the correctness of some predicate q , the strategy is as follows.

1. Check if q is already in the knowledge base . If it is, return `True`.
2. Find all implications I , whose conclusion matches q .

3. Recursively establish the premises of all i in I via backward chaining.

The great advantage of this proving strategy is that it avoids inferring unrelated facts. For example, assume that we are dealing with the following knowledge base:

$$KB = \{\alpha * (\beta + \gamma) = \delta \rightarrow \alpha * \beta + \alpha * \gamma = \delta, \alpha = \beta \rightarrow \alpha - 1 < \beta, x * (y + z) = w\}$$

We want to prove that $x * y + x * z = w$. A proof that uses resolution will generate, amongst others clauses, the clause $x * (y + z) - 1 < w$. However correct, this clause is useless and does not play any role in the proof. If however, we would use backward chaining, we see that only the first rule is an implication whose conclusion matches the goal $x * y + x * z = w$. Therefore, In the next step, we recursively try to find a proof for the premise $x * (y + z) = w$. This proof is trivial, since it is present in the knowledge base. Clearly, this directed proving strategy is more effective, but implementing it correctly is far from trivial. For example, it requires careful detection of cycles in the proving process. A disadvantage of using backward chaining is that the knowledge base is required to consist solely of facts (i.e. atoms) and implications of the form $\alpha_0 \wedge \alpha_1 \wedge \dots \wedge \alpha_n \rightarrow \beta$, where all literals must be positive (i.e no negations). Implications like these are called *Horn clauses* and are the basis for the Prolog language. Therefore, it is likely that imposing the requirement that the knowledge base contains only Horn clauses is likely not to impact Aladin's usability very much.

As described in Chapter 3, in an attempt to find counterexamples, we solve a constraint satisfaction problem (CSP) in a rather naive way. We described also that we did not spend much time to implement an efficient CSP solver because this is not the main topic of this thesis. However, many more sophisticated CSP solvers exist using smart heuristics techniques that drastically increase the performance of these solvers. In a future release of Aladin, one of these sophisticated solvers could be incorporated.

It always seems impossible until it's done
(Nelson Mandela)

Bibliography

- [1] The coq proof assistant. <https://coq.inria.fr>.
- [2] The hol proof assistant. <https://hol-theorem-prover.org>.
- [3] The isabelle proof assistant. <https://isabelle.in.tum.de>.
- [4] The pvs proof assistant. <https://pvs.csl.sri.com>.
- [5] The z3 proof assistant. <https://github.com/z3prover/z3>.
- [6] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers principles, techniques, and tools*. Addison-Wesley, Reading, MA, 1986.
- [7] J. Bakker. *A generic solver for Constraint Satisfaction Problems*. RuG FSE thesis database, <http://fse.studenttheses.ub.rug.nl/13059/1/genericsolvercsp.pdf>, 2015.
- [8] Robert S Boyer, Matt Kaufmann, and J Strother Moore. The boyer-moore theorem prover and its interactive enhancement. *Computers & Mathematics with Applications*, 29(2):27–62, 1995.
- [9] Alain Colmerauer and Philippe Roussel. The birth of prolog. *SIGPLAN Not.*, 28(3):37–52, March 1993.
- [10] Edsger Wybe Dijkstra. *A Discipline of Programming*. Prentice Hall PTR, USA, 1st edition, 1997.
- [11] Robert W Floyd. Assigning meanings to programs. In *Mathematical Aspects of Computer Science*, pages 19–32. XIX American Mathematical Society, 1967.
- [12] Herman Geuvers. Proof assistants: History, ideas and future. *Sadhana*, 34(1):3–25, 2009.
- [13] Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.
- [14] Donald E. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming*. Addison-Wesley, second edition, 1981.
- [15] Simon Marlow et al. Haskell 2010 language report. Available online [http://www.haskell.org/\(May 2011\)](http://www.haskell.org/(May 2011)), 2010.
- [16] Alberto Martelli and Ugo Montanari. An efficient unification algorithm. *ACM Trans. Program. Lang. Syst.*, 4(2):258–282, April 1982.
- [17] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall Press, USA, 3rd edition, 2009.
- [18] Mark E. Stickel. Resolution theorem proving. *Annual Review of Computer Science*, 3(1):285–316, 1988.