View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by DSpace at Tartu University Library

UNIVERSITY OF TARTU
FACULTY OF MATHEMATICS AND COMPUTER SCIENCE
Institute of Computer Science

Tiina Turban

Type Inference for a Cryptographic Protocol Prover Tool

Bachelor's Thesis (6 ECTS)

Supervisor: Liina Kamm, MSc
Supervisor: Sven Laur, PhD

Author: .... "....." 2012

Supervisor: .... "....." 2012
Supervisor: .... "....." 2012

Allowed to defence
Professor: .... "....." 2012

Tartu 2012

Contents

1 **Introduction** 3

2 **Proving Cryptographic Protocols** 5
2.1 Models in Cryptography 5
2.2 Proving Methods 6
2.2.1 Game-Based Proofs 8
2.3 Proving Tools 9

3 **The ProveIt Language and Types** 10
3.1 ProveIt as a System 10
3.2 The ProveIt Language 12
3.3 Types in the ProveIt language 14

4 **Formal Type System for ProveIt** 16
4.1 General Type Rule 16
4.2 Type Rules for Statements 17
4.3 Type Rules for Expressions 17
4.3.1 Arithmetic Operations 19

5 **Typing Algorithm for ProveIt** 21
5.1 Implementation Details 21
5.2 Type Info Propagation 22
5.3 Examples of Possible Type Errors 25
5.4 Algorithm Description 26
5.4.1 Parsing Statements 27
5.4.2 Parsing Expressions 28
5.4.3 Updating the Maximum Type 30
5.4.4 Updating the Minimum Type 31
5.5 Implementation 34

6 **Conclusion** 35

7 **Tüübituletus krüptograafiliste protokollide tõestaja jaoks** 36

1 Introduction

Cryptography is a science of information security. Information security systems are frequently built using well-established, low-level cryptographic algorithms (e.g. symmetric encryption, signatures, MACs, hash functions). The communication that uses these algorithms can be written down as a cryptographic protocol. More precisely, cryptographic protocols are ways for machines to interact and communicate using some mathematical techniques, with a goal of being secure against malicious adversaries. Depending on what security level one wants, it can mean, for example, that the adversary cannot intercept, manipulate, eavesdrop nor otherwise disrupt the communication.

One possible way to write down a cryptographic protocol is to describe it with programming code. This kind of representation is called a game. Via changing the game step-by-step we can sometimes reach another game which we know to be secure. Such a sequence of games (also known as a game-based proof) is one possible way to give the security proof of a protocol.

Game-based proving is a way to analyse security of a cryptographic protocol [BR04, Sho04], because it is more intuitive even to people not very familiar with cryptographic proving. Although there exist automatic provers, such as CertiCrypt [BGZ09] and ProVerif [Bla], paper and pencil are still used to write down the games. The downside of doing the proofs by hand appears when a change is made in an earlier game and one has two choices: rewriting the whole game sequence or keeping track of the changes to the original game. Both options are bad because of taking a lot of time and being error-prone. In addition, as proving cryptographic protocols is not the easiest thing to do, people often end up with tons of papers everywhere including some dead ends and lots of errors. Our mission is to change that and give the prover the possibility to concentrate on figuring out the next step instead of rewriting the game after each step.

In this thesis, I work with and expand a game-based cryptographic protocol prover called ProveIt, which is being developed by the cryptography research group in Tartu. It is not an automatic prover, but an auxiliary tool for cryptographers. ProveIt allows researchers to make changes to the code-based probabilistic game and does the rewriting automatically in order to guarantee correctness and save the researcher's time. The idea behind ProveIt is to simplify the process of proving a cryptographic protocol in a game-based way and also to keep it understandable.

Although cryptography has an important role in this thesis, I mainly focus on type theory. Introducing types into ProveIt improves correctness and user comfort. Via type checking we can make sure that the user did not add numbers to candies. Having information about variable types comes in handy when something goes wrong, allowing us to provide helpful hints to direct the user back to the right track. We do not require cryptographers to define all variable types before using,

but we still want to do type checking. In order to do that, we need to infer the types automatically based on the uses of the variables, i.e., do type inference.

In this thesis I will firstly explain how to do type inference for the ProveIt language and, secondly, implement it. More precisely, in Chapter 2 I talk about proving cryptographic protocols in general. I go on to describe the prover tool – ProveIt– in Chapter 3. In Chapter 4, I will write out type rules to the ProveIt language. Typing function declarations and logical expressions are not in the scope of this thesis. The algorithms to perform type inference will be given in the last chapter, where I will also summarise the implementation.

2 Proving Cryptographic Protocols

2.1 Models in Cryptography

There are two different ways to analyse the security of a cryptographic protocol: in the formal and the computational model. The formal model, also known as the Dolev-Yao model, makes it easier to analyse the security properties of the protocol, the computational model, on the other hand, is much closer to real life [AR00].

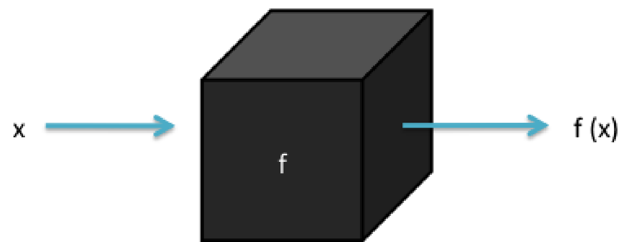


Figure 1: Applying a function to an argument in the formal model

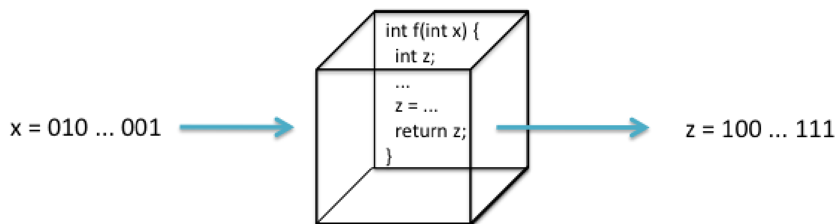


Figure 2: Applying a function to an argument in the computational model

The difference between the formal and the computational model is about what information the adversary knows. In order to better understand the difference we can look at Figures 1 and 2. Protocols given in the formal model can be viewed as sequences of black boxes. This means that neither we nor the adversary know what goes on inside them, everybody only has access to the function call, e.g. we call the encryption function $\mathbf{Enc}(x)$ but nobody knows exactly how x is encrypted. In addition, the input x and the output $f(x)$ can be considered as envelopes, that no-one can decompose into smaller pieces, one can only apply known functions and compare terms. As a consequence, all possible attacks are either clever sequences

of function applications or a sort of brute forcing over all potential inputs or secret keys. Protocols in the computational model, on the other hand, can be thought to consist of white boxes from bit-string to bit-string. This means that the adversary knows what happens during encryption and decryption, and the input and output bit-strings can also be manipulated.

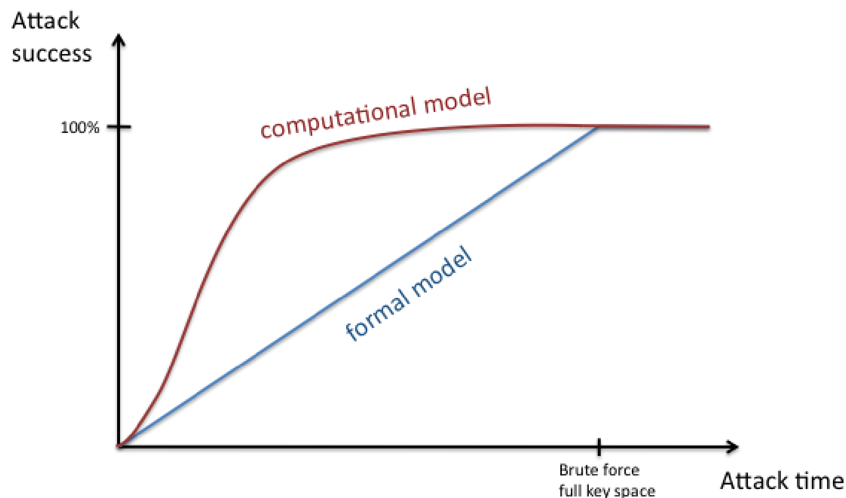


Figure 3: Attack time and success relation in different models

A function in the formal model is like a black box, and therefore, usually the only attack against a function would be to brute force possible keys. Hence, the probability of the attack success grows linearly until the the key space is exhausted in which case the attack will definitely be successful. In reality, there are other attacks based on the actual implementation, such as factoring or finding discrete logarithms. This is the reason why the curve for breaking a function in the computational model is different as the protocol can in some cases be broken faster than by brute force. Figure 3 illustrates this comparison.

2.2 Proving Methods

There are many different ways to write down cryptographic protocols, for example process algebra, finite automata and probabilistic program language. But in the end, the result of the security proof – either success or failure – is important, not how it was discovered, so the representation of a protocol is just a question of preference.

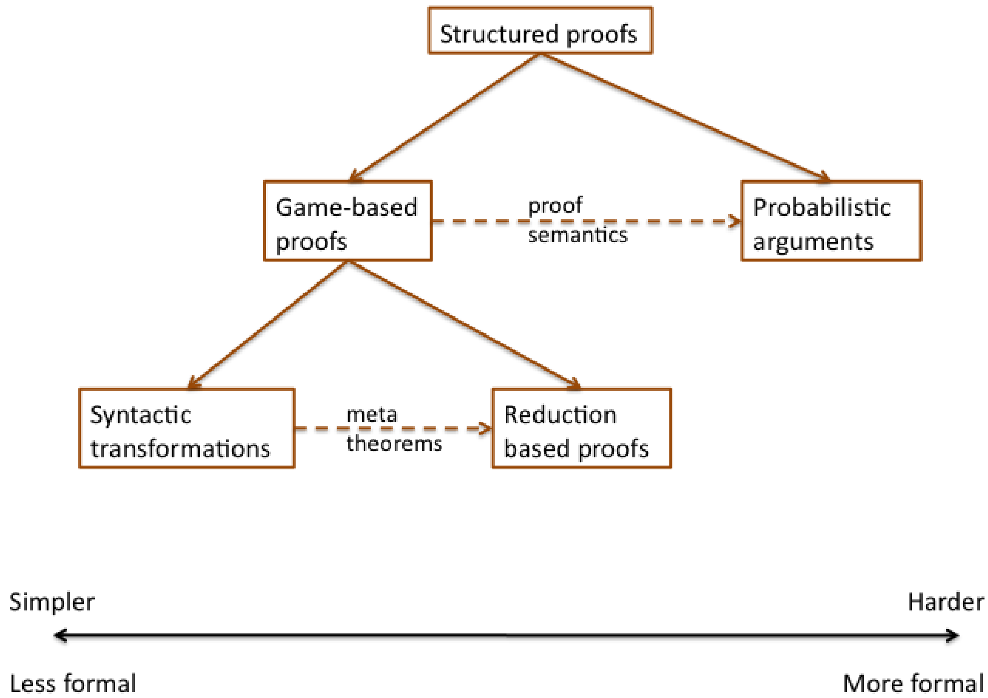


Figure 4: Handling complex proofs

Complex proofs are usually split into more manageable smaller parts (Figure 4). These structured proofs can be combined using arguments about success probabilities. This can sometimes be difficult, so the underlying structure of the proof is usually captured by a series of games, known as game-based proving, which is easier but also less formal. If one wants, then he or she can use their preferred proof semantics on the game-based proof to write it down with probabilistic arguments. A game-based proof consists of many syntactic transformations, e.g. switching 2 rows or deleting an unnecessary statement in the code of the game. Each of these corresponds to a reduction, which shows how close the output distributions of resulting games are (meta theorem **??**). Thus, with applying a reduction schema we can say how the probability changes when doing a transformation.

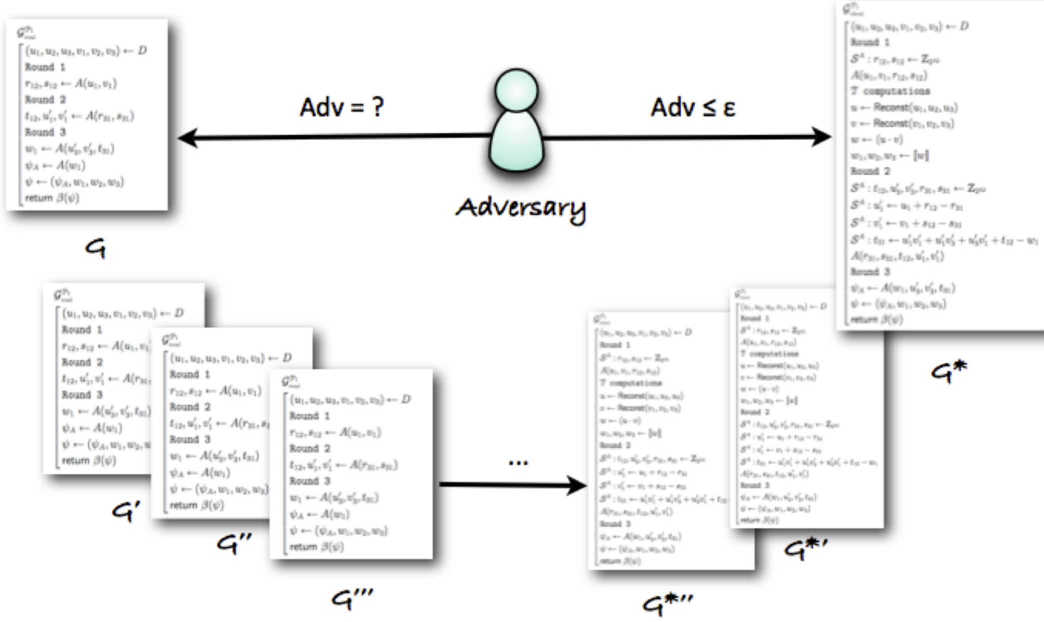


Figure 5: How game-based proving works [Kam12]

2.2.1 Game-Based Proofs

To prove security in a game-based way one constructs a sequence of games: G, G' ... G^* (see Figure 5) where the first game models a real world threat. Firstly the initial game is written down using some probabilistic programming language, e.g. pWhile, so that the game measures the success of an attack. Usually, the game ends with the output 1 if the attacker is successful. Making a syntactic transformation on the original game G we get game G'. Doing another small change on G' we reach G'' and so on. Each step can change the success of an attack by some degree. Our goal is to reach the game G^*, whose attack success we already know. We can also do transformations on G^* and meet somewhere in the middle of the chain of games. As a result, we create a game tree, where we know the security difference of adjacent nodes. Knowing how the transitions between the games change the security properties, and taking into account the security parameters of G^*, it is possible for us to calculate the security properties of the original game G.

2.3 Proving Tools

There is always the possibility to prove everything by hand, but as time passes, we use technology more and more to help us achieve our goals faster. There are quite a few automatic cryptographic provers out there, for example CertiCrypt [BGZ09] and ProVerif [Bla]. The problem is that if a process is automatic, it is often also hard to understand, at least for beginners or for people, who have not seen it before, as the learning curve is very steep. Moreover sometimes protocols are not automatically provable. Firstly, the number of potential proof steps can be not manageable as it might grow exponentially. Secondly, some proofs have a large branching factor unless prerequisite knowledge or heuristics are used. Thirdly, there are hard limits stemming from the Rice theorem, which states that automatic tools are bound to fail proving non-trivial properties about the code in general. Testing a program means checking if the program satisfies certain conditions which mimic functional requirements. Rice theorem states that even for the simplest properties it is undecidable whether a program satisfies these requirements, and therefore automatic testing of preconditions of certain game transformations is not possible. One can prove that the program is not correct or a condition does not hold by providing a counterexample, but in order to prove the correctness we would have to run the program on all the possible inputs of which there is an infinite amount.

3 The ProveIt Language and Types

3.1 ProveIt as a System

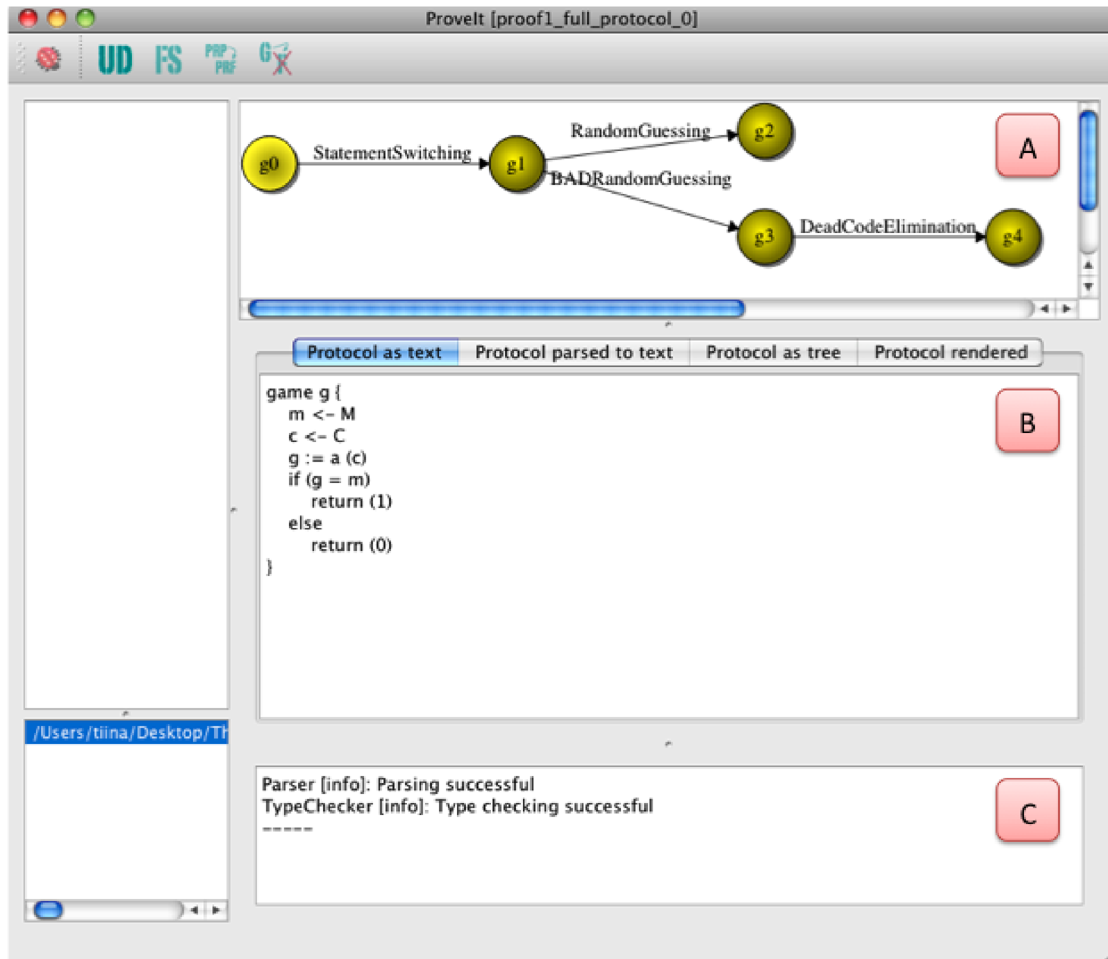


Figure 6: Screenshot of ProveIt

The tool we are developing is called ProveIt. A screenshot of ProveIt can be seen on Figure 6. It is a tool to help cryptographers to prove protocols in a game-based way in the computational model. The user first enters the pseudocode of the initial game in panel B. He or she can then apply transformations on the game. There are several of predefined proof steps, e.g. pseudorandom permutation/pseudorandom function switching, function rename and dead code elimination. For these steps ProveIt calculates the changes in security and guarantees the correctness. There is also a free step to give more freedom to the user. After the proof

step is chosen, the game is rewritten automatically, which reduces the number of rewriting errors and makes the proving process faster. ProveIt also keeps track of all the previous games showing the progress on a graph seen in panel A. One can make changes to any of the games shown on the graph, accessing them via clicking on the node. In addition, ProveIt offers help to researchers, students, or teaching staff by showing which proof steps could be used in the current state and checking for correct rule usage.

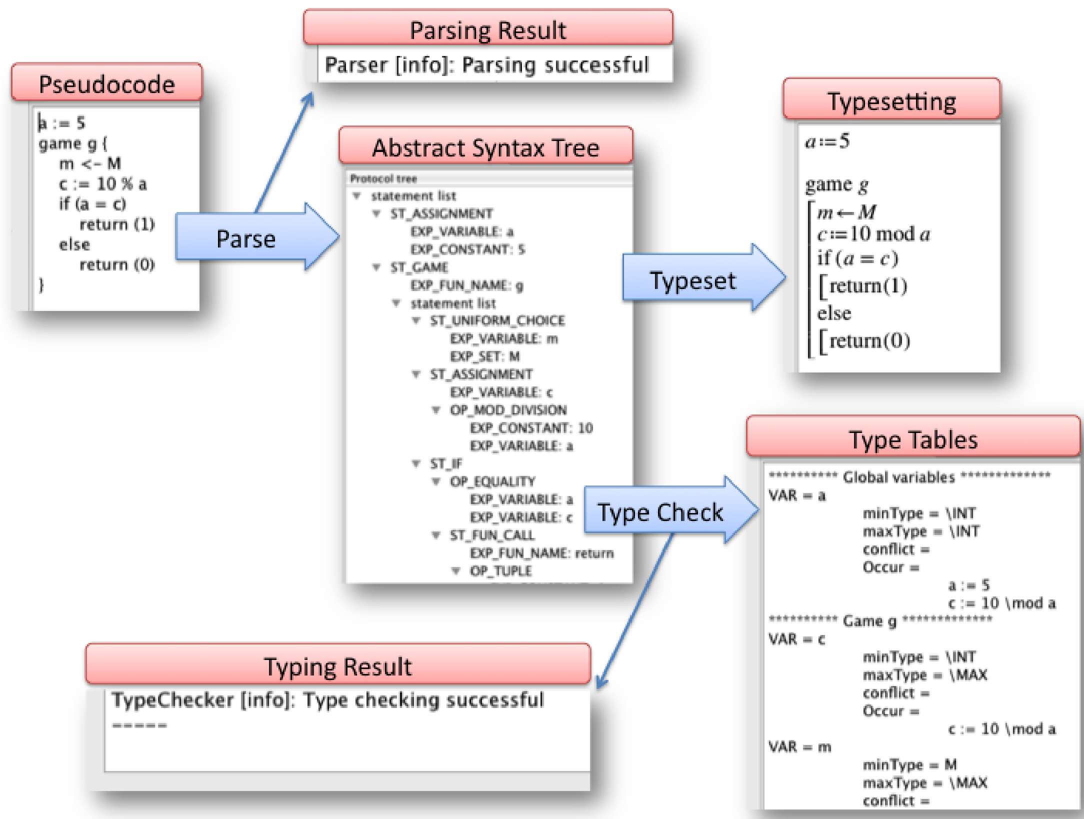


Figure 7: What happens to the pseudocode of the game

Let us now look more closely at what happens after the initial protocol is entered and after every proof step (see Figure 7). Firstly, ProveIt parses the game the pseudocode to an abstract syntax tree and outputs erros if any (see panel C on Figure 6). Next steps are typesetting and type checking, which are both done based on the parsed abstract syntax tree. If the parsing, the typesetting, and the type checking all succeed, then the proving process may continue.

3.2 The ProveIt Language

Syntax of the ProveIt language is briefly specified in [Kam11]. Here we briefly review the most basic concepts. The ProveIt language is a probabilistic program language. It is used to write down games in the ProveIt prover tool. The ProveIt language consists of statements. We have statements for assignment, uniform choice, functions signature, the main game definition, function definition and statements that modify the control flow. Functions must have a signature before the definition. The function signature specifies the name, parameter types and the return value type of the function. The function definition holds the function body, which must contain a return statement at the end.

Both assignment and uniform choice give a value to the variable on the left. Uniform choice takes a random value from a given set. Assignment, on the other hand, evaluates the arithmetic expression given on the right side. The statements that affect the execution order of the program are if-condition with an optional else-branch, for-cycle and while-loop. They use logical expressions to determine the control flow. Table 1 describes statements in ProveIt.

Statement	Typeset Example
assignment	$a := 5 + b$
uniform choice	$m \leftarrow M$
if statement	`If` $(a = c)$ [`return` a
for statement	`For` $(i := 0; i < 9; i := i + 1)$ [$x := x - i$
while statement	`while` $(x = 0)$ [$x := x - 1$
function signature	$f : K \times M \rightarrow L$
function definition	`fun f(`k, i`)` [$b \leftarrow L$ `return` b
main game definition	`game g` [`return` 0

Table 1: Statements in the ProveIt language

Arithmetic expressions can be constructed from constants, variables, function calls and other arithmetic expressions combined together with arithmetic operators. All operators are given in the order of precedence (highest to lowest) grouped together with other same precedence operators, e.g. addition and subtraction have the same precedence, which is lower than multiplication but higher than equality (logical operator). Table 2 shows the arithmetic operations in ProveIt.

Operation	**Typeset Example**
opposite	$-a$
exponentiation	a^b
multiplication	$a \cdot b$
division	$a \ / \ b$
integer division	$\lfloor a \ / \ b \rfloor$
remainder	$a \ \% \ b$
addition	$a + b$
substraction	$a - b$

Table 2: Arithmetic operations in the ProveIt language

Logical expressions can be constructed from arithmetic expressions or other logical expressions combined together with logical operators. Logical operations in ProveIt are shown in Table 3.

Operation	**Typeset Example**
negation	$\neg a$
equality	$a = b$
inequality	$a \neq b$
less than	$a < b$
greater than	$a > b$
less than or equal to	$a \leq b$
greater than or equal to	$a \geq b$
logical and	$a \ \&\& \ b$
logical or	$a \ \|\| \ b$

Table 3: Logical operations in the ProveIt language

Additionally, we have set operations. They allow us to find the union, intersection or complement of sets and check the membership of a variable for a set. Table 4 describes set operations in ProveIt.

Statement	Typeset Example
union	$K \cup M$
intersection	$K \cap M$
complement	$K \setminus M$
in	$a \in K$
not in	$a \notin K$

Table 4: Set operations in the ProveIt language

3.3 Types in the ProveIt language

Proving can be done without introducing types, but with type inference we can further guarantee the correctness and provide help to the user. For example, it can be useful to know what type of parameters the function takes.

Similarly to programming code, each game also consists of different variables and functions. To be more precise, in each pseudocode we can have global variables and local variables in functions or in the game. To illustrate this, we can look at the following code example.

$$f : K \times \mathsf{Int} \to L$$

$$a \leftarrow L$$

$\texttt{fun f(}k, i\texttt{)}$

$$\left[\begin{array}{l} b \leftarrow L \\ \texttt{return } b \end{array}\right.$$

$\texttt{game g}$

$$\left[\begin{array}{l} b \leftarrow K \\ c := a + f(b, 6) \\ \texttt{return } c \end{array}\right.$$

In this example, we have a *game g*, which can be thought of like a main function in programming. The variable a is a global variable and therefore can be accessed from everywhere. In the main game the local variables are b and c, and in the function f we have b, k and i. The scope of c is limited by *game g*. Although b exists in both *game g* and *fun f*, it is not the same variable. It is not possible to create a local variable with the same name as a global one.

We have two important semantics rules:

1. Every variable has exactly 1 type - the same variable name cannot be first used as an integer and later as something else.
2. Every variable has to be initialised before using - the variable has to have a value before it can be used as an expression.

4 Formal Type System for ProveIt

First of all, we introduce two notations we are going to use in this section. Parser structure is written using the following syntax shown on Figure 8. This means that for `something` we have three possibilities, it is either `option A`, `option B` or `option C`.

```
something
   : option A
   | option B
   | option C;
```

Figure 8: ProveIt parser structure syntax

Type rules will be given in the standard notation:

$$\frac{premisses}{conclusion},$$

where *premisses* is a set of conditions that have to be satisfied in order to apply the rule and *conclusion* is a statement that is true if the *premisses* are satisfied. For example the rule:

$$\text{addition}\frac{\Gamma \vdash x : T \qquad \Gamma \vdash y : T}{\Gamma \vdash x + y : T}$$

means that if x and y are of type T then their sum is also of type T. Γ represents the environment, which in our case is the game written in the ProveIt language. For a simple overview of type inference see [Sch95].

4.1 General Type Rule

We now introduce one general type rule to make all the other rules and type inference process simpler. This rule applies for all the variables:

$$\text{var}\frac{\Gamma \vdash x : S \qquad \Gamma \vdash S \subseteq T}{\Gamma \vdash x : T},$$

which states that if x is of type S and S is a subtype of T, then x is also of type T. Due to this rule, all the other rules only state that a variable is at least of some type T, but do not fix a concrete type.

4.2 Type Rules for Statements

Each protocol consists of statements. Each statement could either be a condition ($if(...)\{...\}$), a loop ($while(...)\{...\}$), a uniform choice ($a \leftarrow A$), an assignment ($a := 5 + b$), a function declaration ($f : K \times M \to C$) or the beginning of a game or a function. This thesis includes only typing for arithmetic expressions, but not function declarations nor logical expressions. We consider this future work. It is important for us to understand the scope of functions, but we do not check the parameters nor the return value types.

First, let us look at the uniform choice statement, which takes a random value from a set and assigns it to a variable

$$VARIABLE \leftarrow SET \ .$$

We can write the following type rule for uniform choice

$$\text{uChoice}\frac{\Gamma \vdash x \leftarrow T}{\Gamma \vdash x : T} \ .$$

Second, we have assignments. The value of the expression is assigned to the variable

$$VARIABLE := expression \ .$$

To get the type of the variable we first need to find out the type of the expression

$$\text{assign}\frac{\Gamma \vdash x := exp \qquad \Gamma \vdash exp : T}{\Gamma \vdash x : T} \ .$$

Here *exp* can be any kind of expression as we can see from Section 4.3.

4.3 Type Rules for Expressions

Expressions exist on the right side of assignments (see Section 3.2). Figure 9 describes the parser structure for expressions in ProveIt. Firstly, an expression can be a constant, which by default is of type Int

$$\text{const}\frac{}{\Gamma \vdash NUMBER : \mathsf{Int}} \ .$$

Secondly, it can be a variable. Variables get their initial type during initialisation either with uniform choice or an assignment (see Section 4.2). Thirdly, an expression can be constructed from numbers and variables, using different mathematical functions. The possible arithmetic operations in ProveIt are shown in Figure 10. Finally, an expression can be a function call, but typing function calls is not in the scope of this thesis. The use of parentheses allows one to nest the arithmetic

```
expression
  : NUMBER
  | VARIABLE
  | '-' expression
  | expression '+' expression
  | expression '-' expression
  | expression '*' expression
  | expression '/' expression
  | expression '\div' expression
  | expression '%' expression
  | expression '^' expression
  | function_name '(' parameters ')'
  | '(' expression ')'
```

Figure 9: Grammar of arithmetic expressions

1. opposite ($-$) – as a unary operator
2. addition ($+$)
3. subtraction ($-$) – as a binary operator
4. multiplication ($*$)
5. division (/) - the resulting type is at least of Rat
6. integer division (\div) - only usable between Int types
7. remainder (%) - only usable between Int types
8. exponentiation (^) - only usable to the power of Int or Rat

Figure 10: Arithmetic operations in ProveIt revisited

operations any way one wants. Neither finding the opposite nor using parentheses changes the type of the expression

$$\text{opposite}\frac{\Gamma \vdash x : T}{\Gamma \vdash -x : T} \quad ,$$

$$\text{parentheses}\frac{\Gamma \vdash x : T}{\Gamma \vdash (x) : T} \quad .$$

4.3.1 Arithmetic Operations

We will now define type rules for arithmetic operations and with this complete our type system. Before we can do this, it is important to understand how different types are related. Consider the following relation:

$$\mathsf{Min} \subseteq T \subseteq \mathsf{Max} \ .$$

Here T is an arbitrary type, Min (usually denoted by $\perp$ in type theory) and Max (usually denoted by $\top$ in type theory) are types that do not really exist and denote our universal subtype and universal supertype. Integers are embedded in rational numbers and like any other type they are subtype of Max and supertype of Min

$$\mathsf{Min} \subseteq \mathsf{Int} \subseteq \mathsf{Rat} \subseteq \mathsf{Max} \ .$$

Based on standard algebraic structures and operations defined on them, our predefined type classes are described in Figure 11. Our typing is optimistic, that is, we give the construction a type if we believe it might exist. As inverse elements exist in Rings for some cases, we allow division at the user's discretion. If the user does not say anything about the type of set T, we assume that $T \in$ Field. Additionally, Int is a Ring and Rat is a Field.

1. Type *Field* with operations: $+, -, *, /$.
2. Type *Ring* with operations: $+, -, *, /$.
3. Type *Multiplicativegroup* with operations: $*, /$.
4. Type *Additivegroup* with operations: $+, -$.
5. Type *Set*

Figure 11: Predefined type classes in ProveIt

Now we can look at the addition and the subtraction operator. Let T be an additive group, a ring or a field. Then

$$\text{addG}\frac{\Gamma \vdash x : T \qquad \Gamma \vdash y : T}{\Gamma \vdash x \oplus y : T}$$

where $\oplus \in \{+, -\}$. If two variables of the same type are added together or one is subtracted from the other then the result is also of the same type. The rule together with the `var` rule states, that if addition or subtraction is performed between two variables then the result is of type T, which is their common supertype.

For multiplication and division let T be a multiplicative group, a ring or a field, then

$$\text{multiG}\frac{\Gamma \vdash x : T \quad \Gamma \vdash y : T}{\Gamma \vdash x \otimes y : T}$$

where $\otimes \in \{*, /\}$. This rule is similar to addition and subtraction stating the same about the type of the multiplication or division result.

Exponentiation can be done only to the power of an integer or a rational number. In order to do exponentiation with positive integers we have to have the multiplication operator defined, with negative integers we also need division. This might sometimes limit the user (the need to prove a variable being positive), so we consider making sure the exponentiation is legal to be at the user's discretion. Let T be a multiplicative group, a ring or a field

$$\text{expon}\frac{\Gamma \vdash x : T \quad \Gamma \vdash y : \mathsf{Rat}}{\Gamma \vdash x\hat{\ }y : T} \quad .$$

The type of the result for exponentiation is the same as the type of the base.

Special rules for integers. Integers are treated differently in our system and therefore some rules are different for them. We have a rule `multiG` for multiplication and division. The result of dividing two integers can be an integer in some cases, however, in all cases where one integer is divided by another, we can be sure that the result is at most a rational number. Therefore, we have a rule for division

$$\text{intDiv}\frac{\Gamma \vdash x : \mathsf{Int} \quad \Gamma \vdash y : \mathsf{Int}}{\Gamma \vdash x \ / \ y : \mathsf{Rat}} \quad ,$$

which must be applied if both operands are integers. Additionally we allow modulo operation and integer division

$$\text{modulo}\frac{\Gamma \vdash x : \mathsf{Int} \quad \Gamma \vdash y : \mathsf{Int}}{\Gamma \vdash x \ \% \ y : \mathsf{Int}} \quad ,$$

$$\text{intDivFloor}\frac{\Gamma \vdash x : \mathsf{Int} \quad \Gamma \vdash y : \mathsf{Int}}{\Gamma \vdash x \ \backslash\mathsf{div} \ y : \mathsf{Int}} \quad .$$

For exponentiation the situation is more complex, because extending the type to a Rat does not help us, as the user can try to evaluate 5 ^ (1 / 2), which is an irrational number and we do not have irrational numbers in ProveIt. Therefore, we do not make a special rule for exponentiation, but use the assumption that the users know what they are doing.

5 Typing Algorithm for ProveIt

In this chapter we provide algorithms for type inference in ProveIt. The algorithms and the implementation are done in an imperative programming language. Even though it is considered easier to implement type checking in a functional programming language we chose C++ because this is the language ProveIt is written in. Adding type inference to ProveIt requires us to start with finding out where all the variables exist in the code of the game.

5.1 Implementation Details

The result of the typing algorithm is information about all the variables with their types and typing errors if they exist. Since it is tedious for a prover to fix one type error at a time and therefore parse the same code multiple times, we are giving as much information about errors as we possibly can. This means that when we find the first type error we will continue checking and write out all the other errors as well, similarly to several programming language compilers.

In ProveIt we have both global and local variables as mentioned in Section 3.3. Therefore, in order to allow the same variable names in different functions, we have two choices to represent the information: making a big data structure and having a field describing the scope of the variable or having several smaller data structures. We use the second option because it gives the user a better overview of all the variables and their scopes. More precisely we have a table for global variables, for the main game and for each function separately.

```
a := 5
fun f()
  a ← A
  b := 6
  a := 7^b
  return b
game g
  b ← B
  return b
```

Global variables

variable	minType	maxType
a	A	Max

Variables in function f

variable	minType	maxType
b	Int	Rat

Variables in game g

variable	minType	maxType
b	B	Max

Figure 12: Example of pseudocode and corresponding type tables

We introduce the notation of a minimum and a maximum type for each variable,

such that, the type of a single variable can be chosen as long as it is a subtype of the maximum type and a supertype of the minimum type. But as the purpose of ProveIt is to make proving easier, we hide this fact from the user and in the end simply assign one concrete type for each variable (the minimum type). For illustration, see the pseudocode and corresponding type tables in Figure 12.

In this example,we can say that the variable a has to be at least of type A, but is not limited by this type – that is what the Max represents. Variable b from function f, on the other hand, is either a rational number or an integer, since it is used on the right side of exponentiation. Also, note that variable a appears only in the table of global variables, whereas the label b represents two different variables: one from function f and the other from the game g.

It is important to notice that after type inference and type checking are done we can safely choose minimum types for all the variables to represent their uniquely determined type, because variables get valid types via initialisation. If we, on the other hand, choose anything but the minimum type for a variable, then we will have to retype all the other variables as this decision can affect them. As an example for why retyping is needed, look at the following code.

$$a := 5$$
$$b := a$$

Here the algorithm will tell us that both a and b are of Int. If, instead of the minimum type, we choose a to be of type Rat (the maximum type of a is unlimited), then b cannot be of type Int. Another solution to this problem is to keep track of the dependencies between variables, but we do not need such complexity for ProveIt.

The reason why we cannot choose the maximum type is that Max does not exist as we might not have a supertype of everything. For minimum type we demand initialisation, therefore, if type checking succeeds we have a valid type for each variable. The purpose of the maximum type is to simplify type inference and assist in providing better error messages to the user. After applying all rules to the minimum and maximum type separately, we only have to check that the minimum type is a subtype of the maximum type.

5.2 Type Info Propagation

To better explain how minimum and maximum types are determined, we can look at Figures 13, 14 and 15. These figures represent how variable types change by showing the type propagation process for an assignment statement:

$$c := a + b \ .$$

We are looking at one statement in the middle of a game, where we have already parsed some statements and gathered information about variables a, b and c. In Figure 13 the state before parsing this assignment is shown. Under the variables (green boxes) the type ranges are shown. For example the initial minimum type is Int and maximum type c is T.

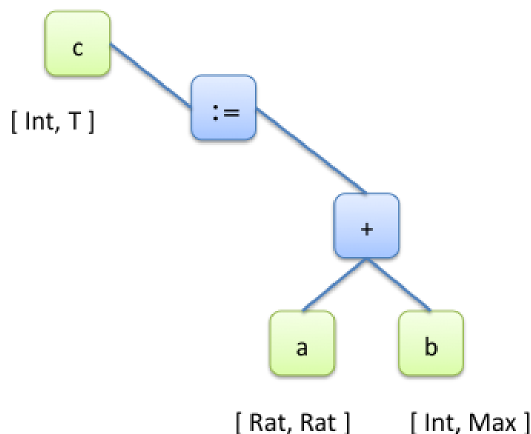


Figure 13: Type info propagation – initial state

Firstly, let us parse the statement from the bottom up. We see the operation + for which we can use the addition and subtraction rule `addG` (see Section 4.3.1). As the rule is applied to two variables of the same type the `var` rule (see Section 4.1) must be applied first. As we can apply the `var` rule on either of the variables an unlimited number of times, we can only determine the minimum type of the variables on the left side, i.e, going up only gives us information about the minimum types (Figure 14). The dots (...) represent that the type is not limited, which currently denotes that we do not care about the maximum type. The propagation of type info upwards includes the following steps. First we do an extra step to be able to use addition

$$\text{var}\frac{\Gamma \vdash b : \mathsf{Int} \qquad \Gamma \vdash \mathsf{Int} \subseteq \mathsf{Rat}}{\Gamma \vdash b : \mathsf{Rat}} \quad .$$

Secondly, we use the addition and subtraction rule for two rationals

$$\text{addG}\frac{\Gamma \vdash a : \mathsf{Rat} \qquad \Gamma \vdash b : \mathsf{Rat}}{\Gamma \vdash a + b : \mathsf{Rat}} \quad .$$

Finally, we use the assignment rule

$$\text{assign}\frac{\Gamma \vdash c := a + b \qquad \Gamma \vdash a + b : \mathsf{Rat}}{\Gamma \vdash c : \mathsf{Rat}} \quad .$$

When we put it all together, we get

$$\text{assign}\;\frac{\Gamma \vdash c := a + b \qquad \text{addG}\;\dfrac{\Gamma \vdash a : \mathsf{Rat} \qquad \text{var}\;\dfrac{\Gamma \vdash b : \mathsf{Int} \qquad \Gamma \vdash \mathsf{Int} \subseteq \mathsf{Rat}}{\Gamma \vdash b : \mathsf{Rat}}}{\Gamma \vdash a + b : \mathsf{Rat}}}{\Gamma \vdash c : \mathsf{Rat}}\ .$$

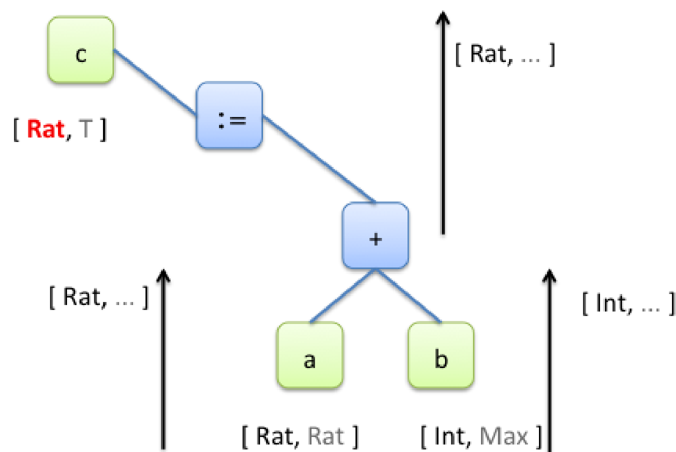


Figure 14: Type info propagation – upwards (minimum type)

From this we have learned that c must be at least of type Rat. As mentioned before, because of the rule `var`, it is impossible to set any limitations on the maximum type of c.

Secondly we parse the statements downwards (see Figure 15). We know that c has to be of type T, this knowledge allows us to use the type rules from conclusion to premisses for the maximum type. This helps us make sure it is not possible to assign something that is not of subtype of T to c. If c is of type T then $a + b$ has to be of a subtype of T. This in turn concludes that both a and b are of subtype T. This means that their maximum type has to be at most T. We do not update the maximum type of a as it is already stricter. Again, as with going upwards, we cannot say anything about the minimum type of a or b because of the `var` rule. Putting it all together, we get

$$\text{assign}\;\frac{\Gamma \vdash c := a + b \qquad \text{addG}\;\dfrac{\text{var}\;\dfrac{\Gamma \vdash a : \mathsf{Rat} \qquad \Gamma \vdash \mathsf{Rat} \subseteq T}{\Gamma \vdash a : T} \qquad \Gamma \vdash b : T}{\Gamma \vdash a + b : T}}{\Gamma \vdash c : T}\ .$$

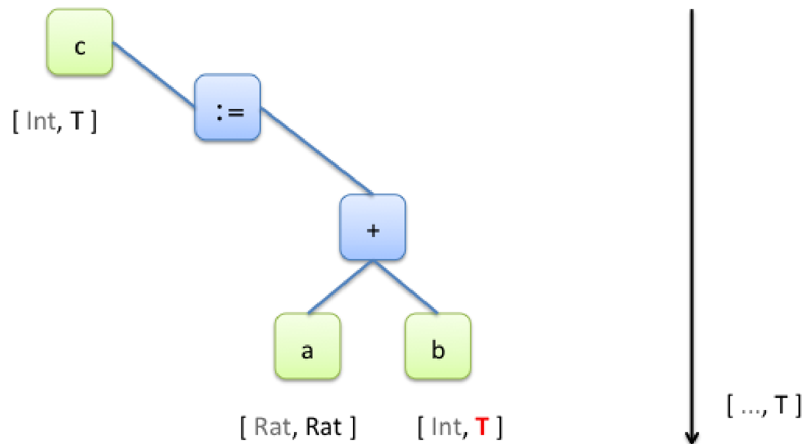


Figure 15: Type info propagation – downwards (maximum type)

5.3 Examples of Possible Type Errors

To inform the user of type errors, we will first look at some of the possible type errors that can occur.

Variable is used before initialisation. For instance, consider the following example

$$y \leftarrow T$$
$$a := x + y$$

where variable x is not initialised before use in the second line.

Variable is assigned something that is not from the subset of the variable's type. Let us assume that we have a variable a, that is, an integer, then the following assignment

$$a := 5 \ / \ 2$$

gives us an error because a rational value cannot be assigned to a variable with integer type.

An operation is done for types for which it does not exist or is not allowed. Consider the expression

$$(5 \ / \ 4) \ \% \ 3$$

where we try to use the modulus operator, but as division gives us a rational number and modulus is only defined for integers this produces an error.

5.4 Algorithm Description

In this subsection, we provide 6 algorithms for type checking and type inference in ProveIt:

1. `typeChecking` (Algorithm 1) is the main function, that is responsible for dealing with the scopes of the variables;
2. `parseStatement` (Algorithm 2) applies type rules for uniform choice and assignment;
3. `parseExpression` (Algorithm 3) helps parse the right side of assignments;
4. `updateMaxType` (Algorithm 4) updates the maximum type of the variable if it is stricter than the current one and subtype of the current maximum type;
5. `updateMinType` (Algorithm 6) updates the minimum type of the variable if it is stricter than the current one and supertype of the current minimum type;
6. `LCA` (Algorithm 5) finds the lowest common ancestor and helps with updating the minimum types.

First, we discuss the main function (see Algorithm 1), where the type checking and inference begin and the result of type checking is returned. One of the first things that can be noticed about type checking is that statements are parsed multiple times. The reason is illustrated by the following example:

$$a := 5$$
$$b := a$$
$$a := 5 \ / \ 4 \ .$$

In this case, after the first round of type checking we will learn that the type of a is at least Rat and the type of b is at least Int. Only after checking the types in the second round we discover that b is also at least of type Rat, as we can conclude from the assignment on the second line, knowledge about the type of variable a and the assignment type rule.

In the while-loop in `typeChecking`, statements are parsed and passed to the `parseStatement` function. As an argument we add the type table, to which the variables are added, based on the scope. In the main game, all the variables have local scopes similarly to functions, therefore we treat $\mathsf{type}_{\mathsf{game}}$ as we would any function from F (collection of all functions). Doing this gives us the possibility to more easily handle different scopes of variables. Now the scope is either $\mathsf{type}_{\mathsf{G}}$ in case of global statements or local otherwise.

Algorithm 1: typeChecking

Input: Abstract syntax tree

Result: Type checking result; In addition, type table $\mathsf{type_G}$ for global variables, $\mathsf{type_{game}}$ for the main game and $\mathsf{type_{fun}}$ for each $fun \in F$; Error messages with statements if they occur

```
1 while something changed in last cycle do
2     foreach global statement s do parseStatement(s, type_G);
3     foreach function fun do
4         foreach statement s in fun do parseStatement(s, type_fun);
5     foreach statement s in game do parseStatement(s, type_game);
```

As mentioned in the results of Algorithm 1, it outputs error messages and statements where they occur. For simplicity, we have left out most of the details concerning the implementation of error handling. In the implementation, we keep track of the type checking result and the active statement. With this we can achieve that the same error messages are output only once. The result of the type checking algorithm is failure if errors exist and success otherwise.

5.4.1 Parsing Statements

Now let us look closer at what happens to each statement. Algorithm 2 gets two arguments: a statement to work on and the existing type table where to add unknown variables. The `parseStatement` function, as the name suggests, parses the statements – in this thesis, as mentioned before, we only deal with uniform choice and assignment. The type rules from Section 4.2 are used according to type info propagation (see Section 5.2). Note, that the input statement is a tree with nodes having a left and a right child. We use shorthand left for the left child and right for the right child.

Taking a look at the first loop (lines 1-3), we can see that for every variable we first check whether it exists in the global table and if not, the given type table is checked. In the end, the variable, if not encountered before, gets added to the type table $\mathsf{type_x}$. Initially, the minimum type for a variable is Min as it is the universal subtype of everything else. The initial maximum type is denoted as Max – the universal supertype of everything else.

It is important to notice that in case of global statements received from the function call in Algorithm 1 we get that $\mathsf{type_X}$ is $\mathsf{type_G}$, which means that new variables will be added to the global variables table. Local type tables are filled when new variables are found inside the game or functions.

To protect ourself against uninitialised variables, we have a set named Defined.

Algorithm 2: parseStatment

Input: statement s, type table $\mathsf{type_X}$

Result: Updated type info in $\mathsf{type_X}$ or $\mathsf{type_G}$ for each variable in s; Error messages if they occur

```
1  foreach variable var in s do
2      if var ∉ type_G and var ∉ type_X then
3          add var to type_X;
4  add s.left to Defined;
5  switch s do
6      case statement is uniform choice (←)
7          updateMinType(s.left, s.right);
8      case statement is assignment (:=)
9          foreach variable var in s.right do
10             if var ∉ Defined then
11                 errorMessage(var is not defined before using);
12             updateMaxType(var, s.left.maxType);
13         updateMinType(s.left, parseExpression(s.right));
```

Variables are added to this set after appearing on the left side of a statement ($\underline{a} := b + c \cdot 5$; $\underline{x} \leftarrow X$). As the parsing works line by line, the existence in the Defined set is checked for every variable found in any expression (right side of assignments, $a := \underline{b} + \underline{c} \cdot 5$).

For the variable on the left side of any statement, the maximum type cannot be updated, since no additional information is received. Updating the minimum type, on the other hand, is a bit more tricky and requires parsing the expression (type info propagation upwards in Section 5.2).

Now let us look at the variables on the right. When randomly choosing from a set, there is only one variable and that is the one on the left side ($\underline{\mathrm{x}} \leftarrow \mathrm{X}$) of the statement, therefore, our work here is done. With assignment, on the other hand, we know that all the variables on the right side, must be in a subtype of the maximum type of the variable on the left side (type info propagation downwards in Section 5.2).

5.4.2 Parsing Expressions

Next we discuss `parseExpression` (Algorithm 3). The only argument passed to Algorithm 3 is an expression. It can either be a number, a variable, a unary

Algorithm 3: parseExpression

```
   Input: Expression exp
   Output: Minimal type of the expression exp
   Result: Updated maximum types for all variables in exp; Error messages if
           they occur.
 1 if exp is a number then
 2     return Int;
 3 if exp is a variable then
 4     return exp.minType;
 5 if exp.operator is finding the opposite (−) then
 6     return parseExpression(exp.right);
   // parsing binary operations
 7 left ← parseExpression(exp.left);
 8 right ← parseExpression(exp.right);
 9 switch exp.operator do
10     case integer division (\div) or remainder (%)
11         foreach variable var in exp do
12             updateMaxType(var, Int);
           // to catch errors of wrong division use, e.g. (5/4)%3
13         if (left ≠ Int) or (right ≠ Int) then
14             errorMessage(Both sides of the expression exp have to be of type
               Int. Did you want to use integer division (\div) instead of (/)?);
15         return Int;
16     case division (/)
17         if left = Int and right = Int then
18             return Rat;
19     case exponentiation (^)
20         foreach variable var in exp.left do
21             updateMaxType(var, Rat);
22 return LCA(left, right);
```

operation of another expression or a binary operation between two expressions. The subexpressions can be parsed separately and the results are merged using type rules. The job of figuring out the order of operations is already done by ProveIt parser. The abstract syntax tree also allows us to easily access both sides of the main operator. As a result we are interested only in updating the maximum

types and returning the minimum type (as we use it to update minType, see line 13 in Algorithm 2).

The limitations posed by different operators (see Section 4.3) are covered in the switch construction (lines 9 to 21). Note, that we can always use the lowest common ancestor of both sides (LCA(left, right)) as the result, but with operations only defined on integers we return Int instead. This is done to get less false negatives because if the function updateMaxType (line 12) fails, we will return a value for the remainder (%) or integer division (\div) operation that is not Int. This returned value may then get updated as the minType of the assignee, which causes errors in every location, where the assignee is used as an integer. From this we can see why it is good to output errors in the order they occur as opposed to grouping them by the name of the variable.

5.4.3 Updating the Maximum Type

When updating the minimum and the maximum type, if an error occurs, the type will not be updated and an error message will be output instead. This way less false negatives gets broadcast. Consider the following simple example

$$a := 5 \ / \ 2$$
$$b := a \ \% \ 6$$
$$c := 8 \ \backslash\mathsf{div} \ a \ .$$

Based on the first line, the minimum type of a is Rat. The second line demands a to be Int producing an error. If we change a to be of type Int then we would also get an error on the first line in the second round of type inference. We do not change the type to Min or any other special type to represent an error either, because we want to get an error message for the third line also to show all wrong uses of variable a.

As updating the maximum type is easier, we will start with that. There are two possible places where we have to deal with the maximum type of the variable. Firstly, we have to consider whether an operation is allowed with the variable in question (see lines 12 and 21 in Algorithm 3). Secondly, we parse assignments (see line 12 in Algorithm 2), where we know that the types of all variables on the right side have to be in a subtype of the variable type on the left side.

Updating maxType is relatively easy, because we only need to check whether there is a mismatch with the minimum type (minType has to be a subset of maxType). Next, if the new type is stricter, we update it (see Section 5.2).

Algorithm 4: updateMaxType

Input: Variable var, Type type

Result: Updated variable type or an error message

1 **if** var.minType $\not\subseteq$ type **then**

2   errorMessage(var *cannot be used as* type *as it has to be a subtype of* var.minType);

3 **else if** type $\subset$ var.maxType **then**

4   var.maxType = type;

5.4.4 Updating the Minimum Type

Updating the minimum type is more complicated compared to updating the maximum type. To better understand the problem, let us discuss the type relation graph.

In every game there are different types, such as Int, Rat, A, D, X. Since we do not need to have multiple equivalent type classes, type relations will be represented as a directed acyclic graph. In this thesis, we are not checking the correctness of the type relations graph. To get a better overview let us look at Figure 16 for an example. In the graph the arrows point to subtypes. We do not have Max or Min on the graph as they do not really exist.

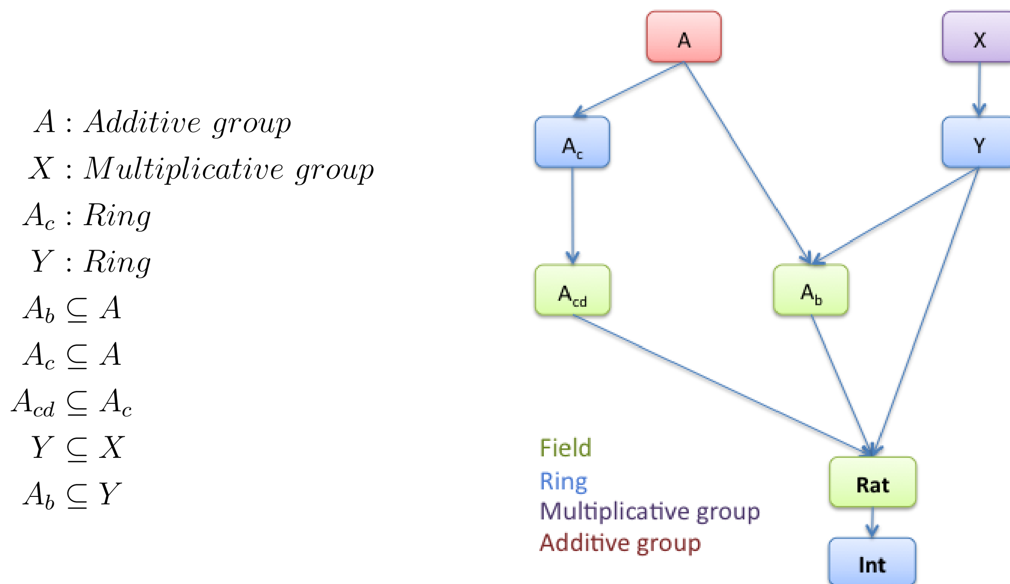


Figure 16: Type relations example with the according graph on the right

We need the type graph to know whether there exists a sufficient type for a variable and what it is. For example (using type relations from Figure 16)

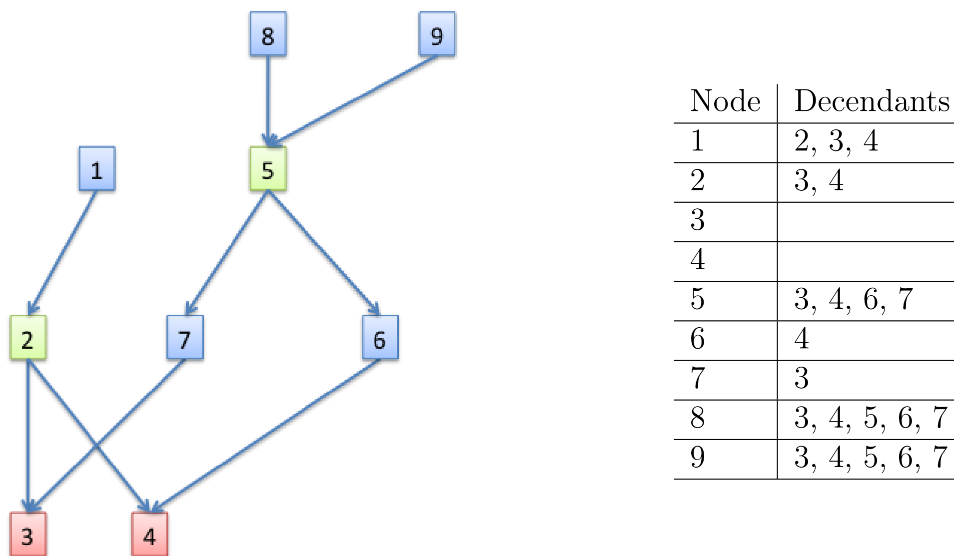
$$a \leftarrow A$$
$$a \leftarrow X$$

will give us a type error because there does not exist a type that is a supertype of both A and X. This example

$$g \leftarrow A_{cd}$$
$$g \leftarrow A_b$$

is a valid one and gives us information that variable g is of type A.

In order to update the minimum type, we first need to create the type relations graph and then find the minimum supertype. The latter is known as finding the lowest common ancestor (LCA) in a directed acyclic graph (DAG) [KL05, CKL07]. LCA of vertices u and v in a DAG is an ancestor of both u and v which has no descendant that is an ancestor of u and v [KL05], see Figure 17 as an example. The LCA for two nodes in a DAG may have many solutions, but we want the variable to have a concrete type. If we cannot determine the one minimum supertype then an error is output asking the user to provide more information. Algorithm 5 gives the function for finding the minimum supertype of the two given types.



Node	Decendants
1	2, 3, 4
2	3, 4
3	
4	
5	3, 4, 6, 7
6	4
7	3
8	3, 4, 5, 6, 7
9	3, 4, 5, 6, 7

Figure 17: DAG with nodes that have two LCAs (LCA(3, 4) = {2, 5}). On the right side are the descendants for the DAG

```
Algorithm 5: LCA - Lowest Common Ancestor
   Input: Types A, B
   Output: minimal supertype if it exists, Min otherwise
 1 Result = {};
   // Breadth first search from A
 2 Queue Q = {A};
 3 while Q is not empty do
 4     Node = take first element of Q;
 5     if B is a descendant of Node then
 6         Result = Result ∪ {Node};
 7     else
 8         foreach parent p of Node do
 9             add p to the end of Q;

   // Deal with the result
10 if there is exactly one result (|Result| = 1) then
11     return Result;
12 else if Result is empty then
13     errorMessage(There is no common supertype for A and B);
14 else
15     errorMessage(There are multiple options for the minimum common
       supertype of A and B (Result), please provide more info);
16 return Min;
```

In Algorithm 5, we do a breadth first search from the first type A. We have found an LCA if B is a descendant of the current node. There are two options to find out if a node is a descendant of another node. We can either precompute the list of descendants for all the nodes or do a search down from every node we reach. As we might traverse the DAG multiple times pre-computation is more efficient. The list of descendants for every node based on Figure 17 is can be seen in Table 17. If the current node is an LCA then we will not look at its parents as they are common ancestors but not lowest common ancestors. Finally we will either return the one LCA or write out an error and return Min – as then the minimum type will not get updated and false types will not go further.

When we have found the lowest common ancestor then updating minimum types is very similar to updating maximum types (see Algorithm 6). We make sure minType is a subtype of maxType and if the new type is stricter, we update minType.

Algorithm 6: updateMinType

Input: Variable var, Type type

Result: Updated variable type or an error message

```
1 type = LCA(var.minType, type);
2 if type ⊄ var.maxType then
3     errorMessage(var cannot be used as type as it has to be a subtype of
      var.maxType);
4 else if var.minType ⊂ type then
5     var.minType = type;
```

5.5 Implementation

ProveIt is written in C++ using the Qt framework. To make code merging easier, we chose the same working developer toolset to implement type inference and type checking. Implementation of typing for all of the constructions of the ProveIt language is not completed as important parts, such as typing logical expressions and functions are not in the scope of this thesis. The type checker works with arithmetic expressions. It successfully recognises the scopes of the variables and creates tables with the info about type ranges and finds the kind of errors given in Section 5.3.

Some things are done differently in the implementation compared to the algorithms given to gain better efficiency. The code can be accessed by contacting the ProveIt research team.

6 Conclusion

This thesis introduces types into the ProveIt language, making it possible to perform type inference and type checking. The ProveIt language is a is a probabilistic program language used in ProveIt, a tool that helps cryptographers prove cryptographic protocols in a game-based way in the computational model. The work begins by explaining the necessary background knowledge: the computational model, game-based proving, the ProveIt prover tool and the ProveIt language. How they work and why they are relevant is also described.

Introducing types into ProveIt allows further guarantee the correctness and providing better help to the user. Knowing that the user is tying to find the square root of a banana and where, might save a lot of his or her time. Although it would be easier to demand the user to state all the types for all the variables and only perform type checking, we decide not to do so and instead infer the types. This is done because our goal is to make the proving process as convenient for the user as possible.

To do type inference, the work starts with defining the formal type system for the ProveIt language. Type rules for all the arithmetic operations available in ProveIt are provided and also type rules for the uniform choice statement and the assignment statement. Adding rules for typing functions and logical expressions is not in the scope of this thesis and is considered future work.

In addition to the type system description, type checking and type inference were implemented. To do that, the thesis first deals with the scopes of the variables in ProveIt, as it has both local and global variables. Secondly, the relations between types are covered, including the need and means to find lowest common ancestors in directed acyclic graphs. This is necessary to be able to find the minimal common supertype if an arithmetic operation was performed between variables of different types. Next, the concepts of a minimum and a maximum type are introduced to make type inference easier and also be able to output better error messages. Finally six algorithms are given to complete the type inference and type checking.

7 Tüübituletus krüptograafiliste protokollide tõestaja jaoks

Bakalaureusetöö (6 EAP)
Tiina Turban
Resümee

Krüptograafia uurimisrühm Tartus arendab mängupõhiste krüptograafiliste protokollide tõestajat nimega ProveIt. Idee ProveIt'i taga on lihtsustada tõestamise protsessi. Tegemist ei ole automaatse tõestajaga, vaid abivahendiga krüptograafidele. ProveIt võimaldab teadlastel teha muudatusi koodina üleskirjutatud tõenäosuslikele mängudele ja teeb koodi ümberkirjutamise automaatselt. See aitab garanteerida korrektsuse, hoida kokku aega ja samas jätta tõestuse arusaadavaks.

Üks võimalus kirjutada üles krüptograafilisi protokolle on kirjeldada neid programmeerimiskeelele sarnase pseudokoodiga, mida nimetatakse mänguks. Mängu muutmisel samm-sammult saame mõnikord jõuda teise mänguni, mille turvalisuse hinnangut me teame. Nende mängude jada ongi üks võimalik viis, kuidas saab anda protokolli turvalisuse tõestuse - seda nimetatakse mängupõhiseks tõestamiseks. Mängupõhine tõestamine on populaarne viis analüüsimaks protokollide turvalisust, sest see on võrdlemisi intuitiivne. Tavaliselt kasutatakse paberit ja pliiatsit, et kirjutada üles mänge, kuid kuna krüptograafiliste protokollide tõestamine ei ole just kõige lihtsam asi, mida teha, siis inimesed sageli lõpetavad suure kuhja paberitega igal pool, sealhulgas mitmete tupikute ja vigadega. Meie eesmärgiks on seda muuta ja anda krüptograafile võimalus mängu ümber kirjutamise asemel keskenduda järgmise sammu väljamõtlemisele.

Kuigi krüptograafial on oluline roll selle bakalaureusetöö juures, siis põhi fookus lasub siiski tüübiteoorial. Nimelt on selle töö eesmärgiks luua tüübikontroll ProveIt'i jaoks. Mängupõhiseid tõestusi saab edukalt kirjutada ilma tüüpidest midagi teadmata, aga tüüpide kasutamine ProveIt'is võimaldab tagada kindlamalt korrektsuse ja kasutajat paremini aidata. Tüübikontrolli abil saame näiteks veenduda, et krüptograaf ei ürita omavahel liita numbreid ja pesukarusid. Selle fakti ja vea asukoha teadmine võib kokku hoida palju kasutaja aega.

Lihtne lahendus tüübikontrolli jaoks oleks nõuda kasutajalt kõigi muutujate tüüpide defineerimist, kuid see nõuaks temalt lisatööd, mis on aga vastuolus ProveIt'i eesmärgiga. Järelikult tuleb lisaks tüübikontrollile realiseerida ka tüübituletus ehk programm, mis üritab ise kõigi muutujate tüüpe määrata. Selleks, et teha tüübituletust defineerisime me kõigepealt tüübisüsteemi ProveIt'i keele jaoks. Andsime tüübireeglid aritmeetiliste avaldiste, juhuslikult hulgast valimise ja omistamise jaoks. Funktsioonide ja loogiliste avaldiste tüüpimine ei ole käesoleva bakalaureusetöö skoobis, kuid need on vajalikud osad ProveIt'ile täieliku

tüüpija lisamiseks.

Lisaks tüübisüsteemi ja tüübireeglite kirjeldamisele kuulus käesoleva bakalaureuse töö juurde aritmeetiliste avaldiste tüüpimise realiseerimine ja ProveIt'ile lisamine. Selle jaoks on töös antud kuus algoritmi. Arendusvahenditena valisime Qt raamistiku ja programmeerimiskeele C++. Otsustasime keele C++ kasuks, sest ProveIt on selles keeles kirjutatud, ning see võimaldab ühildamise lihtsamaks muuta.

References

[AR00] Martín Abadi and Phillip Rogaway. Reconciling two views of cryptography (the computational soundness of formal encryption). In *Proceedings of the International Conference IFIP on Theoretical Computer Science, Exploring New Frontiers of Theoretical Informatics*, TCS '00, pages 3–22, London, UK, UK, 2000. Springer-Verlag.

[BGZ09] Gilles Barthe, Benjamin Grégoire, and Santiago Zanella Béguelin. Formal certification of code-based cryptographic proofs. In *36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009*, pages 90–101. ACM, 2009.

[Bla] Bruno Blanchet. Proverif: Cryptographic protocol verifier in the formal model. `http://www.proverif.ens.fr/`.

[BR04] Mihir Bellare and Phillip Rogaway. Code-based game-playing proofs and the security of triple encryption. Cryptology ePrint Archive, Report 2004/331, 2004. `http://eprint.iacr.org/`.

[CKL07] Artur Czumaj, Miroslaw Kowaluk, and Andrzej Lingas. Faster algorithms for finding lowest common ancestors in directed acyclic graphs. *Theoretical Computer Science*, 380(1-2):37–46, 2007.

[Kam11] Liina Kamm. ProveIt - Cryptographic Protocol Proving Assistant Protocol Language. Cybernetica AS, University of Tartu, October 2011.

[Kam12] Liina Kamm. ProveIt - How to make proving cryptographic protocols less tedious. Talk at the 21st Estonian Computer Science Theory Days at Kubija, January 2012.

[KL05] Miroslaw Kowaluk and Andrzej Lingas. LCA queries in directed acyclic graphs. In *Proceedings of the 32nd international conference on Automata, Languages and Programming*, ICALP'05, pages 241–248, Berlin, Heidelberg, 2005. Springer-Verlag.

[Sch95] Michael I. Schwartzbach. Polymorphic type inference. Lecture notes (`http://cs.au.dk/~mis/typeinf.pdf`), 3 1995. Last viewed 13.05.2012.

[Sho04] Victor Shoup. Sequences of games: a tool for taming complexity in security proofs. Cryptology ePrint Archive, Report 2004/332, 2004. `http://eprint.iacr.org/`.