

# Automatic Program Verification in Mist

Stijn de Gouw (cdegouw@liacs.nl)  
Michiel Helvensteijn (mhelvens@liacs.nl)

August 31, 2009

## Abstract

Programming is among those processes most prone to human error. A promising way to quickly diagnose and eliminate bugs is formal verification using contract programming and the redundancy offered by in-code assertions. In this thesis we describe our work on Mist, a new verified programming language with contract programming features. We describe its compiler, which can automatically find a formal proof of program correctness and we describe its advantages over a separate verifying tool. An extended Hoare logic is introduced, especially suited to function specification, reducing the burden of proof to a set of implications. We explain how the compiler is integrated with the theorem-prover KeY, which subsequently solves these implications. Finally, a case-study is used to demonstrate the current capabilities of the Mist compiler.

## Contents

<b>1</b>	<b>Introduction</b>	<b>4</b>
1.1	The Problem . . . . .	4
1.2	Possible Solutions . . . . .	4
1.3	A Verifying Compiler . . . . .	5
1.4	A Verified Programming Language . . . . .	5
1.5	Mist . . . . .	6
1.6	Organization of this Thesis . . . . .	6
<b>2</b>	<b>The Mist Programming Language</b>	<b>7</b>
2.1	Syntax . . . . .	7
2.2	Scoping . . . . .	8
2.2.1	Scope Introduction . . . . .	8
2.2.2	Symbol Visibility . . . . .	9
2.2.3	Paths . . . . .	9
2.3	Contract Programming . . . . .	10
2.4	Memory Management and System Communication . . . . .	12
2.5	Non-determinism . . . . .	13
2.6	Comparison . . . . .	13

2.6.1	Syntax . . . . .	14
2.6.2	Assertion Constructs . . . . .	14
2.6.3	Assertion Expressiveness . . . . .	15
2.6.4	Function-call Verification . . . . .	16
2.6.5	Verification and Compilation . . . . .	16
2.6.6	Proofs . . . . .	16
2.6.7	Other Aspects . . . . .	17
<b>3</b>	<b>Verification Techniques</b>	<b>17</b>
3.1	Basic Concepts . . . . .	17
3.2	Consequence rule . . . . .	19
3.3	Assignment . . . . .	19
3.4	If-statement . . . . .	20
3.5	While-statement . . . . .	20
3.6	Procedure Call . . . . .	22
3.6.1	Weakest liberal precondition . . . . .	23
3.7	Sequential Composition . . . . .	25
<b>4</b>	<b>The Compiler</b>	<b>26</b>
4.1	Fundamentals . . . . .	26
4.2	Compilation Passes . . . . .	27
4.3	The Symbol-tree . . . . .	28
4.4	Mist Runtime Environment . . . . .	29
4.5	Integration with KeY . . . . .	29
4.5.1	Prolog . . . . .	29
4.5.2	KeY . . . . .	29
<b>5</b>	<b>Case Study: Polynomial Evaluation</b>	<b>32</b>
<b>6</b>	<b>Conclusion</b>	<b>35</b>
<b>A</b>	<b>Full Case Study Proof</b>	<b>37</b>
<b>B</b>	<b>Grammar</b>	<b>44</b>
B.1	Keywords . . . . .	44
B.2	Tokens . . . . .	44
B.3	Comments . . . . .	44
B.4	Identifier . . . . .	45
B.5	Integer Literal . . . . .	45
B.6	Boolean Literal . . . . .	45
B.7	String Literal . . . . .	45
B.8	Declarations (global scope) . . . . .	46
B.9	Function Declaration . . . . .	47
B.10	Variable Declaration . . . . .	47
B.11	Type Declaration . . . . .	48
B.12	Type . . . . .	48

B.13	Expression . . . . .	49
B.14	Operator Call . . . . .	49
B.15	Function Call . . . . .	51
B.16	Subscripting . . . . .	51
B.17	Statement . . . . .	51
B.18	Expression Statement . . . . .	51
B.19	C++ Statement . . . . .	52
B.20	If Statement . . . . .	52
B.21	Compound Statement . . . . .	52
B.22	For Statement . . . . .	52
B.23	While Statement . . . . .	53
B.24	Assertion Area . . . . .	53
B.25	Reference . . . . .	54
<b>C</b>	<b>Comments</b>	<b>54</b>
C.1	Syntax . . . . .	54
C.2	Recommended Usage . . . . .	55
C.2.1	Documentation . . . . .	55
C.2.2	Zapping Code . . . . .	56
C.2.3	Comment Switches . . . . .	56
<b>D</b>	<b>Tuples</b>	<b>57</b>
D.1	Tuple Variable . . . . .	57
D.2	Parallel Assignment . . . . .	57
D.3	Tuple Member Access . . . . .	57
D.4	Tuples and Arrays . . . . .	58
D.5	Lexicographical Ordering . . . . .	59
D.6	Tuples and Intervals . . . . .	59
D.7	Tuple Flattening . . . . .	60

# 1 Introduction

## 1.1 The Problem

Programming is a very error-prone process. The computer can only protect programmers from a tiny fraction of it. When they commit syntactic errors, the compiler will be quick to point it out. Most modern programming languages have also introduced ways to statically prove type correctness [8] and const correctness (in, for example, the C++ programming language [21]). However, the compiler cannot yet detect the more complex mistakes programmers tend to make. These mistakes will, in the best case, cause a program to crash at runtime. In the worst case, they will result in unintended behavior. These mistakes are known as *bugs*.

A programming language is a tool for programmers to express their ideas. Bugs are discrepancies between the programmer's intention and the idea his code actually expresses. Since no one is perfect, bugs are an unavoidable part of the creative process. Even a simple algorithm such as binary search is complex enough to have contained an undetected bug for half a century [6]. Joshua Bloch comments:

It is hard to write even the smallest piece of code correctly, and our whole world runs on big, complex pieces of code.

## 1.2 Possible Solutions

Many solutions to this general problem have been suggested. The aforementioned type and const correctness systems have been a big step. Other leaps forward include assertions and contract programming [14]. They allow the programmer to express a set of admissible states in the form of a condition that *should* hold at the point of assertion. Basically, the intention of the programmer is expressed twice. The code itself is used for execution, and the assertions offer redundancy. Consistency between the two is tested at runtime. If used properly, this can cause the program to abort closer to the actual location of a bug. Assertions also exist in the form of a function pre- and postcondition (the function *contract*), which specify the conditions that must hold upon calling the function and the conditions that are guaranteed to hold after execution respectively. Other types of assertions exist, which we will explain in Section 2.3.

However, since these assertions are checked at *runtime*, passing an assertion any number of times does not guarantee the absence of bugs. Assertions checked at runtime can merely be used to show their presence. Ongoing research suggests we can do better. Formal methods may be used to statically verify consistency between code and assertion. Examples of such methods include Hoare logic [18], Dynamic Logic [17], VDM [23] or the B-method [4]. Using logic, one may be able to *prove* that for every possible execution path, a given assertion will hold, which has obvious advantages.

If you wish to maintain a correctness proof alongside your code, there are several options available at this time. One solution would be to write the formal

proof manually. But we've established that humans are imperfect, and mistakes are easy to make. It would be an unnecessary burden to manually keep an evolving code-base in sync with its proof at all times. So it would be preferable to automate the process. Various tools have been created to automatically verify programs written in existing programming languages. Examples include several tools for verifying JML [7], VCC [9] for concurrent C and KeY-tool [5] for Java.

### 1.3 A Verifying Compiler

The downside to these tools is that they are completely separate from the compilation process. This has several consequences that we believe cannot be ignored. One obvious advantage of a verifying compiler [20] as opposed to a compiler and a separate verifying tool is that runtime assertion checks may be used exactly where a proof could not be found, and nowhere else. Or, at the user's option, the compiler may refuse compilation if a proof is not found. This is important for safety-critical systems. Another advantage is that the extra information provided by the assertions may be used for optimization of the code in various ways. For example, by finding upper and lower bounds for integral variables, a compiler may discover the maximum amount of memory required to represent them. More significant examples follow in Section 1.4.

### 1.4 A Verified Programming Language

There are limits to what even a verifying compiler can do for a programming language unprepared for such static analysis. One obvious criterium is that the programming language should support the syntax to express assertions. This already rules out most modern programming languages, which offer at most an `assert` function in the standard library. Dedicated syntax is also required for specifying loop (in)variants and function contracts. For an explanation of these, read Section 2.3. Suffice it to say that a programming language needs to be prepared to handle verification.

Aside from the minimum requirements for a verified programming language, there are opportunities for exploiting the potentially powerful optimizations offered by a verifying compiler. A programming language based on the assumption that it will be compiled by a good verifying compiler can afford to be more elegant than other programming languages, secure in the knowledge that the compiler is smart enough to greatly reduce the resulting runtime cost.

For example, if the programming language were to offer the possibility of defining functions based on their contract, rather than their implementation, the compiler would be free to replace any calls to such functions by the most efficient code with a compatible contract. The standard library of such a language might contain a function called simply `sort`, defined by an inductive contract [13], meant to be a placeholder for the implementation of a specific sorting algorithm, which the compiler may select based on any available complexity criteria.

Such a language could also afford to feature only a single unbounded integral type, modeling mathematical integers. Static analysis will, in most cases, indi-

cate the actual amount of memory required to represent a specific `int` variable. In cases where such a limit cannot be found, the variable would automatically be represented by a dynamic arbitrary-precision integer (see, for example, [27]). In such cases, any fixed precision integer representation would be insufficient anyway, as exemplified by the bug described in Section 1.1.

## 1.5 Mist

We believe a verified programming language that takes full advantage of these possibilities does not yet exist. And so we introduce the Mist programming language, designed to do just that.

Mist offers the syntax to express function contracts, loop (in)variants and in-code assertions. It has only one integral type, as explained in Section 1.4. It allows function-definition by contract. Such functions guarantee nothing more than what their post-condition states, so any implementation can remain encapsulated and be verified separately from the calling code. This ensures modularity.

The Mist compiler will verify the consistency between assertion and code where it can. It will store the proofs and reuse them if the related code has not changed.<sup>1</sup> These proofs may also be made human-readable.

If proof of assertion correctness cannot be found, the compiler will fall back to runtime checks, and show the programmer which assertion may fail; if possible, with a state description constituting a counter-example.

The compiler will guarantee that each statement is executed in a state consistent with past assertions, as the assertion system may not be circumvented. When an assertion fails, there is no recovery, since it indicates a flaw in the program. Such strong guarantees offer many opportunities for optimization, which the compiler will take full advantage of.

## 1.6 Organization of this Thesis

The remainder of this thesis is organized as follows: Section 2 describes in more detail the design of the Mist programming language. This is followed in Section 3 by a formal treatment of the logic used for program verification. In Section 4, we explain how the compiler works, and how it is integrated with KeY. Section 5 takes you through a proof of an example Mist program, step by step. (The complete proof of the case study is available in Appendix A.) Finally, Section 6 concludes this thesis, and discusses future work.


This thesis has a number of appendices, describing several aspects of the Mist programming language that are not relevant for the thesis proper, but nonetheless provide a good insight into the Mist language (current and future versions). Appendix B contains the grammar of the current Mist language in EBNF [1], with descriptions. Appendix C explains the comment system of Mist


---

<sup>1</sup>It takes far less time to verify an existing proof than to find a new one.

in more detail and includes recommended usage. Appendix D describes the future tuple system of Mist, which will become an integral part of the language.

Throughout this thesis, the following two icons will be used:

 *This icon indicates a warning. Something that is important to know, but may not be immediately obvious.*

 *This icon indicates information about future development of the Mist programming language. It is important to read these, for they also illustrate current limitations.*

## 2 The Mist Programming Language

Mist is an imperative programming language, with plans to expand into the object oriented and functional paradigms. It is statically typed and statically scoped. But its most noteworthy aspect is its integrated assertion framework and the resulting choices with regard to its design.

In Section 2.1, the look-and-feel of the Mist language is introduced. In Section 2.2, we explain the scoping system of Mist, which is important for understanding the case-study in Section 5. Then we describe the Mist contract programming facilities in Section 2.3. Section 2.4 explains the consequences of its design on the lower level system programming possibilities. There are many interesting Mist features, but they exceed the scope of this thesis. Some of them are described in the appendices. In Section 2.6, we compare Mist to other existing verified programming languages.

### 2.1 Syntax

The Mist programming language will offer a syntax close to mathematical notation, yet not so far from that of C-family programming languages that it would confuse experienced programmers after an introduction. It is designed to be intuitive. The full design of the language is incomplete as of yet, but the current syntax has been planned to elegantly accommodate the object orientation and functional programming styles in the future.

A brief example follows to give the reader a feel for the language. For a full grammar, consult Appendix B.

```

| a line comment, starting with a pipe
| character and ending with a newline

int i <- 42;    | variable declaration and initialization
i <- i * i;    | assignment using <-, arithmetic operators
@ i = 42 * 42  | assertion using @, equality using =
@   = sqr(42) | forward referencing of functions
@   = 1764;   | linebreaks in assertions, comparison chaining

int sqr(int v) {      | function signature
    result <- v * v; | function body, implicit result variable
}

```

Note that this example is meant merely to illustrate the look-and-feel of the language, and displays not nearly all its features.


## 2.2 Scoping

This section will introduce the scoping mechanism of Mist, which aids in comprehension of the coming code examples. Its content originates from the Mist 0.1 manual [10].

### 2.2.1 Scope Introduction

The main purpose of scopes is to determine symbol visibility and variable lifetime. Scopes come in some different shapes and sizes. The following different kinds of scopes exist:


- There's the global scope. It is unnamed. It can contain variables and functions.
- There are functions. They introduce a named, opaque scope. They can contain variables, (nested) functions and local scopes.
- There are local scopes. They may be named or unnamed. They are opaque. They can contain variables, (nested) functions and local scopes. They are introduced by a compound statement.
- If-statements, else-clauses, while-loops and for-loops also always introduce an opaque scope. If a compound statement is used there, it takes the place of the scope that would have been created, so these scopes may also be named.

 *These scopes are all opaque, except perhaps for the global scope, for which this classification is meaningless. An opaque scope will not allow an outside reference to a symbol inside. This, of course, implies the existence of transparent scopes. In the future, classes will introduce transparent scopes.*

### 2.2.2 Symbol Visibility

Symbol visibility is determined by the static scope rule: for any reference to a symbol, one can determine which (if any) declaration applies to that reference by examining the program text.

A reference to a symbol  $x$  refers to the *most deeply nested declaration*  $x$  in a block that encloses the reference. Variables have to be declared before they are referenced. Function declarations, however, do not need to precede any reference to them (forward referencing). Forward referencing of a constant symbol (such as a function) is possible since its definition will never change during its lifetime. Whereas referencing a variable before its declaration would be meaningless.

 *Currently the only constant symbols are functions. In the future, constants (variables with an immutable value) will be supported as well.*

The precondition has access to the formal parameters, the logical variables and the template types. The postcondition has access to the same, and the `result` variable (storing the return-value of the function). The function body has access to all of these, and to local variables. All three constructs have access to symbols in shallower visible scopes.

A nested function can refer to variables declared in the enclosing scope (or any of its ancestors). Calling such a function is only permitted if all variables declared in the same scope as the function-call referenced by the callee have been declared before the call. Otherwise you would be referencing an undeclared and uninitialized variable. The compiler conservatively gives an error if there is the possibility of a reference to an undeclared variable.

### 2.2.3 Paths

The simplest way to reference a symbol is to give only the name of this symbol. However, in different scopes, symbols with identical names may be declared.

```
1  int a;  
2  void f() {  
3      int a;  
4      void g () {  
5          int a;  
6          a <- 42;  
7      }  
8  }
```

According to the most-deeply-nested rule (see Section 2.2.2), the reference to `a` on line 6 refers to the `a` declared on line 5. Suppose in the `g`-function we want to assign 42 to the variable `a` declared on line 3. Without a more sophisticated scheme than just using the name, this is impossible. To address this issue, we introduce the notion of a path to a symbol.

We can refer to the symbol `a` from line 3 by giving more information about the scope it's declared in. The statement `f.a <- 42;` assigns 42 to the `a` de-

clared on line 3. `f.a` is called the *path* to that variable. Global symbols can be referred to by starting with a dot: `.a` refers to the `a` from line 1. Note that `.f.a <- 42;` would also refer to the `a` declared on line 3 (there can be multiple paths to the same symbol). Paths starting with a dot are called *absolute paths* and paths starting with a scope are *relative paths*.

## 2.3 Contract Programming

The most basic assertion is the one you can put anywhere a statement is expected. It specifies a condition that must hold at that point. If it does not always hold, there is a flaw in either the assertion or the preceding code.

```
if (b) { @ f(g); 0 <= i <
        @ j < N <= MAX; }
```


This example illustrates three points:

1. You can put more than one assertion on the same line using only one `@`. Terminate each one with a `;` (semicolon).
2. You can split one assertion over multiple lines, as long as each succeeding line starts with a `@` (except for whitespace).
3. A closing bracket that cannot be matched within the assertion area terminates the area and tries to match an opening bracket outside it.

Assertions have a dedicated syntax to set them apart from functions. They should be seen as checkpoints in the code, but not really part of it.

Assertions are not allowed to contain any side-effects. So given a correct program, assertions not used in function-definitions (i.e. used solely for redundancy) may be omitted without changing the observable behavior of the code.

Otherwise, assertions may contain anything a regular Mist boolean expression can contain, and nothing more. This means that function-calls are permitted, if they do not perform side-effects. But universal and existential quantification are may not be used. It is important as a fallback mechanism that any assertion be testable at runtime. Quantification over infinite sets is not.




*Because quantification may not be used in Mist, it is currently not possible to express array properties such as ‘is sorted’. However, in the near future Mist will support specification by induction, which will enable you to express such properties, as well as some properties not expressable with first-order quantification, such as ‘is permutation of’.*

In order to verify programs with while-loops, it is important for the programmer to be able to specify the *loop invariant* and the *loop bound*. The loop invariant is an assertion that should hold before the loop and after each iteration of the loop. It is specified as a loop section with the `invariant` keyword. It can be used to prove partial correctness of the containing function. The loop bound (also known as loop variant) is an integer expression that strictly decreases in

value each iteration, but remains greater or equal to 0. It is specified with the `bound` keyword. The existence of such an expression proves termination of the loop.

```
int i <- 0;
while (i < 10)
bound (10 - i)
invariant { @ 0 <= i <= 10; }
do { i <- i + 1; }
```

But the feature that lends ‘Contract Programming’ its name is the possibility to specify function contracts. A *function precondition* specifies the condition that must hold for any call to the function to be defined. It is specified as a function section with the `pre` keyword. The *function postcondition* specifies the condition that is guaranteed to hold after execution of the function. It is specified with the `post` keyword. Because a function with a non-void return-type implicitly declares the `result` variable, the postcondition automatically has access to the return value of the function. *Logical variables* may be declared in function sections and given meaning in the precondition. Logical variables may only be used in assertions and are guaranteed, for each function execution, to remain constant, so the postcondition can use them to refer to expressions as evaluated in earlier states. They are declared using the `logical` keyword.

 *We should make it clear that these logical variables are not global; they do not have only one value throughout the execution of the program. They are local to each function-invocation: each call manifests its own instantiation of the logical variables.*

```
| .....|
| Calculates the square of the value v. |
|-----|

int sqr(int v)
logical int v0;
pre { @ v0 = v; }
body { result <- v * v; }
post { @ result = v0 * v0; }
```

If the precondition is left out, it is taken as the precondition ‘true’, accepting any arguments or state in the calling code.



Currently, all functions in Mist are defined by their contract, not their implementation. If the postcondition is left out, it is implicitly taken as the postcondition ‘true’. This would mean giving no guarantees at all<sup>a</sup>, and the compiler would be free to skip any call to this function or, in fact, replace it with any code at all.

<sup>a</sup>Assuming partial correctness. In the setting of total correctness, even the triple  $(\text{true}) f (\text{true})$  guarantees that the function  $f$  terminates.



In the future, it will become possible to define a function by its implementation. This implementation would have to be public, since that would be the only way calling code could be verified. Such a function can, of course, still have a precondition as part of the definition.

To illustrate the importance of logical variables, let us try to rewrite the `sqr` contract without them. It would probably look like this:

```
int sqr(int v)
pre { @ true; }
post { @ result = v * v; }
```

But if `sqr` were defined by this contract, any functions calling `sqr` and expecting it to return the square of the passed argument could not be proved correct. For all you know, the following could be used as an implementation of that contract:

```
body { result <- v <- 0; } | assign 0 to both result and v
```

It is important that the postcondition uses the old value of `v`, and logical variables are the way to refer to it.

A last possible function-section, the *recursive function bound*, is used to prove termination of recursive functions. It is an integer expression that is guaranteed to decrease between the beginning of the function and just before any recursive call, yet remain greater or equal to 0. It serves the same purpose as the loop bound. An example of its use can be found in Section 5.

## 2.4 Memory Management and System Communication

Mist is not a system programming language (as opposed to an application programming language). As a result, the programmer will not have fine-grained control over communication with the operating system or the hardware. Giving programmers this control would limit the amount of useful static analysis that can be performed. Optimization and automatic proof generation would be much more problematic, if not impossible.

To be more precise: System calls will be wrapped by Mist functions and only their contract will be exposed to the programmer. Memory management will be out of the programmer’s hands. This includes memory layout, garbage collection and calculation of memory addresses. Most programmers will not be hindered by this. There are, however, programmers that prefer to have a firm

grip on the time and space complexity of their code. We will do our best to accommodate them, without giving direct access to low-level facilities.

A useful by-product of this decision is that programs written in Mist will be inherently platform-independent. Although it may become possible in the future to invoke OS-specific system calls through a wrapper, such wrappers will be clearly marked in the documentation, and the compiler will understand and report the limits of the resulting software.

## 2.5 Non-determinism

A programming statement  $S$  is said to be *deterministic* if, given an initial state, executing  $S$  can result in not more than one possible final state, as determined by the semantics of the programming language. If a statement is *non-deterministic*, the language standard allows multiple possible final states. An example of a non-deterministic program is one that uses multiple threads to influence the global state. How the execution of the threads is interleaved is typically not specified, and can influence the final state of the program.

If there are a finite number of possible outcomes, a program is said to exhibit *bounded non-determinism*. The example of arbitrarily interleaved program threads is non-deterministic in this way, since there are only so many ways to interleave the operations of each thread.

*Unbounded non-determinism* does not have this restriction. A statement with such non-determinism may result in any of an infinite number of states.

Mist has two sources of non-determinism. Expressions with side-effects may introduce bounded non-determinism, as the order in which sub-expressions (and so, side-effects) are evaluated, is not determined.

Mist features functions that are defined by their contract (rather than their implementation). Calls to such functions introduce unbounded-nondeterminism, as the compiler is free to replace the call with any code that satisfies that contract. Contracts exist that allow infinitely many resulting states. The function postcondition  $(x > 0)$ , for example, is satisfied by any function implementation that leaves the value of  $x$  positive. And nothing is guaranteed about possible other variables, which may take any value.

The compiler can not be expected to derive a function body from the contract. The point is that it's allowed to. It may also choose between any implementation provided by the programmer, which is what will happen in practice.

Unbounded non-determinism allows for potent high-level optimization by the compiler. If a number of sorting-algorithms are known, the compiler is free to replace any call to `sort` by the most efficient implementation available.

## 2.6 Comparison

This section offers a comparison between the plans for Mist and the design decisions and toolchains of existing verified programming languages. We compare Mist to Verified C, to JML and to Eiffel. This section will be subdivided by aspect.

### 2.6.1 Syntax

Verified C is basically concurrent C with verification syntax added through the use of preprocessor macro's. If compiled, these macro's expand to nothing, so the function specifications and assertions are always ignored by the compiler.

```
void foo(int x)
    requires(0 < x && x < 20)
{
    x++;
}
```

The `requires` clause provides the precondition.

JML is Java, with contracts and assertions given by means of annotation comments.

```
/*@ requires a != null
    @      && (\forall int i;
    @          0 < i && i < a.length;
    @          a[i-1] <= a[i]);
    @*/
int binarySearch(int[] a, int x) {
    // ...
}
```

Eiffel may be the only programming language in existence that started out with advanced contract programming facilities in the core design of the language.<sup>2</sup>

```
set_hour (h: INTEGER) is
    require
        0 <= h and h <= 23
    do
        hour := h
    ensure
        hour = h
    end
```

Mist is, like Eiffel, a language with contract programming facilities built in. Code examples have already been given in Section 2.3.

Eiffel and Mist have the advantage of a language tailored to work with the assertion framework, whereas Verified C and JML add an additional layer of syntax on top of an existing language.

### 2.6.2 Assertion Constructs

Verified C, JML, Eiffel and Mist all support in-code assertions, function pre- and postconditions and loop (in)variants. The first three also support type invariants.

---

<sup>2</sup>In fact, Eiffel popularized and trademarked the phrase "Design by Contract"



*Mist does not support user-defined types yet, so there has been no place for type invariants. But both will be available in future versions.*

JML introduces the **assignable** function-clause, which expresses the fact that no non-local variables other than the ones in the given list may be modified by the function. This makes it possible to write modular specifications that do not require change whenever another field is added to the class. While this clause is used in verification of calls to the function, its validity is not actually tested in the function itself, so there is the possibility for unsound proofs. Eiffel has a similar clause, denoted **only**.



*Mist will soon have a similar function/loop clause in the language, called **changesonly**. However, it will always be considered in the proof of the function, unlike in JML.*

Verified C offers constructs designed to help verification of concurrent code. The other three languages do not. Mist in particular is focussing on verification of sequential programs at the moment.

One other thing that JML supports, but the other languages do not, is a function specification of exceptional termination. This specification can list the guarantees that hold when the function throws an exception.

### 2.6.3 Assertion Expressiveness

The assertions and contracts given in Verified C are interpreted by VCC, the C verifier. They can contain C boolean expressions, but also additional constructs such as universal and existential quantification operators. This makes the Verified C assertion language more powerful, but it also implies that its assertions cannot in general be tested at runtime. The C library provides its own runtime-checked assertions. It would require code duplication if the programmer wants both.

A JML program may be compiled with an ordinary Java compiler. But there is also a JML compiler available that can transform assertions and specifications into runtime checks. Other tools are also available which can do a more thorough static analysis. JML supports quantification in its assertions as well, but requires a finite range to make them executable. If the range is omitted, the quantifier may still serve as documentation, and may perhaps be used in automatic verification.

The Eiffel standard reference [3] mentions:

With advances in formal methods technology, [assertions] open the way to proofs of software correctness.

While this is certainly true, as of yet, no automatic proof derivation is attempted in its compiler or associated tools. Eiffel only does runtime checks of assertions.

Neither Eiffel nor Mist supports quantification over infinite ranges. This makes the assertion language less powerful, but it gives the advantage of always being able to execute assertions at runtime if necessary.



*Mist may in the future allow quantification over finite collections or sequences, even outside assertions, because they can always be evaluated in finite time, if the predicate bound to the quantifier can.*

In Verified C, JML and Mist, an assertion failure constitutes a flaw in the program that can not be recovered from. In Eiffel, this is not so, because it throws an exception that may be caught and handled. In this way, Eiffel assertions offer weaker guarantees than do those in Mist. In Eiffel, a contract violation is handled the same as a recoverable user error. In Mist, the two are completely separate concepts.



*Certain kinds of failures exist that a program should be able to recover from, yet cannot be tested for beforehand because they occur asynchronously. For example: 'out of memory' failures and filesystem/network access failures. These should throw an exception, and so Mist will eventually feature exception handling. However, exceptions will remain separate from assertion failures. What's more, the programmer will have the opportunity to provide a special postcondition, as is possible in JML (see Section 2.6.2), describing the guarantees after a function throws an exception. Eiffel does not offer this possibility.*

#### 2.6.4 Function-call Verification

In both Verified C and JML, function-calls are verified by looking at the contract of the function being called, not its implementation. This has the advantage of encapsulation, and thus modularity.

Mist also supports this, for functions defined by contract. But Mist will also support implementation-defined functions, however, which are 'inlined' into the calling code for verification. This only works for non-recursive functions. Calls to implementation-defined recursive functions can not be verified automatically.

#### 2.6.5 Verification and Compilation

Verification of Verified C code is completely separate from its compilation. An ordinary C compiler is used to generate the executable code, so apart from the verification itself, verified C has none of the advantages of a verified programming language. The same goes for JML. Eiffel does not verify at all.

In this aspect, the Mist compiler has the advantage of being aware of the verification information provided by the programmer, and Mist is designed accordingly. The compilers of the other three languages can not optimize based on programmer provided assertions.

#### 2.6.6 Proofs

Verified C is translated to BoogiePL [11], an intermediate language for verification purposes. JML is also first translated to another language.

Mist, however, generates a proof directly around the given code. This makes proofs accessible to the programmer. He or she may be able to verify by hand that the proof is sound. This is not practical for Verified C and JML. The automatically translated code is not designed for human eyes.

### 2.6.7 Other Aspects

Because Verified C is really just C, it is more suited to system programming than to application programming. In that respect, the opposite of Mist.

The design of Eiffel is in general also quite different from that of Mist. Eiffel does not feature function overloading, global functions or functions with side-effects, to name a few. Some of these decisions are meant to enforce a certain programming style. Lack of global functions enforces object orientation. *Command/query separation* (the decision that a routine may either return a value or produce side-effects, but not both) enforces Design by Contract principles. Lack of function overloading is meant to protect the programmer from himself.

Mist does not enforce a specific style of programming. And, as a rule, the programmer is trusted to use the tools handed to him wisely, so we try not to restrict him unless there are technical reasons for doing so.

## 3 Verification Techniques

In this section we introduce Hoare-style proof-rules for the basic constructs used in Mist-programs: assignments (Section 3.3), `if`-statements (Section 3.4), `while`-statements (Section 3.5), procedure calls (Section 3.6) and sequential composition (Section 3.7). This is only a small subset of the Mist-language. More advanced constructs such as `for`-loops and expressions containing side-effects are introduced in Appendix B. There are essentially two ways to deal with these constructs if one wants to verify programs containing them:

1. Proceed by introducing new proof rules for the new constructs. This can result in very complex rules, but is suitable for `for`-loops.
2. Apply program transformations to write the new constructs in terms of known constructs. This can be used to translate side-effecting expressions into non-side-effecting expressions (in the form of three-address code statements) by creating temporary variables to hold intermediate results.

We will not consider these constructs in the remainder of the section.

### 3.1 Basic Concepts

Our proof calculus is an extension of the traditional Hoare-calculus [18] with parameterless (recursive) procedure calls and a modified consequence rule to accommodate logical variables. Both rules are based on the work of Kleymann [28].

Syntactically, Mist assertions are first order logic formulae, only without quantifiers. Semantically, assertions are modelled as sets of states. All free

variables in an assertion are either program variables or logical variables, so an assertion depends on the logical state, determining the values of the logical variables. For example, if  $x$  and  $y$  are program variables and  $L$  is a logical variable with value 0, the assertion  $x = L$  and  $y > 0$ ; represents the set of all states in which  $x$  has the value 0 and  $y$  is positive.

We will use the notation  $\sigma \xrightarrow{\mathbf{S}} \tau$  if we execute a program  $\mathbf{S}$  in an initial state  $\sigma$ , and the resulting state is  $\tau$ . We can now introduce the concept of a weakest (liberal) precondition. Given a program  $\mathbf{S}$  and a postcondition  $\Psi = \psi(Z)$  (assertion with logical variable  $Z$ ) the weakest liberal precondition is the set of states

$$\text{wlp}(\mathbf{S}, \Psi) \stackrel{\text{def}}{=} \{ \sigma \mid \forall \tau (\sigma \xrightarrow{\mathbf{S}} \tau \implies \tau \in \Psi) \}. \quad (1)$$

The weakest precondition of  $\mathbf{S}$  and  $\Psi$  is the set of states

$$\text{wp}(\mathbf{S}, \Psi) \stackrel{\text{def}}{=} \{ \sigma \mid \sigma \in \text{wlp}(\mathbf{S}, \Psi) \wedge \begin{array}{l} \text{all computational paths of } \mathbf{S} \text{ terminate} \\ \text{when executed in initial state } \sigma \end{array} \}. \quad (2)$$

Intuitively the difference between the weakest precondition and the weakest liberal precondition concerns the termination of  $\mathbf{S}$ : if we execute  $\mathbf{S}$  in an initial state  $\sigma \in \text{wlp}(\mathbf{S}, \Psi)$  then  $\mathbf{S}$  need not terminate. On the other hand, if we execute  $\mathbf{S}$  in an initial state  $\sigma \in \text{wp}(\mathbf{S}, \Psi)$  then  $\mathbf{S}$  is certain to terminate.

Informally a hoare triple  $\langle \phi \rangle \mathbf{S} \langle \psi \rangle$  is:

- partially correct if in any state  $\sigma \in \phi$ , execution of the program  $\mathbf{S}$  results in a state  $\tau \in \psi$ , or  $\mathbf{S}$  does not terminate.
- totally correct if it is partially correct and  $\mathbf{S}$  terminates.

Note that for any program  $\mathbf{S}$  and postcondition  $\psi$  the triple  $\langle \text{wp}(\mathbf{S}, \psi) \rangle \mathbf{S} \langle \psi \rangle$  is totally correct and the triple  $\langle \text{wlp}(\mathbf{S}, \psi) \rangle \mathbf{S} \langle \psi \rangle$  is partially correct.

Proving the correctness of a program amounts to proving correctness of all the functions in the program. To ensure modularity this can be done by considering one function body at a time. Suppose one wants to prove a triple  $\langle \phi \rangle \mathbf{S} \langle \psi \rangle$ , where  $\mathbf{S}$  is the body of an arbitrary function. We can reason about the program backwards by computing the weakest precondition  $\text{wp}(\mathbf{S}, \psi)$ . The proof burden is then reduced to one implication  $(\phi \implies \text{wp}(\mathbf{S}, \psi))$  for each function.<sup>3</sup> In the following sections, we will show weakest preconditions for each program construct.

---

<sup>3</sup>We do not compute the weakest precondition for while-statements but instead ask the programmer for an invariant. Proving that the given invariant is indeed an invariant and that it is strong enough gives rise to additional implications. See section 3.5 for more detail.

### 3.2 Consequence rule

In the following sections, Hoare-rules for program constructs are considered. However, if we have proved the triple  $\llbracket \phi \rrbracket \mathbf{S} \llbracket \psi \rrbracket$  is valid, then we should be able to also prove  $\llbracket \phi' \rrbracket \mathbf{S} \llbracket \psi' \rrbracket$  if  $\phi' \longrightarrow \phi$  and  $\psi \longrightarrow \psi'$ . Hoare has given the rule

$$\frac{\phi' \longrightarrow \phi \quad \llbracket \phi \rrbracket \mathbf{S} \llbracket \psi \rrbracket \quad \psi \longrightarrow \psi'}{\llbracket \phi' \rrbracket \mathbf{S} \llbracket \psi' \rrbracket} \text{Cons} \quad (3)$$

This rule is sound, but in the presence of logical variables this rule is not complete. Let  $Z$  be a logical variable and  $x$  a program variable and consider the triple  $\llbracket x = Z \rrbracket \mathbf{S} \llbracket x = Z \rrbracket$ . With rule 3 it's not possible to derive, for example,  $\llbracket x = Z + 1 \rrbracket \mathbf{S} \llbracket x = Z + 1 \rrbracket$ .

Based on the work of Hoare [19] and Olderog [26], Kleymann [28] has presented a new consequence rule (which does lead to a complete proof calculus):

$$\frac{\llbracket \phi \rrbracket \mathbf{S} \llbracket \psi \rrbracket}{\llbracket \phi' \rrbracket \mathbf{S} \llbracket \psi' \rrbracket} \text{Kley-Cons} \quad (4)$$

$$\text{if } \forall Z \forall \sigma (\phi'(Z, \sigma) \longrightarrow \forall \tau \exists Z_1 (\phi(Z_1, \sigma) \wedge (\psi(Z_1, \tau) \longrightarrow \psi'(Z, \tau))))$$

Note that the side condition of Rule 4 is a verification condition that cannot be expressed in Mist directly. Because of quantification over states ( $\forall Z, \forall \sigma, \forall \tau$ ), it can not even be expressed in first-order logic, and so neither in the language of our theorem prover KeY (which is described in more detail in Section 4.5.2). However, quantification over states can be replaced by quantification over the concrete variables of the program, which are known at the static analysis phase of compilation. This makes the formula expressible in KeY. If we move all universal quantification to the beginning of the formula and eliminate the existential quantifier ( $\exists Z_1$ ) by restricting program preconditions as will be described in Section 3.6.1, the condition is also expressible in Mist.

### 3.3 Assignment

An assignment is the only way to modify the state of the program. Syntactically, Mist assignments have the form:

$x \leftarrow E;$

where  $x$  is a program variable and the expression  $E$  (which does not contain side-effects; see Section 3 introduction) is evaluated in the state  $\sigma$  right before the assignment. In the resulting state the new value for  $x$  replaces the prior value of the variable. In Hoare logic one can reason about assignments with the following axiom:

$$\overline{\llbracket \phi[E/x] \rrbracket x \leftarrow E; \llbracket \phi \rrbracket} \text{Assign} \quad (5)$$

Since an assignment itself cannot introduce non-termination, this rule is valid for partial *and* total correctness. Computing the weakest (liberal) precondition of an assignment can be done mechanically: only a substitution on  $\phi$  is necessary. From the given assignment rule we can easily deduce

$$\text{wp}(x \leftarrow E; , \phi) = \text{wlp}(x \leftarrow E; , \phi) = \phi[E/x] \quad (6)$$

### 3.4 If-statement

Conditional statements can be used to influence the control flow of a program. Mist uses the if-statement with an optional else clause:

```

if (B)
  S1
else
  S2

```

The condition B, an arbitrary boolean expression, must not contain side-effects. Semantically, the if-statement tests if B is true in the initial state  $\sigma$ . If B is true, statement S1 is executed (in the initial state  $\sigma$ ) and the resulting state of the if-statement is the resulting state of S1. If B is false, S2 is executed to form the resulting state (if the else clause is not specified,  $S2 \stackrel{\text{def}}{=} \{\}$  (no operation)). This is reflected by the Hoare-rule:

$$\frac{(\phi_1) \text{ S1 } (\psi) \quad (\phi_2) \text{ S2 } (\psi)}{((B \implies \phi_1) \wedge (\neg B \implies \phi_2)) \text{ if (B) S1 else S2 } (\psi)} \text{ If} \quad (7)$$

The If-statement itself cannot introduce non-termination thus the rule is valid for both partial and total correctness.

The weakest (liberal) precondition of the if-statement can be easily inferred from the rule if we know what  $\phi_1$  and  $\phi_2$  are.  $\phi_1$  must be a precondition of S1, given  $\psi$ . In order to find the *weakest* precondition of the if-statement,  $\phi_1$  must be also be the *weakest* precondition ( $\text{wp}(\text{S1}, \psi)$ ). For the weakest *liberal* precondition,  $\phi_1 = \text{wlp}(\text{S1}, \psi)$ . Analogously, we can find  $\phi_2$ . The weakest precondition for the if-statement is now easy to deduce:

$$\begin{aligned} \text{wp}(\text{if (B) S1 else S2}, \psi) = \\ (B \implies \text{wp}(\text{S1}, \psi)) \wedge (\neg B \implies \text{wp}(\text{S2}, \psi)) \end{aligned} \quad (8)$$

And also the weakest liberal precondition:

$$\begin{aligned} \text{wlp}(\text{if (B) S1 else S2}, \psi) = \\ (B \implies \text{wlp}(\text{S1}, \psi)) \wedge (\neg B \implies \text{wlp}(\text{S2}, \psi)) \end{aligned} \quad (9)$$

### 3.5 While-statement

In addition to if-statements, while-statements can also be used to influence control flow of a program. In Mist, a programmer can specify an optional invariant and bound for the while as follows:

```
while (B) invariant { @ I; } bound (E) do S
```

The while-statement repeatedly evaluates the boolean guard  $B$  and executes  $S$  if  $B$  is true. If  $B$  is false (after 0 or more executions of  $S$ ), the while-loop terminates.

Since it's possible that  $B$  never becomes false, while-loops can potentially introduce non-termination in a program. A common approach to proving termination of a loop is to show that there is an integer expression which strictly decreases every iteration of the loop, but always remains non-negative. Such expressions are called *variants* or *bounds*, and can be specified in Mist by the programmer with the optional `bound`-keyword. Finding such bounds automatically for arbitrary while-loops has been shown to be impossible [2].

In addition to a bound, the programmer can also specify a loop *invariant*. The invariant  $I$  is a boolean expression that must evaluate to true before the loop, and after each iteration of the loop sub-statement  $S$ . Since  $I$  will be true after any number of executions of  $S$ , we can conclude that  $I$  must be true after execution of the loop. This leads to the following two Hoare-rules (we distinguish between partial and total correctness for loops):

$$\frac{\langle B \wedge I \rangle S \langle I \rangle}{\langle I \rangle \text{ while } (B) \text{ invariant } \{ @ I; \} \text{ do } S \langle \neg B \wedge I \rangle} \text{Par-While} \quad (10)$$

In order to show that  $E$  decreases each time  $S$  is executed, we use a logical variable  $E_0$  to remember the value of the bound  $E$  before execution of  $S$ :

$$\frac{\langle B \wedge I \wedge 0 < E = E_0 \rangle S \langle I \wedge 0 \leq E < E_0 \rangle}{\langle I \wedge 0 \leq E \rangle \text{ while } (B) \text{ invariant } \{ @ I; \} \text{ bound } (E) \text{ do } S \langle \neg B \wedge I \rangle} \text{Tot-While} \quad (11)$$

The weakest precondition of a while can be expressed by a recursive formula [12], but verifying the loop becomes difficult because complex side-conditions of the Consequence rule 4 must be proven. An alternative to calculating the weakest precondition for a loop is to ask the programmer for the invariant  $I$ . Suppose we want to prove total correctness of  $\langle \phi \rangle \text{ while } (B) \text{ do } S \langle \psi \rangle$  (establishing partial correctness is analogous). First applying the loop rule 11 and next applying the consequence rule 4 results in the following proof obligations:

1.  $I$  is an invariant and  $E$  is a bound:  $(B \wedge I \wedge 0 < E = E_0) \implies \text{wp}(S, I \wedge 0 \leq E < E_0)$
2.  $\phi$  is strong enough:  $\phi \implies (I \wedge 0 \leq E)$
3.  $\psi$  is weak enough:  $(\neg B \wedge I) \implies \psi$

Establishing correctness of  $\langle \phi \rangle \text{ while } (B) \text{ do } S \langle \psi \rangle$  then amounts to a proof of validity of those three implications. Finding suitable invariants automatically has been shown to be non-trivial, but possible for special cases [25].

### 3.6 Procedure Call

A procedure, sometimes also called a subprogram, consists of a sequence of statements (the *body*), and a *signature* (the name of the procedure and parameter names and types). We will consider parameterless procedures in this section. A procedure call is a statement that executes the statements of the given procedure. In Mist a procedure call to a procedure  $\mathbf{f}$  can be denoted by:

$\mathbf{f} ();$

A procedure can also call itself. Such a procedure is called a *recursive procedure*. Recursive procedures are, like while-loops, a source of non-termination. A procedure can potentially call itself indefinitely. In the original Hoare-calculus [18] procedures were not considered. We will use the results of Kleymann [28] in the remainder of this section.

Let  $\mathbf{f}$  be a recursive procedure with body  $\mathbf{S}$ . In order to prove correctness of a triple  $\{\phi\} \mathbf{f} () \{\psi\}$ , we may simply assume  $\{\phi\} \mathbf{f} () \{\psi\}$  and show that  $\{\phi\} \mathbf{S} \{\psi\}$  holds under this assumption.

This rule leads to the notion of a context: provability of a Hoare-triple must now be considered with respect to an additional set of assumed Hoare-triples (the context). We will use the notation  $\{\{\phi\} \mathbf{S} \{\psi\}\} \vdash \{\phi'\} \mathbf{S} \{\psi'\}$  if we can prove correctness of  $\{\phi'\} \mathbf{S} \{\psi'\}$  under the assumption of  $\{\phi\} \mathbf{S} \{\psi\}$ . In this case, the context is  $\{\{\phi\} \mathbf{S} \{\psi\}\}$ . A context contains 0 or more Hoare-triples. Proving a Hoare-triple in the empty context (a context with 0 Hoare-triples), can be seen as proving correctness of the Hoare-triple under the assumption *true*. All Hoare-rules given in previous sections are valid in any context.<sup>4</sup> For more detail on contexts, refer to [28].

Correctness for a call to a procedure  $\mathbf{f}$  with body  $\mathbf{S}$  can now be expressed with the rule:

$$\frac{C \vdash \{\phi\} \mathbf{S} \{\psi\}}{\{\phi\} \mathbf{f} () \{\psi\}} \text{ Call} \quad (12)$$

In the case of partial correctness for a recursive procedure  $\mathbf{f}$ , the context  $C = \{\{\phi\} \mathbf{f} () \{\psi\}\}$ . The rule expresses total correctness for calls to non-recursive procedures. In that case, the context  $C = \emptyset$ . In order to prove total correctness of a recursive procedure call, the programmer has to supply a bound (refer to section 3.5 for more details) for the procedure, as was the case with while loops. Let  $\mathbf{E}$  be the bound of  $\mathbf{f}$ . Total correctness is then expressed by:

$$\frac{\{\{\phi \wedge 0 \leq E < E_0\} \mathbf{f} () \{\psi\}\} \vdash \{\phi \wedge 0 < E = E_0\} \mathbf{S} \{\psi\}}{\{\phi \wedge 0 \leq E\} \mathbf{f} () \{\psi\}} \text{ Tot-Rec-Call} \quad (13)$$

Kleymann has provided weakest (liberal) preconditions of procedure calls. In the next section, we will only show how to calculate the weakest **liberal** precondition of a procedure call as the weakest precondition can be derived similarly.

<sup>4</sup>If a Hoare-triple without a context is given, it must be valid in any context.

### 3.6.1 Weakest liberal precondition

Given a triple  $\llbracket \phi' \rrbracket S \llbracket \psi' \rrbracket$  and a postcondition  $\psi$ , we want to calculate the weakest liberal precondition  $\phi = \text{wlp}(S, \psi)$ . Morris [22] has proposed

$$\phi(Z, \sigma) \stackrel{\text{def}}{=} \forall \tau (\forall Z_1 (\phi'(Z_1, \sigma) \implies \psi'(Z_1, \tau)) \implies \psi(Z, \tau)) \quad (14)$$

This precondition cannot be expressed directly in our assertion language, due to the presence of the existential quantifier over the logical variables.

In order to eliminate the existential quantifier we restrict the precondition  $\phi'(Z_1, \sigma)$  syntactically to the form  $(\phi''(Z_1, \sigma) \wedge Z_1 = E)$  for some formula  $\phi''$  and expression  $E$  (which may depend on constants and program variables in the initial state  $\sigma$ ). It's easy to see that the equation  $\forall Z_1 (\phi'(Z_1, \sigma) \iff \phi''(Z_1, \sigma) \wedge Z_1 = E)$  then holds. We show that with the syntactic restriction Equation 14 is equivalent to:

$$\phi(Z, \sigma) \stackrel{\text{def}}{=} \forall \tau ((\phi''(Z_1, \sigma)[E/Z_1] \implies \psi'(Z_1, \tau)[E/Z_1]) \implies \psi(Z, \tau)) \quad (15)$$

It's important to realize the expression  $E$  is evaluated in state  $\sigma$  (since  $\phi$  is), even in the formula  $\psi'(Z_1, \tau)[E/Z_1]$ : this is achieved by substituting  $E$  for  $Z_1$  *after* evaluating the program variables appearing in  $\psi'$  in state  $\tau$ .

We will now prove  $\phi(Z, \sigma)$  is indeed a precondition:

$$\frac{\llbracket \phi'(Z_1, \sigma) \rrbracket S \llbracket \psi'(Z_1, \tau) \rrbracket}{\frac{\llbracket \forall \tau (\forall Z_1 (\phi'(Z_1, \sigma) \implies \psi'(Z_1, \tau)) \implies \psi(Z, \tau)) \rrbracket S \llbracket \psi(Z, \tau) \rrbracket}{\llbracket \forall \tau ((\phi''(Z_1, \sigma)[E/Z_1] \implies \psi'(Z_1, \tau)[E/Z_1]) \implies \psi(Z, \tau)) \rrbracket S \llbracket \psi(Z, \tau) \rrbracket} \text{Morris}} \text{Cons}(\ast) \quad (16)$$

( $\ast$ ): Trivially  $\psi(Z, \tau) \implies \psi(Z, \tau)$ , so it suffices to show

$$\begin{aligned} & \forall \tau ((\phi''(Z_1, \sigma)[E/Z_1] \implies \psi'(Z_1, \tau)[E/Z_1]) \implies \psi(Z, \tau)) \\ \vdash & \quad \forall \tau (\forall Z_1 (\phi'(Z_1, \sigma) \implies \psi'(Z_1, \tau)) \implies \psi(Z, \tau)) \end{aligned} \quad (17)$$

We proceed by using a proof-outline for reasons of space and clarity.

1	$\forall\tau((\phi''(Z_1, \sigma)[E/Z_1] \implies \psi'(Z_1, \tau)[E/Z_1]) \implies \psi(Z, \tau))$	premise			
2	$\forall Z_1(\phi'(Z_1, \sigma) \iff \phi''(Z_1, \sigma) \wedge Z_1 = E)$	premise			
3	$\phi'(Z_1, \sigma)[E/Z_1] \iff \phi''(Z_1, \sigma)[E/Z_1] \wedge E = E$	$\forall E, 2$			
4	<table style="border-collapse: collapse; width: 100%;"> <tr> <td style="width: 5%; text-align: right; vertical-align: top;">a</td> <td style="width: 15%; border-left: 1px solid black; padding-left: 5px; vertical-align: top;"><math>(\phi''(Z_1, \sigma)[E/Z_1] \implies \psi'(Z_1, a)[E/Z_1]) \implies \psi(Z, a)</math></td> <td style="width: 10%; vertical-align: top;"><math>\forall E, 1</math></td> </tr> </table>	a	$(\phi''(Z_1, \sigma)[E/Z_1] \implies \psi'(Z_1, a)[E/Z_1]) \implies \psi(Z, a)$	$\forall E, 1$	$\forall E, 1$
a	$(\phi''(Z_1, \sigma)[E/Z_1] \implies \psi'(Z_1, a)[E/Z_1]) \implies \psi(Z, a)$	$\forall E, 1$			
5	<table style="border-collapse: collapse; width: 100%;"> <tr> <td style="width: 5%;"></td> <td style="width: 15%; border-left: 1px solid black; padding-left: 5px; vertical-align: top;"><math>\forall Z_1(\phi'(Z_1, \sigma) \implies \psi'(Z_1, a))</math></td> <td style="width: 10%; vertical-align: top;">assumption</td> </tr> </table>		$\forall Z_1(\phi'(Z_1, \sigma) \implies \psi'(Z_1, a))$	assumption	assumption
	$\forall Z_1(\phi'(Z_1, \sigma) \implies \psi'(Z_1, a))$	assumption			
6	<table style="border-collapse: collapse; width: 100%;"> <tr> <td style="width: 5%;"></td> <td style="width: 15%; border-left: 1px solid black; padding-left: 5px; vertical-align: top;"><math>\phi'(Z_1, \sigma)[E/Z_1] \implies \psi'(Z_1, a)[E/Z_1]</math></td> <td style="width: 10%; vertical-align: top;"><math>\forall E, 5</math></td> </tr> </table>		$\phi'(Z_1, \sigma)[E/Z_1] \implies \psi'(Z_1, a)[E/Z_1]$	$\forall E, 5$	$\forall E, 5$
	$\phi'(Z_1, \sigma)[E/Z_1] \implies \psi'(Z_1, a)[E/Z_1]$	$\forall E, 5$			
7	<table style="border-collapse: collapse; width: 100%;"> <tr> <td style="width: 5%;"></td> <td style="width: 15%; border-left: 1px solid black; padding-left: 5px; vertical-align: top;"><math>\phi''(Z_1, \sigma)[E/Z_1]</math></td> <td style="width: 10%; vertical-align: top;">assumption</td> </tr> </table>		$\phi''(Z_1, \sigma)[E/Z_1]$	assumption	assumption
	$\phi''(Z_1, \sigma)[E/Z_1]$	assumption			
8	<table style="border-collapse: collapse; width: 100%;"> <tr> <td style="width: 5%;"></td> <td style="width: 15%; border-left: 1px solid black; padding-left: 5px; vertical-align: top;"><math>E = E</math></td> <td style="width: 10%; vertical-align: top;"><math>=I</math></td> </tr> </table>		$E = E$	$=I$	$=I$
	$E = E$	$=I$			
9	<table style="border-collapse: collapse; width: 100%;"> <tr> <td style="width: 5%;"></td> <td style="width: 15%; border-left: 1px solid black; padding-left: 5px; vertical-align: top;"><math>\phi''(Z_1, \sigma)[E/Z_1] \wedge E = E</math></td> <td style="width: 10%; vertical-align: top;"><math>\wedge I, 7, 8</math></td> </tr> </table>		$\phi''(Z_1, \sigma)[E/Z_1] \wedge E = E$	$\wedge I, 7, 8$	$\wedge I, 7, 8$
	$\phi''(Z_1, \sigma)[E/Z_1] \wedge E = E$	$\wedge I, 7, 8$			
10	<table style="border-collapse: collapse; width: 100%;"> <tr> <td style="width: 5%;"></td> <td style="width: 15%; border-left: 1px solid black; padding-left: 5px; vertical-align: top;"><math>\phi'(Z_1, \sigma)[E/Z_1]</math></td> <td style="width: 10%; vertical-align: top;"><math>\implies E, 3, 9</math></td> </tr> </table>		$\phi'(Z_1, \sigma)[E/Z_1]$	$\implies E, 3, 9$	$\implies E, 3, 9$
	$\phi'(Z_1, \sigma)[E/Z_1]$	$\implies E, 3, 9$			
11	<table style="border-collapse: collapse; width: 100%;"> <tr> <td style="width: 5%;"></td> <td style="width: 15%; border-left: 1px solid black; padding-left: 5px; vertical-align: top;"><math>\psi'(Z_1, a)[E/Z_1]</math></td> <td style="width: 10%; vertical-align: top;"><math>\implies E, 6, 10</math></td> </tr> </table>		$\psi'(Z_1, a)[E/Z_1]$	$\implies E, 6, 10$	$\implies E, 6, 10$
	$\psi'(Z_1, a)[E/Z_1]$	$\implies E, 6, 10$			
12	$\phi''(Z_1, \sigma)[E/Z_1] \implies \psi'(Z_1, a)[E/Z_1]$	$\implies I, 7-11$			
13	$\psi(Z, a)$	$\implies E, 4, 12$			
14	$\forall Z_1(\phi'(Z_1, \sigma) \implies \psi'(Z_1, a)) \implies \psi(Z, a)$	$\implies I, 5-13$			
15	$\forall\tau(\forall Z_1(\phi'(Z_1, \sigma) \implies \psi'(Z_1, \tau)) \implies \psi(Z, \tau))$	$\forall I, 4-14$			

This shows Equation 15 is sound (in that our precondition is indeed a precondition), but it also shows it's at least as strong as Equation 14 (and thus may possibly not be the *weakest* liberal precondition). We will show next the converse is also true, i.e. that Equation 14 implies our quantifier-free rule 15.

1	$\forall\tau(\forall Z_1(\phi'(Z_1, \sigma) \implies \psi'(Z_1, \tau)) \implies \psi(Z, \tau))$	premise
2	$\forall Z_1(\phi'(Z_1, \sigma) \iff \phi''(Z_1, \sigma) \wedge Z_1 = E)$	premise
3	$\phi'(Z_1, \sigma)[E/Z_1] \iff \phi''(Z_1, \sigma)[E/Z_1] \wedge E = E$	$\forall E, 2$
4	$a \mid (\forall Z_1(\phi'(Z_1, \sigma) \implies \psi'(Z_1, a)) \implies \psi(Z, a))$	$\forall E, 1$
5	$(\phi'(Z_1, \sigma)[E/Z_1] \implies \psi'(Z_1, a)[E/Z_1]) \implies \psi(Z, a)$	$\forall E, 4$
6	$\mid \frac{\phi''(Z_1, \sigma)[E/Z_1] \implies \psi'(Z_1, a)[E/Z_1]}{\phantom{\phi''(Z_1, \sigma)[E/Z_1] \implies \psi'(Z_1, a)[E/Z_1]}}$	assumption
7	$\mid \mid \frac{\phi'(Z_1, \sigma)[E/Z_1]}{\phantom{\phi'(Z_1, \sigma)[E/Z_1]}}$	assumption
8	$\mid \mid \frac{\phi''(Z_1, \sigma)[E/Z_1] \wedge E = E}{\phantom{\phi''(Z_1, \sigma)[E/Z_1] \wedge E = E}}$	$\implies E, 3, 7$
9	$\mid \mid \frac{\phi''(Z_1, \sigma)[E/Z_1]}{\phantom{\phi''(Z_1, \sigma)[E/Z_1]}}$	$\wedge E, 8$
10	$\mid \mid \frac{\psi'(Z_1, a)[E/Z_1]}{\phantom{\psi'(Z_1, a)[E/Z_1]}}$	$\implies E, 6, 9$
11	$\mid \frac{\phi'(Z_1, \sigma)[E/Z_1] \implies \psi'(Z_1, a)[E/Z_1]}{\phantom{\phi'(Z_1, \sigma)[E/Z_1] \implies \psi'(Z_1, a)[E/Z_1]}}$	$\implies I, 7-10$
12	$\mid \frac{\psi(Z, a)}{\phantom{\psi(Z, a)}}$	$\implies E, 5, 11$
13	$\mid \frac{(\phi''(Z_1, \sigma)[E/Z_1] \implies \psi'(Z_1, a)[E/Z_1]) \implies \psi(Z, a)}{\phantom{(\phi''(Z_1, \sigma)[E/Z_1] \implies \psi'(Z_1, a)[E/Z_1]) \implies \psi(Z, a)}}$	$\implies I, 6-12$
14	$\forall\tau((\phi''(Z_1, \sigma)[E/Z_1] \implies \psi'(Z_1, \tau)[E/Z_1]) \implies \psi(Z, \tau))$	$\forall I, 4-13$

This establishes that our quantifier-free rule 15 is in fact the *weakest* liberal precondition:

$$\forall\tau((\phi''(Z_1, \sigma)[E/Z_1] \implies \psi'(Z_1, \tau)[E/Z_1]) \implies \psi(Z, \tau)) = \text{wlp}(S, \psi(Z, \tau)). \quad (18)$$

### 3.7 Sequential Composition

In previous sections we have seen assignments, if-statements, while-statements and procedure calls, but no way to combine these constructs in a program. Sequential composition of two statements **S1 S2** means we first execute **S1**, and in the resulting state we execute **S2** to form the final state. This allows the programmer to form programs with more than one statement. The Hoare-rule for sequential composition is given by:

$$\frac{(\phi) \mathbf{S1} (\chi) \quad (\chi) \mathbf{S2} (\psi)}{(\phi) \mathbf{S1} \mathbf{S2} (\psi)} \text{Seq} \quad (19)$$

This rule is valid for partial and total correctness. The weakest precondition  $\text{wp}(\mathbf{S1} \mathbf{S2}, \psi)$  can be computed by first computing the weakest precondition  $\text{wp}(\mathbf{S2}, \psi)$  and then using this as postcondition for **S1**, hence:

$$\text{wp}(\mathbf{S1} \mathbf{S2}, \psi) = \text{wp}(\mathbf{S1}, \text{wp}(\mathbf{S2}, \psi)) \quad (20)$$

The weakest liberal precondition can be computed completely analogously:

$$\text{wlp}(S1 \ S2, \psi) = \text{wlp}(S1, \text{wlp}(S2, \psi)) \quad (21)$$

## 4 The Compiler

This section describes the current Mist compiler. Section 4.1 describes the basic facts about the compiler. Section 4.2 explains the passes of compilation. Section 4.3 shows the way in which the Mist compiler keeps track of symbols. The Mist runtime environment is described briefly in Section 4.4. Finally, in Section 4.5, the integration of the compiler and the theorem prover is explained.

### 4.1 Fundamentals

The Mist compiler takes as input a single Mist source file. It finds and reports any syntactic or semantic errors in the source code. If there are none, it generates C++ source code that, in turn, can be compiled to a binary using your favorite C++ compiler.

Since the focus of the project is on the design of the language itself, there has been no need for a more efficient intermediate language. But in the future, LLVM [24] may be used instead.

A Mist program is run in a managed environment. Some errors can at this moment only be reported at runtime. But the errors that can be caught at compile-time (a set that will continue to grow) will not require run-time checks. The compiler will simply prove that certain bugs do not exist.

To build the compiler, several tools are necessary:

- The compiler is written in C++, so a C++ compiler is required.
- The lexical analyzer is generated with Flex.
- The parser is generated with GNU Bison.
- The code makes much use of the Boost library to improve readability and maintainability of the code.
- The compiler is automatically built using GNU Make.
- Unit-testing is performed by the Boost test library.
- API documentation is generated with Doxygen.
- At present, Linux is required as a building environment, since certain tools (such as Flex) have not been ported to other operating systems for a long time.

If all these tools are present, one only has to run `make` to build the compiler and run unit-tests.

## 4.2 Compilation Passes

The compiler receives a Mist source-file as input. It feeds this input through the lexical analyzer, which will tokenize it. The token-stream is interpreted by the parser, which uses the grammar given in Appendix B to transform the tokens into an abstract syntax tree.

The abstract syntax tree nodes act as visited elements in the Visitor pattern [15]. The concrete visitor classes that perform operations on this tree include the echo-visitor (reproduce the original Mist code), the symbol-visitor (builds the symbol-table), the type-visitor (finds the type of expressions), resolve-visitor (resolve references) and the proof-visitor (attempt to find proof of correctness of a given function).

The information some compilers calculate during subsequent passes (symbols, scopes, resolved references, types, side-effects, access-rights, etc.), are in the Mist compiler lazy properties of the relevant AST nodes. For example, when the type of an expression is required, you may just query the expression node and it will, only once, compute its own type by querying its subexpressions, and so on.

This has several advantages over a finite amount of discrete compiler passes:

- Mist has some features that require an approach such as this. It has function overloading, which means the compiler needs to know the types of the actual parameters before it can be sure which specific function is referenced. On the other hand, before it can know the type of a function-call, it has to first resolve the function reference. That's non-trivial using a fixed number of compiler passes, because the programmer can make his expression arbitrarily complicated. Mist will also feature the `auto` keyword to get automatic type deduction for variable declarations. This requires that the compiler know the type of the initialization value before it can add the variable to the symbol table.
- If some information is never needed, it need never be computed. On the other hand, if it is needed often, it is never computed more than once.
- The compiler code is much simpler this way. The developer does not have to worry about the order of the compiler passes. Information is always available.

It is of course possible to get a cyclic data dependency this way. But this is always a mistake, either of the compiler (e.g. it forgot to report an error when a variable is referenced before it is declared, possibly resulting in two auto-typed variables initializing each other) or of its user (e.g. two functions, whose textual order is irrelevant, use each other in automatic type deduction).

To catch both kinds of mistake, the automatically computed properties carry a 'computing' flag. The first time their value is queried, they turn it on before starting computation. If it was already on, that means there was a cyclic dependency and the compiler will either report this to the developer by throwing an exception or to the user by reporting an appropriate error-message.

Optimizing the intermediate code and generating the executable still both use their own compiler pass.

### 4.3 The Symbol-tree

The symbol-table in the Mist compiler is in actuality a ‘symbol-tree’. This tree reflects both the symbols and the scoping of the entire program. This makes it relatively simple to resolve references. One only has to walk up the tree and use the first properly named symbol one finds. This automatically works for nested functions and contracts as well.

A function introduces a rather complex symbol-subtree. It is described by its root node ( $R$ ), the scope which houses the formal parameter symbols, the logical variable symbols and the template type symbols.  $R$  is also the direct parent of the precondition scope. Any reference in the precondition may refer to any of these symbols, or shallower ones.

The last child of  $R$  is a dummy symbol ( $D_1$ ), used only so the structure of the tree directly reflects symbol visibility.  $D_1$  has as its children the `result` variable symbol and the scope of the postcondition. This means that the postcondition can see the result variable, but the precondition cannot.

The last child of  $D_1$  is a second dummy symbol ( $D_2$ ), which is the location of the function implementation, including its local variables. This means that the function body can see local variables, but the pre- and postcondition cannot.

Each symbol node in the tree has several properties related to the structure of the tree. A node can be either transparent or opaque (see Section 2.2.1). If it is transparent, a reference higher up the tree can reference a symbol inside. As of yet, every node in the tree is opaque, however. But the reference resolving algorithm is already capable of understanding transparent scopes, in anticipation of classes.

A symbol node may introduce a scope, or it may not. A scope, in this context, means that it would be an explicit part of any absolute path (see Section 2.2.3) that includes it. Also, between the depth of two scopes, no two symbols may have the same name, except by function overloading (in that case, no two symbols may have the same function signature). The only symbol-tree nodes that do not introduce a scope are variables, non-class types and dummy nodes ( $D_1$  and  $D_2$ ).

Of course, the symbols of the tree contain information traditionally expected in a symbol-table, such as type, access-rights, side-effect reach, etc.

Note that this tree-structure is already prepared for many future alterations of the language. If and when we decide to allow arbitrary code in pre- and postconditions, this will already be handled gracefully. `pre` and `post` introduce a scope so they can contain local variable declarations, even other nested functions, without issue. Absolute paths to symbols inside are no problem. Even if multiple `pre` sections are allowed to exist inside a single function scope, the scope `pre` itself is useless to reference from outside, and when referenced from inside, there is no ambiguity as to the meaning.

## 4.4 Mist Runtime Environment

Almost all of the Mist runtime environment, the definition of basic types and basic functions and operators, is handled in the Mist language itself, but with the help of special constructs that specify translation to the execution language (C++) and the theorem prover language (KeY).

This demonstrates that the compiler itself is very flexible, and that most Mist constructions can be implemented in Mist itself by the programmer.

For example, Mist features function overloading, and through extension of this feature, operator overloading. Every operator is associated with a function-name. Overloading that function also overloads the associated operator. The Mist runtime simply uses this feature to implement the basic operations of the basic types. This makes the code-generation phase of the compiler especially generic and clean, since it does not need to contain the implementations for each and every operation.

## 4.5 Integration with KeY

As explained in Section 3.1, using weakest precondition logic, the burden of proof of correctness can be reduced to a set of implications. After the wp-visitor has been used, these implications are all found, and all that is left to do is attempt to prove them.

However, proving satisfiability of first-order logic formula's is undecidable. The Mist language does not allow arbitrary quantification, so its boolean expressions are a strict subset of first-order logic. Even so, the resulting proof obligations are at least non-trivial to prove. We did not have a reason to invent that wheel again, so we went looking for an existing theorem prover.

### 4.5.1 Prolog

The first possibility we explored was Prolog. We tried several Prolog distributions, but we discovered that their theorem proving capabilities were wholly inadequate.

By default, Prolog does not even know about simple arithmetic. We tried several libraries for arithmetic and constraint programming. We attempted to prove such simple tautologies as  $(x > 2 \implies x > 1)$ , but Prolog was unable to verify their truth, instead offering a different but more complicated way to describe the same formula.

We soon concluded that Prolog was not the right tool for the job.

### 4.5.2 KeY

Through the authors' association with the HATS project<sup>5</sup>, we came into contact with Dr. Wolfgang Ahrendt and the KeY project [5]. The KeY project involves

---

<sup>5</sup>Highly Adaptable and Trustworthy Software using Formal Methods; an Integrated Project supported by the 7th Framework Programme of the EC within the FET (Future and Emerging Technologies) scheme

a sophisticated theorem prover for dynamic logic. It may be used to verify Java code, but its theorem prover seemed very well-suited for proving our implications as well.

KeY uses an internal rewriting engine to, step by step, simplify the given formula to find out if it's a tautology. If it cannot prove that it is a tautology, it stops and specifies the proof-goals left open. Still proving those would be sufficient for proving the original formula, but KeY cannot reduce it further.



*Note that KeY not finding a proof does not necessarily mean that a proof does not exist. KeY may have exhausted its allocated resources, or it may simply not be smart enough. KeY can only give a 'yes' or a 'maybe', and that's why the Mist compiler can do no better.*

While suitable in theory, KeY proved awkward to interface with external applications. KeY-tool is a stand-alone application with a graphical user interface. While its core is exactly what the Mist compiler requires, it was hard to access it without requiring user interaction. Thankfully, a command-line tool was also available.

Nonetheless, the interface remains clumsy. The Mist compiler translates the remaining proof obligations to the KeY language. It writes them to problem files in the system `/tmp` directory in a format that KeY expects. The compiler process then forks the command-line solver of KeY and points it to the problem file. KeY will output a proof file to `/tmp` and reports the proof status with its exit code. The Mist compiler uses only this exit code to find out if a proof was found.

While this approach has the benefit of working, it is far from ideal. The proof for most programs written in Mist requires multiple implications to be solved. To be precise: two for each while-loop and one for each function. Right now, each implication requires starting a separate KeY process. Each process can take about 10 seconds to load, while in theory this only has to happen once. These 10 seconds are used to load the rewriting rules. The proof itself usually takes less than a second.

Also, the proof file that KeY outputs is not meant to be interpreted by any external application. The output format is not guaranteed to remain stable as KeY evolves. So if a proof fails, the Mist compiler can currently not report the open goals to the programmer. The programmer will have to load the proof file with the KeY graphical interface, which of course cannot report the open goals in the Mist language.



*The authors are in further contact with the members of the KeY project, to possibly collaborate on a KeY API. It is possible all of these problems (and more) will be solved, and KeY will become even more suitable as a theorem proving engine for Mist.*

To translate the implications to the KeY language, the Mist runtime environment specifies a modular KeY translation for each primitive operation. The following table lists these translations:

Mist Operator	Mist Function	KeY Notation
a <==> b	logical_equal(bool a, bool b)	a <-> b
a ==> b	implied(bool a, bool b)	b -> a
a <== b	implies(bool a, bool b)	a -> b
a or b	disjunct(bool a, bool b)	a   b
a and b	conjunct(bool a, bool b)	a & b
a <>= b	comparable(int a, int b)	true
a <>= b	comparable(bool a, bool b)	a <-> b
a <= b	smaller_equal(int a, int b)	a <= b
a <= b	smaller_equal(bool a, bool b)	a <-> b
a <> b	inequal(int a, int b)	a != b
a <> b	inequal(bool a, bool b)	false
a >= b	greater_equal(int a, int b)	a >= b
a >= b	greater_equal(bool a, bool b)	a <-> b
a < b	smaller(int a, int b)	a < b
a < b	smaller(bool a, bool b)	false
a > b	greater(int a, int b)	a > b
a > b	greater(bool a, bool b)	false
a = b	equal(int a, int b)	a = b
a = b	equal(bool a, bool b)	a <-> b
a + b	add(int a, int b)	a + b
a - b	subtract(int a, int b)	a - b
a * b	multiply(int a, int b)	a * b
a / b	divide(int a, int b)	\if (b < 0) \then (-a / -b) \else (a / b) <sup>6</sup>
a mod b	modulo(int a, int b)	a - b * (\if (b < 0) \then (-a / -b) \else (a / b) ) <sup>6</sup>
! a	not(bool a)	! a
- a	negative(int a)	- a
+ a	positive(int a)	a
# a	count?(T)(T[] a)	a.length
a [ i ]	subscript?(T)(T[] a, int i)	a [ i ]
	abs(int v)	\if (v < 0) \then (-v) \else (v)

---

<sup>6</sup>Mist uses Floored Division but Key uses Euclidean Division so a conversion is required.

## 5 Case Study: Polynomial Evaluation

As an example of the current capabilities of the Mist compiler, we present the following example program, for which we will automatically find a proof of correctness.


```
| '-----'|  
| poly() parameters and result |  
|-----|  
  
int p_a, p_b, p_c, p_x;  
int p_res;  
  
| '-----'|  
| sqr() parameters and result |  
|-----|  
  
int s_a;  
int s_res;
```


Because our compiler has not yet implemented the Hoare rule for function-calls with parameters and return-values, we have to use global variables for both. We use the following naming convention: variables starting with `p_` are related to the `poly` function; variables starting with `s_` are related to the `sqr` function.

The rest of the code follows:

```
void poly()  
logical int p_a0, p_b0, p_c0, p_x0;  
pre {  
    @ p_a0 = p_a;  
    @ p_b0 = p_b;  
    @ p_c0 = p_c;  
    @ p_x0 = p_x;  
}  
body {  
    s_a <- p_x;  
    sqr();  
    p_res <- p_a * s_res + p_b * p_x + p_c;  
}  
post {  
    @ p_res = p_a0 * p_x0 * p_x0 + p_b0 * p_x0 + p_c0;  
}
```

The `poly` function uses a rather convoluted way to evaluate a simple expression. This is to demonstrate the weakest precondition rule for non-recursive procedure-calls.

 Note that the postcondition does not guarantee that `p_a`, `p_b`, `p_c` and `p_x` retain their old value after a call to `poly`. In fact, there are no guarantees except that `p_res` will be set to the right value. So, for verification purposes, after a call to `poly`, all values except for that of `p_res` would be lost. This could be prevented by adding such an extra assurance to the postcondition (`@ p_a = p_a0;`, etc.).

 It is far from ideal that this has to be done for each variable that a function doesn't touch. Every time a global variable is added to the program, every function would have to add a guarantee not to change it, which is a disaster for modularity. This will be solved in a later version of `Mist`, in which the `poly` function can have a `changesonly(p_res)` clause. This clause expresses the guarantee that every non-local variable except for `p_res` is invariant in any call to the function.

```
void sqr()
logical int s_a0;
logical int p_a0, p_b0, p_c0, p_x0;
bound (abs(s_a))
pre {
  @ s_a0 = s_a;
  @ p_a0 = p_a;
  @ p_b0 = p_b;
  @ p_c0 = p_c;
  @ p_x0 = p_x;
}
body {
  s_a <- abs(s_a);
  if (s_a = 0) {
    s_res <- 0;
  } else {
    s_a <- s_a - 1;
    sqr();
    s_a <- s_a + 1;
    s_res <- s_res + 2 * s_a - 1;
  }
} post {
  @ s_res = s_a0 * s_a0;
  @ s_a = abs(s_a0);
  @ p_a0 = p_a;
  @ p_b0 = p_b;
  @ p_c0 = p_c;
  @ p_x0 = p_x;
}
```

The `sqr` function is even more awkward than the `poly` function, but there is

a purpose. The function computes the square of its parameter recursively, so we can demonstrate verification of recursive procedure-calls. Note that it is necessary for this function to explicitly express the invariance of the parameters to `poly`, because otherwise the `poly` function would have to operate without knowing if its parameters had been changed by the call to `sqr`, and `poly` could not be verified.

We will now single out a few fragments of the automatically generated proof. For the complete proof, see Appendix A.

```


| proof outline (total correctness)
@ s_a = s_a and p_a = p_a and p_b = p_b and p_c = p_c
@ and p_x = p_x and 0 <= abs(s_a) < oldbound and
@ (s_res_2 = s_a * s_a and s_a_2 = abs(s_a) and p_a =
@ p_a_2 and p_b = p_b_2 and p_c = p_c_2 and p_x = p_x_2
@ ==> s_res_2 + 2 * (s_a_2 + 1) - 1 = .sqr.s_a0 *
@ .sqr.s_a0 and s_a_2 + 1 = .abs(.sqr.s_a0) and
@ .sqr.p_a0 = p_a_2 and .sqr.p_b0 = p_b_2 and .sqr.p_c0
@ = p_c_2 and .sqr.p_x0 = p_x_2);

sqr();

| proof outline (total correctness)
@ s_res + 2 * (s_a + 1) - 1 = .sqr.s_a0 * .sqr.s_a0 and
@ s_a + 1 = .abs(.sqr.s_a0) and .sqr.p_a0 = .p_a and
@ .sqr.p_b0 = .p_b and .sqr.p_c0 = .p_c and .sqr.p_x0 =
@ .p_x;

```

Because the compiler does not simplify the assertions of the proof outline, the result is somewhat difficult to read. But, in order, the weakest precondition of the recursive call specifies: The first five equalities (`s_a = s_a` and `p_a = p_a` and `p_b = p_b` and `p_c = p_c` and `p_x = p_x`) are the `sqr` precondition logical variable simple forms, with the logical variables substituted by its simple definition. This always results in a number of trivial equalities.

 *In future versions, we may remove these, to make the proof more concise.*

The next part expresses that the state at the call is ‘smaller’ than the state at the beginning of the function. This proves that this specific call terminates. Then, between the brackets spanning to the end of the condition, rule 15 of Section 3.6 is instantiated.

```

| PROVED (total correctness): /tmp/key1.auto.0.proof
@ s_a0 = s_a and p_a0 = p_a and p_b0 = p_b and p_c0 = p_c
@ and p_x0 = p_x and 0 <= abs(s_a) = oldbound ==> (abs(s_a)
@ = 0 ==> 0 = .sqr.s_a0 * .sqr.s_a0 and abs(s_a) =
@ .abs(.sqr.s_a0) and .sqr.p_a0 = .p_a and .sqr.p_b0 = .p_b
@ and .sqr.p_c0 = .p_c and .sqr.p_x0 = .p_x) and
@ (!(abs(s_a) = 0) ==> abs(s_a) - 1 = abs(s_a) - 1 and p_a
@ = p_a and p_b = p_b and p_c = p_c and p_x = p_x and 0 <=
@ abs(abs(s_a) - 1) < oldbound and (s_res_2 = (abs(s_a) -
@ 1) * (abs(s_a) - 1) and s_a_2 = abs(abs(s_a) - 1) and p_a
@ = p_a_2 and p_b = p_b_2 and p_c = p_c_2 and p_x = p_x_2
@ ==> s_res_2 + 2 * (s_a_2 + 1) - 1 = .sqr.s_a0 * .sqr.s_a0
@ and s_a_2 + 1 = .abs(.sqr.s_a0) and .sqr.p_a0 = p_a_2 and
@ .sqr.p_b0 = p_b_2 and .sqr.p_c0 = p_c_2 and .sqr.p_x0 =
@ p_x_2));

```

This part shows the implication that KeY has proved correct, completing the proof-outline of the function `sqr`. The largest part of it is accumulated by weakest-precondition pushing. The first two lines constitute the precondition of the function itself. In the file `/tmp/key1.auto.0.proof`, the proof of the implication can be followed step-by-step in the language and environment of KeY.



*In the future, the KeY frontend will not be necessary to follow the implication proof. We will translate the KeY proof to Mist and offer the option to print it in a human readable format.*

For both functions, the compiler has found proof of total correctness. That is, they are certain to terminate, and upon termination they are certain to respect their postcondition. This means no run-time checks are necessary, and the compiler has the opportunity to automatically replace the call to `sqr` with a simple multiplication (`s_res <- s_a * s_a`), improving performance by an order of magnitude.

## 6 Conclusion

The Mist programming language has introduced syntax to make contract programming possible. We have implemented the Hoare-calculus and the consequence rule and procedure-call rules of Kleymann into the Mist compiler, so it can use weakest precondition logic for generating program proofs. We have prepared the compiler to flexibly accept an external theorem prover to prove the resulting implications and we have integrated the KeY theorem prover to do this. The compiler can now, for a subset of programs, generate and print a formal proof of correctness.

Upon constructing the Mist compiler and writing this thesis, we find that with current technology and scientific knowledge on formal verification, it could

soon be feasible for compilers to perform automatic verification on programs for the purposes explained in this thesis.<sup>7</sup> And in a future not too far off, the Mist programming language and compiler will be ready for compilation, verification and optimization of complex annotated projects using higher-level programming language features.

It is also very clear that there is a lot of work still to do before Mist is ready. Formal verification should not have to be restricted to the small subset of Mist it now is. The compiler should be able to verify function-calls with parameters and return-values, expressions with side-effects, array operations and for-loops iterating over collections. The programmer should be able to write function-definitions inductively, so interesting properties of collections may be expressed and sorting, partitioning and other more complex algorithms may be annotated automatically verified.

In the near future, the Mist compiler should, for example, be able to verify the following algorithm for calculation of Fibonacci numbers:

```

int fib(int n)
logical int N;
pre { @ N = n >= 0; }
post { @ N <= 1 ==> result = N;
      @ N > 1 ==> result = fib(N-2) + fib(N-1);
}
body {
  int k <- 0, x <- 0, y <- 1;
  while (k != n) bound (n - k) invariant {
    @ k <= 1 ==> x = k;
    @ k > 1 ==> x = fib(k-2) + fib(k-1);
    @ k > 0 ==> y = x + fib(k-1);
    @ N = n; k >= 0; k = 0 ==> y = 1;
  } do {
    (x, y) <- (y, x + y);
    k <- k + 1;
  }
  result <- x;
}

```

Note the use of parallel assignment (Appendix D), the inductive definition of `fib` and the use of its parameters and return-value. This function is easily verified on paper, and so we expect the Mist compiler to be able to do it within a few months.

The authors are in discussion with the KeY team to make the KeY theorem prover faster and altogether more suitable for use as a backend to our compiler. This will mean that we can try more proofs in less time. It will possibly be feasible to try to verify each assertion separately (rather than together), for a better

<sup>7</sup>Assuming, of course, that the programmer annotates his program with the necessary contracts, bounds and invariants.

indication of where a failed proof might have gone wrong. The KeY backend might be capable of simplifying its formula's, so the progressively growing proof outline assertions don't have to clutter up the proof so much. This requires a way to translate KeY formulae back to Mist, which is also being worked on.

We are working towards a more complete and formal standard for the Mist language, both the verification part and the rest. The Mist compiler is an open source project. Its website can be reached at:

<http://code.google.com/p/mist>

## A Full Case Study Proof

*For completeness sake, this appendix repeats parts from Section 5.*

The program we were trying to prove correct in the case study is the following:

```
| '-----'|
| poly() parameters and result |
|-----|

int p_a, p_b, p_c, p_x;
int p_res;

| '-----'|
| sqr() parameters and result |
|-----|

int s_a;
int s_res;

| '-----'|
| Evaluates the polynomial with coefficients p_a, p_b and p_c |
| in point p_x and puts the result in p_res. |
|-----|

void poly()
logical int p_a0, p_b0, p_c0, p_x0;
pre {
    @ p_a0 = p_a;
    @ p_b0 = p_b;
    @ p_c0 = p_c;
    @ p_x0 = p_x;
}
```

```

body {
    s_a <- p_x;
    sqr();
    p_res <- p_a * s_res + p_b * p_x + p_c;
}
post {
    @ p_res = p_a0 * p_x0 * p_x0 + p_b0 * p_x0 + p_c0;
}

| '-----' |
| Calculates the square of s_a, and puts it in s_res. |
|-----|

void sqr()
logical int s_a0;
logical int p_a0, p_b0, p_c0, p_x0;
bound (abs(s_a))

pre {
    @ s_a0 = s_a;
    @ p_a0 = p_a;
    @ p_b0 = p_b;
    @ p_c0 = p_c;
    @ p_x0 = p_x;
}
body {
    s_a <- abs(s_a);
    if (s_a = 0) {
        s_res <- 0;
    } else {
        s_a <- s_a - 1;
        sqr();
        s_a <- s_a + 1;
        s_res <- s_res + 2 * s_a - 1;
    }
}
post {
    @ s_res = s_a0 * s_a0;
    @ s_a = abs(s_a0);
    @ p_a0 = p_a;
    @ p_b0 = p_b;
    @ p_c0 = p_c;
    @ p_x0 = p_x;
}

```

The proof of this program is the following:

```

void poly()
logical int p_a0, p_b0, p_c0, p_x0;
logical int p_x_1;
logical int p_a_1;
logical int p_b_1;
logical int p_c_1;
logical int s_a_1;
logical int s_res_1;
pre {
  @ p_a0 = p_a;
  @ p_b0 = p_b;
  @ p_c0 = p_c;
  @ p_x0 = p_x;
}
body {
  | PROVED (total correctness): /tmp/key0.auto.0.proof
  @ p_a0 = p_a and p_b0 = p_b and p_c0 = p_c and p_x0 = p_x
  @ ==> p_x = p_x and p_a = p_a and p_b = p_b and p_c = p_c
  @ and p_x = p_x and (s_res_1 = p_x * p_x and s_a_1 =
  @ abs(p_x) and p_a = p_a_1 and p_b = p_b_1 and p_c = p_c_1
  @ and p_x = p_x_1 ==> p_a_1 * s_res_1 + p_b_1 * p_x_1 +
  @ p_c_1 = .poly.p_a0 * .poly.p_x0 * .poly.p_x0 + .poly.p_b0
  @ * .poly.p_x0 + .poly.p_c0);

  | proof outline (total correctness)
  @ p_x = p_x and p_a = p_a and p_b = p_b and p_c = p_c and
  @ p_x = p_x and (s_res_1 = p_x * p_x and s_a_1 = abs(p_x)
  @ and p_a = p_a_1 and p_b = p_b_1 and p_c = p_c_1 and p_x =
  @ p_x_1 ==> p_a_1 * s_res_1 + p_b_1 * p_x_1 + p_c_1 =
  @ .poly.p_a0 * .poly.p_x0 * .poly.p_x0 + .poly.p_b0 *
  @ .poly.p_x0 + .poly.p_c0);

  s_a <- p_x;

  | proof outline (total correctness)
  @ s_a = s_a and p_a = p_a and p_b = p_b and p_c = p_c and
  @ p_x = p_x and (s_res_1 = s_a * s_a and s_a_1 = abs(s_a)
  @ and p_a = p_a_1 and p_b = p_b_1 and p_c = p_c_1 and p_x =
  @ p_x_1 ==> p_a_1 * s_res_1 + p_b_1 * p_x_1 + p_c_1 =
  @ .poly.p_a0 * .poly.p_x0 * .poly.p_x0 + .poly.p_b0 *
  @ .poly.p_x0 + .poly.p_c0);

  sqr();

```

```

| proof outline (total correctness)
@ p_a * s_res + p_b * p_x + p_c = .poly.p_a0 * .poly.p_x0 *
@ .poly.p_x0 + .poly.p_b0 * .poly.p_x0 + .poly.p_c0;

p_res <- p_a * s_res + p_b * p_x + p_c;

| proof outline (total correctness)
@ .p_res = .poly.p_a0 * .poly.p_x0 * .poly.p_x0 + .poly.p_b0
@ * .poly.p_x0 + .poly.p_c0;
}
post {
    @ p_res = p_a0 * p_x0 * p_x0 + p_b0 * p_x0 + p_c0;
}

void sqr()
bound (abs(s_a))
logical int s_a0;
logical int p_a0, p_b0, p_c0, p_x0;
logical int p_x_2;
logical int p_a_2;
logical int p_b_2;
logical int p_c_2;
logical int s_a_2;
logical int s_res_2;
pre {
    @ s_a0 = s_a;
    @ p_a0 = p_a;
    @ p_b0 = p_b;
    @ p_c0 = p_c;
    @ p_x0 = p_x;
}

```

```

body {
  | PROVED (total correctness): /tmp/key1.auto.0.proof
  @ s_a0 = s_a and p_a0 = p_a and p_b0 = p_b and p_c0 = p_c
  @ and p_x0 = p_x and 0 <= abs(s_a) = oldbound ==> (abs(s_a)
  @ = 0 ==> 0 = .sqr.s_a0 * .sqr.s_a0 and abs(s_a) =
  @ .abs(.sqr.s_a0) and .sqr.p_a0 = .p_a and .sqr.p_b0 = .p_b
  @ and .sqr.p_c0 = .p_c and .sqr.p_x0 = .p_x) and
  @ (!(abs(s_a) = 0) ==> abs(s_a) - 1 = abs(s_a) - 1 and p_a
  @ = p_a and p_b = p_b and p_c = p_c and p_x = p_x and 0 <=
  @ abs(abs(s_a) - 1) < oldbound and (s_res_2 = (abs(s_a) -
  @ 1) * (abs(s_a) - 1) and s_a_2 = abs(abs(s_a) - 1) and p_a
  @ = p_a_2 and p_b = p_b_2 and p_c = p_c_2 and p_x = p_x_2
  @ ==> s_res_2 + 2 * (s_a_2 + 1) - 1 = .sqr.s_a0 * .sqr.s_a0
  @ and s_a_2 + 1 = .abs(.sqr.s_a0) and .sqr.p_a0 = p_a_2 and
  @ .sqr.p_b0 = p_b_2 and .sqr.p_c0 = p_c_2 and .sqr.p_x0 =
  @ p_x_2));

  | proof outline (total correctness)
  @ (abs(s_a) = 0 ==> 0 = .sqr.s_a0 * .sqr.s_a0 and abs(s_a)
  @ = .abs(.sqr.s_a0) and .sqr.p_a0 = .p_a and .sqr.p_b0 =
  @ .p_b and .sqr.p_c0 = .p_c and .sqr.p_x0 = .p_x) and
  @ (!(abs(s_a) = 0) ==> abs(s_a) - 1 = abs(s_a) - 1 and p_a
  @ = p_a and p_b = p_b and p_c = p_c and p_x = p_x and 0 <=
  @ abs(abs(s_a) - 1) < oldbound and (s_res_2 = (abs(s_a) - 1)
  @ * (abs(s_a) - 1) and s_a_2 = abs(abs(s_a) - 1) and p_a =
  @ p_a_2 and p_b = p_b_2 and p_c = p_c_2 and p_x = p_x_2 ==>
  @ s_res_2 + 2 * (s_a_2 + 1) - 1 = .sqr.s_a0 * .sqr.s_a0 and
  @ s_a_2 + 1 = .abs(.sqr.s_a0) and .sqr.p_a0 = p_a_2 and
  @ .sqr.p_b0 = p_b_2 and .sqr.p_c0 = p_c_2 and .sqr.p_x0 =
  @ p_x_2));

  s_a <- abs(s_a);

```

```

| proof outline (total correctness)
@ (s_a = 0 ==> 0 = .sqr.s_a0 * .sqr.s_a0 and .s_a =
@ .abs(.sqr.s_a0) and .sqr.p_a0 = .p_a and .sqr.p_b0 = .p_b
@ and .sqr.p_c0 = .p_c and .sqr.p_x0 = .p_x) and (!(s_a = 0)
@ ==> s_a - 1 = s_a - 1 and p_a = p_a and p_b = p_b and p_c
@ = p_c and p_x = p_x and 0 <= abs(s_a - 1) < oldbound and
@ (s_res_2 = (s_a - 1) * (s_a - 1) and s_a_2 = abs(s_a - 1)
@ and p_a = p_a_2 and p_b = p_b_2 and p_c = p_c_2 and p_x =
@ p_x_2 ==> s_res_2 + 2 * (s_a_2 + 1) - 1 = .sqr.s_a0 *
@ .sqr.s_a0 and s_a_2 + 1 = .abs(.sqr.s_a0) and .sqr.p_a0 =
@ p_a_2 and .sqr.p_b0 = p_b_2 and .sqr.p_c0 = p_c_2 and
@ .sqr.p_x0 = p_x_2));

if (s_a = 0) {
  | proof outline (total correctness)
  @ 0 = .sqr.s_a0 * .sqr.s_a0 and .s_a = .abs(.sqr.s_a0)
  @ and .sqr.p_a0 = .p_a and .sqr.p_b0 = .p_b and
  @ .sqr.p_c0 = .p_c and .sqr.p_x0 = .p_x;

  s_res <- 0;

  | proof outline (total correctness)
  @ .s_res = .sqr.s_a0 * .sqr.s_a0 and .s_a =
  @ .abs(.sqr.s_a0) and .sqr.p_a0 = .p_a and .sqr.p_b0
  @ = .p_b and .sqr.p_c0 = .p_c and .sqr.p_x0 = .p_x;
}
else {
  | proof outline (total correctness)
  @ s_a - 1 = s_a - 1 and p_a = p_a and p_b = p_b and p_c
  @ = p_c and p_x = p_x and 0 <= abs(s_a - 1) < oldbound
  @ and (s_res_2 = (s_a - 1) * (s_a - 1) and s_a_2 =
  @ abs(s_a - 1) and p_a = p_a_2 and p_b = p_b_2 and p_c
  @ = p_c_2 and p_x = p_x_2 ==> s_res_2 + 2 * (s_a_2 + 1)
  @ - 1 = .sqr.s_a0 * .sqr.s_a0 and s_a_2 + 1 =
  @ .abs(.sqr.s_a0) and .sqr.p_a0 = p_a_2 and .sqr.p_b0 =
  @ p_b_2 and .sqr.p_c0 = p_c_2 and .sqr.p_x0 = p_x_2);

  s_a <- s_a - 1;

```

```

| proof outline (total correctness)
@ s_a = s_a and p_a = p_a and p_b = p_b and p_c = p_c
@ and p_x = p_x and 0 <= abs(s_a) < oldbound and
@ (s_res_2 = s_a * s_a and s_a_2 = abs(s_a) and p_a =
@ p_a_2 and p_b = p_b_2 and p_c = p_c_2 and p_x = p_x_2
@ ==> s_res_2 + 2 * (s_a_2 + 1) - 1 = .sqr.s_a0 *
@ .sqr.s_a0 and s_a_2 + 1 = .abs(.sqr.s_a0) and
@ .sqr.p_a0 = p_a_2 and .sqr.p_b0 = p_b_2 and .sqr.p_c0
@ = p_c_2 and .sqr.p_x0 = p_x_2);

sqr();

| proof outline (total correctness)
@ s_res + 2 * (s_a + 1) - 1 = .sqr.s_a0 * .sqr.s_a0 and
@ s_a + 1 = .abs(.sqr.s_a0) and .sqr.p_a0 = .p_a and
@ .sqr.p_b0 = .p_b and .sqr.p_c0 = .p_c and .sqr.p_x0 =
@ .p_x;

s_a <- s_a + 1;

| proof outline (total correctness)
@ s_res + 2 * s_a - 1 = .sqr.s_a0 * .sqr.s_a0 and .s_a
@ = .abs(.sqr.s_a0) and .sqr.p_a0 = .p_a and .sqr.p_b0
@ = .p_b and .sqr.p_c0 = .p_c and .sqr.p_x0 = .p_x;

s_res <- s_res + 2 * s_a - 1;

| proof outline (total correctness)
@ .s_res = .sqr.s_a0 * .sqr.s_a0 and .s_a =
@ .abs(.sqr.s_a0) and .sqr.p_a0 = .p_a and .sqr.p_b0 =
@ .p_b and .sqr.p_c0 = .p_c and .sqr.p_x0 = .p_x;
}

| proof outline (total correctness)
@ .s_res = .sqr.s_a0 * .sqr.s_a0 and .s_a = .abs(.sqr.s_a0)
@ and .sqr.p_a0 = .p_a and .sqr.p_b0 = .p_b and .sqr.p_c0 =
@ .p_c and .sqr.p_x0 = .p_x;
}
post {
@ s_res = s_a0 * s_a0;
@ s_a = abs(s_a0);
@ p_a0 = p_a;
@ p_b0 = p_b;
@ p_c0 = p_c;
@ p_x0 = p_x;
}

```

## B Grammar

This section will give a detailed summary of the structure of the Mist language. Sections B.1 to B.7 explain the basic building blocks of the language that are recognized in the lexical analysis phase. Sections B.8 to B.25 give the grammar of the language in EBNF [1] (only without concatenation commas; additionally, non-terminals will not use spaces), with short descriptions as to syntax, semantics and type.

### B.1 Keywords

```
if else while for void pre post logical body bound
do invariant mod cpp KeY type new and or true false
```

These keywords are reserved by the core language and cannot be used as identifiers. Their semantics are explained in the subsections below. The compiler will recognize attempts to declare a symbol with one of these names and will abort compilation if necessary.

### B.2 Tokens

```
.. ^ ' @ ( ) { } [ ] , ; . <- <-> + - * / # ! "
<==> <== ==> = < > <= >= <> <>= | |- -| |+ +|
```

These are the tokens/operators available in the core language. Their semantics are explained in the subsections below. Comments are explained in the ‘Comments’ section.

### B.3 Comments

Comments are ignored and discarded by the parser. They can be used to make code more understandable for (other) programmers where necessary, or as a useful way to temporarily remove code in the debugging phase.

```
| line comment
```

A line comment starts with the | token and ends with the first newline. Everything between the two will be ignored by the parser.

```
|+
  |+ nesting +|
  block comment
+|
```

A nesting block comment starts with the |+ token and ends with the *matching* +| token. You need to keep the opening and closing delimiters well-balanced, like parentheses. This construct can span multiple lines, or part of a single line.

```
|-
    non-nesting
    block comment
-|
```

A non-nesting block comment starts with the `|-` token and ends with the first `-|` token. Everything between them, including other `|-` tokens, will be ignored by the parser. This construct can span multiple lines, or part of a single line.

For recommended comment usage, see Appendix C.

### B.4 Identifier

```
identifier: [_a-zA-Z][_a-zA-Z0-9]*
```

Identifiers are the names of programmer-declared symbols in the language. They can stand for variables, functions, scopes and types, which share the same namespace.

An identifier is formed by starting with an underscore or letter, followed by any string consisting of underscores, letters and numbers.

### B.5 Integer Literal

```
integer_literal: [0-9][0-9]* (1)
                | 0[xX][0-9a-fA-F][0-9a-fA-F]* (2)
                | 0[oOqQ][0-7][0-7]* (3)
                | 0[bB][01][01]* (4)
```

An integer literal can be given in either decimal base (rule 1), hexadecimal base (rule 2), octal base (rule 3) or binary base (rule 4). The following four integer literals would be semantically equivalent: `42`, `0x2a`, `0q52`, `0b101010`. Negative integer literals are not recognized by the lexer, but are recognized by the parser as a unary minus followed by an integer literal. Note that integer literals can have any length, as integers in the language may have any magnitude.

### B.6 Boolean Literal

```
boolean_literal: "true" | "false"
```

A boolean literal represents a single truth value. `true` or `false`.

### B.7 String Literal

```
string_literal: ''' ?string content? '''
```

A string literal can at present only be used for `Key` and `cpp` sections in the code. It is delimited by two `"` characters. The string content may be any sequence of characters, but it may not contain an unescaped `"` character. It can contain escape sequences. An escape sequence is one of:

---

<code>\b</code>	backspace
<code>\t</code>	horizontal tab
<code>\n</code>	new line
<code>\f</code>	formfeed
<code>\r</code>	carriage return
<code>\"</code>	" character
<code>\\</code>	\ character
<code>\[0-9a-fA-F][0-9a-fA-F]</code>	ASCII character in hexadecimal notation

---

## B.8 Declarations (global scope)

```
declarations: { function_declaration
                | variable_declaration
                | type_declaration }
```

At the root of the syntax tree lies the global scope, the entry-point of the grammar, which may contain any number of function declarations, variable declarations and type declaration. In a valid program, the global scope should at least contain a function `int run()`, which will be the function that's called after the global scope initializations are done.

For a formal treatment of the related topic of symbol visibility, see Section 2.2.

## B.9 Function Declaration

```
function_declaration: function_signature
                     [ '{' { statement } '}' ]
                     function_sections

function_signature:  type identifier
                     [ template_type_declaration ]
                     '(' [ formal_parameter_list ] ')

template_type_declaration: '?' '(' [ identifier_list ] ')'

formal_parameter_list: formal_parameter { ',' formal_parameter }

formal_parameter:    type identifier

identifier_list:     identifier { ',' identifier }

function_sections:   "pre" '{' assertion_area '}'
                    | "post" '{' assertion_area '}'
                    | "logical" variable_declaration
                    | "body" '{' { statement } '}'
                    | "bound" '(' expression ')'
                    | "KeY" string_literal
```

A function declaration consists of a function signature followed by one to six other constructs. The function signature specifies the return type of the function, the name of the function and the formal parameters of the function. Each formal parameter has a name and a type.

After the signature will follow at least the body of the function, which will define its behavior at runtime. It is also possible to add a precondition, a postcondition, logical variable declarations, a recursive function bound and a KeY translation (which gives the function meaning in the KeY environment).

Note that if the body is not the first to appear after the signature, it requires the `body` keyword. Know also that this grammar is not complete, since it allows more than one of the same section. Each function declaration may have no more than one body, precondition, postcondition, bound and KeY section.

For more information about scoping, see Section 2.2.


## B.10 Variable Declaration

```
variable_declaration: type variable_declaration_list ';'

variable_declaration_list: variable_declaration_pair
                           { ',' variable_declaration_pair }

variable_declaration_pair: identifier [ "<-" expression ]
```


A variable declaration defines the name and type of one or more new variables. Additionally, it is possible to initialize these variables by assigning a new value immediately. This saves having to repeat the variable name in a separate assignment.

 *In a future version of Mist, the `auto` keyword will be introduced. It can be used instead of a type for initialized variable declarations. In those cases, the type of the variable will be automatically deduced from the initialization.*

## B.11 Type Declaration

```
type_declaration: "type" identifier
                  "cpp" string_literal
                  "KeY" string_literal ';' ;'
```

A type declaration can, at this moment, only expose primitive types to C++ and KeY. It is used to expose `int` and `bool`.

 *In future versions of Mist, more types of type declarations may be used, including the declaration of new classes and type aliases.*

## B.12 Type

```
type: "void"      (1)
      | reference  (2)
      | type '[' ']' (3)
      | type '^'   (4)
      | type '"'   (5)
```

A type represents the set of possible values a variable or expression can represent. Additionally, if the type is a pointer, the type specifies if the reference is strong or weak.

The possible types are:

1. A dummy type to specify the absence of a return value for a function. In future versions, it will be an alias for the nullary tuple (see Appendix D).
2. A primitive type (`int` or `bool`) or a function template type.
3. A sequence of elements from its content-type.
4. A strong pointer to an element from its specified type.
5. A weak pointer to an element from its specified type.

## B.13 Expression

```
expression: operator_call
           | function_call
           | subscripting
           | reference
           | boolean_literal
           | integer_literal
           | '(' expression ')'
```

An expression has (or calculates) a value. Everywhere you need this value in your program, you can substitute the expression.

Every expression has a type, which may be determined by its sub-expressions. Only expressions of certain types are permissible in certain contexts, depending on the available overloads of functions and parameters, for example. Some statements also restrict the type of the expressions they accept.

Every expression has also an access-type. This indicates the access-level to value of this expression.

## B.14 Operator Call

```
operator_call: expression    infix_op  expression
              |              prefix_op  expression
              | expression  postfix_op

prefix_op: '!' | '#' | '+' | '-' | '^' | "new"

postfix_op: '^'

infix_op: "<->" | "<-" | ".." | '+' | '-'
         | '*' | '/' | "mod" | binary_compound_op

binary_compound_op: { '!' } ( "or" | "and" | implication_op
                              | comparison_op )

implication_op: "<==" | "==" | "<==>"

comparison_op: '<' | '>' | '=' | "<=" | ">=" | "<>="
```

Operator precedence and associativity is not built into the grammar specified here. This is important information when you're writing code in Mist, though. The following table lists all operators in order of precedence (higher binds stronger), with related information.

Operator(s)	Arity	Position	Associativity
↑ <i>higher precedence</i> ↑			
( ), [ ], ^	Unary	Postfix	
~, !, #, --, +, new	Unary	Prefix	
*, /, mod	Binary	Infix	Left-associative
+, -	Binary	Infix	Left-associative
..	Binary	Infix	Left-associative
=, <, >, <=, >=, <>, <>=	Binary	Infix	Chain-associative
and	Binary	Infix	Left-associative
or	Binary	Infix	Left-associative
<==, ==>, <==>	Binary	Infix	Chain-associative
<-	Binary	Infix	Right-associative
<->	Binary	Infix	Non-associative
↓ <i>lower precedence</i> ↓			

This table shows the operators grouped by classification. Each operator within a group has equal *precedence*, *arity*, *position* and *associativity*.

The groups are ordered by descending precedence. So, for example,  $a + b * c$  is evaluated like  $a + (b * c)$ , because  $*$  binds stronger than  $+$ .

Each operator is either a binary or a unary operator. For example, **and** accepts two operands. It is *binary*. **!** accepts one operand. It is *unary*. The function-call and subscripting operators may be a little misleading. Since only their base operand matters with regard to precedence and position, we classify them as unary operators for the purpose of these rules.

Each binary operator is placed in an *infix* position. That is, it is placed between its two operands. The first group of operators are placed in a *postfix* position. The braces are placed to the right of their base operand. The second group of unary operators are placed in a *prefix* position. To the left of their operand.

The unary operators have no associativity. That is, if they are used in combination with more operators of the same group, there can be no ambiguity as to the meaning, as they have the same position. It is different for the binary operators, which have, if used in combination with more operators of the same group, an ambiguity to resolve.  $(a - b) + c$  has a different meaning than  $a - (b + c)$ . If no parentheses are specified, the first version is used, because  $+$  and  $-$  are *left-associative*. Only the assignment operator is *right-associative*. The right-most operator is evaluated first in that case.

There are two special cases. The comparison operators and implication operators are *chain-associative*. This means that when they are used in combination with more operators of the same group, each binary operation is evaluated independently and implicit **and** operators are inserted. So, for example,  $0 <= i < j <= N$  is evaluated as  $(0 <= i) \text{ and } (i < j) \text{ and } (j <= N)$ . But the inner operands  $i$  and  $j$  are evaluated only once, including any side effects they may have.

Any evaluation order imposed by these rules may be overridden by using

( ) *parentheses* to isolate a subexpression.



*In Mist 0.2, / will remain the floored division operator and return an integer. But in the next version, Mist will feature the rational number (rat) type. From that version on, the / operator will return a rational number. The floored division operator will become //.*

## B.15 Function Call

```
function_call: expression '(' [ expression_list ] ')'  
expression_list: expression { ',' expression }
```

Using the function-call notation invokes the function specified on the left of the opening parenthesis and passes the parameters in the expression-list between the brackets.

The base of the call and the number and types of the formal parameters of the function called determine the function that is called. The functioncall-expression is of the same type as the return type of the function called.

## B.16 Subscripting

```
subscripting: expression '[' expression ']'
```

Subscripting is accessing a specific element from a sequence. The base of the subscripting (the first expression from the rule) should be an array. The subscript (the second expression) may be any expression of type `int`. The subscripting expression is of the content-type of the base array.

## B.17 Statement

```
statement: declaration  
          | expression_statement  
          | if_statement  
          | while_statement  
          | for_statement  
          | compound_statement  
          | assertion_area  
          | cpp_statement
```

All of the non-terminals listed above represent a statement. They can be used inside function bodies. They can perform an action, declare a symbol or make an assertion.

## B.18 Expression Statement

```
expression_statement: expression ';' ;'
```

When an expression is followed by a semi-colon, it can serve as a statement. This is usually done because of its side-effects. Most likely, the expression is an assignment or function-call, but Mist semantics allow any expression to be used.

## B.19 C++ Statement

```
"cpp" string_literal ';' 
```

At the moment, Mist is translated into C++, and the C++ statement is how all primitive operations are implemented in the Mist runtime environment. The string literal should contain a valid C++ statement, which may contain references to Mist variables.

## B.20 If Statement

```
if_statement: "if" '(' expression ')' statement  
             [ "else" statement ]
```

The if statement is a control-flow mechanism. If, at runtime, the specified expression evaluates to `true`, the first statement is executed. If the else clause is included and the expression evaluates to `false`, the second statement is executed instead.

Both the ‘then’-statement and the ‘else’-statement enter a single new scope, whether { } braces are used or not.

Note that `else if` constructs will automatically work. `else if` really just starts a new if statement in the else-clause of the previous one. As a special exception, no new scope is introduced by such an else clause.

## B.21 Compound Statement

```
compound_statement: [ identifier ] '{' { statement } '}'
```

A compound statement groups several statements together and creates a new scope for them. This scope may be given a name by putting an identifier before the { token. By default, a compound statement scope is unnamed.

For more in-depth information about scoping rules, visit Section 2.2.

## B.22 For Statement

```
for_statement: "for" '(' type identifier ',' expression ')' statement
```

The for statement is a control flow structure. It executes the specified statement once for each element in the collection specified to the right of the comma. Each time, the variable declared to the left of the comma takes the value of the current element in each iteration.

There is, right now, no way to involve a for-loop in a proof-outline. But there will be.

## B.23 While Statement

```
while_statement: "while" '(' expression ')'  
                [ statement ] while_statement_sections  
  
while_statement_sections: { "do" statement  
                           | "bound" '(' expression ')'  
                           | "invariant" '{' assertion_area '}' } }
```

The while statement is a control-flow mechanism. It executes as follows:

1. The specified condition is evaluated.
2. If it evaluated to **false**, go to step 5.
3. Execute the statement.
4. Go back to step 1.
5. Terminate the loop statement.

The condition should be a boolean expression. For verification purposes, you may specify a loop invariant and a loop bound, as explained in Section 2.3. If the loop body is not the first to be specified, it requires the prefix **do** keyword. Exactly one loop body must be specified. Not more than one invariant and one bound may be specified.

## B.24 Assertion Area

```
assertion_area: ?begin? { expression ';' } ?end?
```

The assertion area is the most basic redundant information used for program verification. It can occur inside a body of code, but also inside function contracts and loop invariants. For detailed information about assertions, read Section 2.3.

An assertion starts with a **@** token. It ends with the first newline not directly followed by another **@** token (not including whitespace). Those extra **@** tokens may break up any containing expression into separate lines. Such expressions remain semantically the same. This is handled in the lexical analyzer.

## B.25 Reference

```
reference: reference_no_spec
          | reference_no_spec formal_type_spec
          | reference_no_spec formal_type_spec template_type_spec
          | reference_no_spec template_type_spec
          | reference_no_spec template_type_spec formal_type_spec

reference_no_spec: [ [ reference ] '.' ] reference_part

reference_part: identifier | "pre" | "post" | "invariant"

formal_type_spec: '!' '(' [ type_list ] ')

template_type_spec: '?' '(' [ type_list ] ')

type_list: type { ',' type }
```

A reference can be resolved to a declared symbol, such as a variable, a function or a type. But there are multiple ways to reference the same symbol (see Section 2.2.3), multiple symbols with the same name because of function overloading and because of function templates, template instantiation information also needs to be passed.

A reference can specify formal parameter types to reference a specific function overload using the !() notation. A reference can specify template type instantiations to instantiate a specific templated function using the ?() notation. The two notations have to unambiguously reference a single symbol. If not, the reference is invalid.

Of course, this grammar also allows the most simple variable or function reference, using only one identifier.

For more information about symbol visibility, see Section 2.2.2.

## C Comments

Comments are ignored and discarded by the parser. They can be used to make code more understandable for (other) programmers where necessary, or as a useful way to temporarily remove code in the debugging phase.

### C.1 Syntax

*For completeness sake, this subsection is being repeated from Section B.3.*

```
| line comment
```

A line comment starts with the | token and ends with the first newline. Everything between the two will be ignored by the parser.



Drawing a box around your documentation helps to make it stand out. Especially in editors without syntax highlighting. It is recommended for documentation of large bodies of code, like functions, classes or files. Specialized editors should relieve programmers of the burden of drawing these boxes manually.

### C.2.2 Zapping Code

```
|+  
  
code.code.code.code;  
code.code.code.code;  
  
+|
```

In the debugging phase of development, it may be useful to temporarily hide a large portion of code from the compiler. Nesting block comments are ideal for this purpose. They can 'comment out' entire blocks of code, even if that code already contains other well-balanced block comments.

### C.2.3 Comment Switches

If you find yourself adding and removing the same piece of code repeatedly, most likely for debugging purposes, you may want to use a comment switch: a universally recognized way to quickly toggle certain code on and off. We recommend using non-nesting block comments for this purpose, so they do not interfere with code-zapping, among other reasons.

<pre>  -   code.code.code.code;   - -  </pre>	<pre>  - -   code.code.code.code;   - -  </pre>
---	---

In the above example, the code to the left is currently being ignored by the parser. The code on the right is active.

<pre>  -   code.code.code.code.A;   -   code.code.code.code.B;   - -  </pre>	<pre>  - -   code.code.code.code.A;   -   code.code.code.code.B;   - -  </pre>
--	--

In this last example, on the left, code A is being ignored while code B is active. On the right it's the other way around. Just swap that first `| - |` for a `| - - |`.

While it is true that this technique can save you valuable seconds, that is not the primary reason for using it. The fact that the switch only requires one keystroke gives these character sequences the affordance [16] to be used like this. If programmers use it consistently, the intent will be immediately understood.

## D Tuples



*Tuples currently do not work in the Mist compiler. They are planned for version 0.3 of the language.*

Mathematically, a tuple is an ordered sequence of values of finite length. There are no limits with regard to the type of each value. In particular, it is not necessary for each value to have the same type. Mist will implement the mathematical concept of tuples in the core language. A tuple value could look something like this: `(42, true, "foo")`.

The following sections will, one by one, display the possibilities of Mist tuples. A formal semantics of them has not been written yet, but their design has been fixed.

### D.1 Tuple Variable

A single variable can be of tuple type:

```
(int, bool, char) T <- (1, false, 'a');
```

### D.2 Parallel Assignment

A tuple can be unpacked using parallel assignment:

```
int i;  
bool b;  
char c;  
(i, b, c) <- T;
```

*i*, *b* and *c* are assigned the values 1, `false` and `'a'` simultaneously (we're using the *T* from the previous example). You can also do this:

```
(i, j, k) <- (j, k, i);
```

This effectively rotates (swaps) the values of *i*, *j* and *k*, without needing an explicit temporary variable.

### D.3 Tuple Member Access

The values of a tuple may also be extracted by name, if they were named at declaration.

```
(int a, int b) C <- (1, 2);
```

Now *C* equals `(1, 2)`, *C.a* equals 1 and *C.b* equals 2. Note that the type of *C* is still `(int, int)`. The names *a* and *b* are not part of the type.

A related syntax may be used to simplify an earlier example, where the values of a tuple variable are extracted by position:

```
(int i, bool b, char c) <- T;
```

This declares the global variables  $i$ ,  $b$  and  $c$  while assigning to them the values of  $T$ . Tuples and Functions

A function can be thought of as taking, not a number of parameters, but one tuple parameter. In Mist, this is indeed true:

```
void foo(int i, int j, int k) { ... }

(int, int, int) P <- (1, 2, 3);

foo(1, 2, 3);
foo(P);
```

Also, functions return tuples. The type `void` is the nullary tuple type  $()$ . A single type  $T$  is the unary tuple type  $(T)$ . And functions can also return  $n$ -ary tuples, where  $n > 1$ :

```
(quotient, remainder) <- divide(i, j);
```

## D.4 Tuples and Arrays

The `#` operator in Mist returns the count (or size) of a collection:

```
int[10][10] A;
int c <- #A;
```

$c$  would be equal to 10. But say you have a matrix, rather than an array of arrays:

```
int[10, 10] M;
(int, int) c <- #M;
```

This would make  $c$  equal to  $(10, 10)$ . Unsurprisingly, an index to  $M$  would also be a binary tuple:

```
int[10, 10] M;
(int, int) I <- (5, 5);
M[I] <- 100;
```

Arrays are a lot like functions in this way. In fact, in the future, the distinction between arrays and functions may disappear.

Another relation between tuples and arrays is the fact that a tuple in which every element is of the same type is implicitly convertible to a fixed-size array of that type. So you can write code like this:

```
int[5] A <- (5, 6, 7, 8, 9);
```

## D.5 Lexicographical Ordering

Given a tuple type, if each of its element types has a partial order imposed on it, then so does that tuple type. If each of its element types has a total order imposed on it, then so does the tuple type.

A type has a partial but not a total order iff it contains two values ( $a$  and  $b$ ) that are incomparable, denoted  $a \parallel b$ .  $a$  would be neither smaller, equal nor greater than  $b$ .

The ordering of a tuple type is defined recursively. In the following table,  $V_1$  and  $V_2$  are single values of any (but the same) type.  $T_1$  and  $T_2$  are tuples of any (but the same) type.

If	Then
<b>true</b>	$() = ()$
$V_1 < V_2$	$(V_1, T_1) < (V_2, T_2)$
$V_1 > V_2$	$(V_1, T_1) > (V_2, T_2)$
$V_1 \parallel V_2$	$(V_1, T_1) \parallel (V_2, T_2)$
$V_1 = V_2$ and $T_1 = T_2$	$(V_1, T_1) = (V_2, T_2)$
$V_1 = V_2$ and $T_1 < T_2$	$(V_1, T_1) < (V_2, T_2)$
$V_1 = V_2$ and $T_1 > T_2$	$(V_1, T_1) > (V_2, T_2)$
$V_1 = V_2$ and $T_1 \parallel T_2$	$(V_1, T_1) \parallel (V_2, T_2)$

That means tuples follow a lexicographical order. Note that  $T_1$  and  $T_2$  can be the nullary tuple. This is necessary for the definition to work.

Intuitively, if the elements of two same-type tuples ( $\#A = \#B = n$ ) are pairwise equal, meaning

$$A[1] = B[1] \wedge A[2] = B[2] \wedge \dots \wedge A[n] = B[n],$$

then  $A = B$ . If the elements are not pairwise equal, then  $A$  and  $B$  share the same relation that the first non-equal pair has. So  $(1, 2, 2) < (1, 2, 3)$  because  $1 = 1$ ,  $2 = 2$  and  $2 < 3$ .

Here is a practical example of tuple comparison:

```
bool earlier(Time a, Time b) {
    result <- (a.hour, a.minute, a.second) <
              (b.hour, b.minute, b.second);
}
```

## D.6 Tuples and Intervals

The Mist language has an interval operator:  $a..b$ , which would return a sequence with the values from  $a$  (lower-bound) to  $b$  (inclusive upper bound). For example,  $1..5 = (1, 2, 3, 4, 5)$ .

If you want to iterate over all indices of a 10x10 matrix  $M$ , you can simply do this:  $(1, 1)..(10, 10)$  or  $(1, 1)..\#M$ , which is a sequence of 100 binary tuples which can be used to index  $M$ , in lexicographical order:

(1, 1), (1, 2), ..., (1, 9), (1, 10), (2, 1), (2, 2), ..., (10, 9), (10, 10)

## D.7 Tuple Flattening

In this section, we will describe a property of tuples you may have noticed from the earlier examples.

A single value is equal (even identical) to a unary tuple with that value. Syntactically, it makes sense, since parentheses are also used to isolate subexpressions. It also works conceptually. It is a simple case of tuple flattening. In fact, (5, 6, ()) is the same as (5, 6) and (1, (2, 3), 4) is the same as (1, 2, 3, 4). A way of defining tuples is to use nested ordered pairs. Tuple ordering is also defined this way (see above).

There may be situations, however, when you want to avoid tuple flattening in your code. For example, when you're writing a multi-dimensional array literal:

```
int[2][3] A <- ((1, 2), (3, 4), (5, 6)); | ERROR
```

This does not work, because the type of the right-hand side is actually `int[6]`, if converted to array type.

So we want another notation next to this one to denote non-flattening tuples. That would be the following notation:

```
int[2][3] A <- ([1, 2], [3, 4], [5, 6]);
```

Note that the three inner tuples now use `[ ]` brackets, so they do not flatten. If it does not matter if a tuple is flattening or not, either notation can be used.

Tuple flattening goes as deep as tuple variables, which can also be flattening:

```
(int, int) t <- (2, 3);  
(int, int, int, int) u <- (1, t, 4);  
(t, int a, int b) <- u;
```

After this example,  $u$  equals (1, 2, 3, 4),  $t$  equals (1, 2) and  $(a, b)$  equals (3, 4). However:

```
[int, int] t <- (2, 3);  
(int, int, int, int) u <- (1, t, 4); | ERROR
```

The second line of this example is an error. Because of  $t$  being non-flattening, the assignment is a type-mismatch.

Flattening tuple variables may be useful for many tasks, but they were designed with a very specific purpose in mind. Variadic functions, functions which can take a variable number of actual parameters:

```
int sum?(T)(int a, T b) {
    result <- a + sum(b);
}

int sum() {
    result <- 0;
}

int x <- sum(1, 2, 3, 4);
```

This is what's happening:

The function `sum!(int, int, int, int)` is called.  $a = 1, b = (2, 3, 4)$ .

It calls the function `sum!(int, int, int)`.  $a = 2, b = (3, 4)$ .

It calls the function `sum!(int, int)`.  $a = 3, b = 4$ .

It calls the function `sum!(int)`.  $a = 4, b = ()$ .

It calls the function `sum!()`.

It returns 0.

It returns  $(4 + 0 = 4)$ .

It returns  $(3 + 4 = 7)$ .

It returns  $(2 + 7 = 9)$ .

It returns  $(1 + 9 = 10)$ .

$x$  is assigned the value 10.

The second function serves as the base-case of the recursion. The first function is the one that gets initially called. Its formal parameter  $b$  will automatically be the tuple containing every actual parameter except the first, which will go to  $a$ . In effect, because of the function-call, four functions are automatically generated, each one with a different type for  $T$ .

## Acknowledgements

The authors would like to thank Frank de Boer and Marcello Bonsangue for the discussions, ideas and support that lead to the development of our verification theory as it is today. We also thank Wolfgang Ahrendt and Richard Bubel, of the KeY project, for being a part of the creation of KeY and for answering our e-mail questions about it with enthusiasm.

## References

- [1] *ISO/IEC 14977:1996, Extended BNF*, first edition, 1996.
- [2] *Logic in Computer Science: Modeling and Reasoning About Systems*. Cambridge University Press, second edition, 2004.

- [3] *Standard ECMA-367, Eiffel: Analysis, Design and Programming Language*, second edition, June 2006.
- [4] J.R. Abrial, M.K.O. Lee, D. Neilson, PN Scharbach, and I. Sørensen. The B-method. In *Proceedings of the 4th International Symposium of VDM Europe on Formal Software Development*, volume 2, pages 398–405. Springer.
- [5] Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*. LNCS 4334. Springer-Verlag, 2007.
- [6] Joshua Bloch. Extra, Extra - Read All About It: Nearly All Binary Searches and Mergesorts are Broken. <http://googleresearch.blogspot.com/2006/06/extra-extra-read-all-about-it-nearly.html> [online], 2006.
- [7] Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joseph R. Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer (STTT)*, pages 212–232, 2004.
- [8] L. Cardelli. Type systems. *ACM Computing Surveys*, 28(1), 1996.
- [9] Markus Dahlweid, Michal Moskal, Thomas Santen, Stephan Tobies, and Wolfram Schulte. VCC: Contract-based Modular Verification of Concurrent C. Unpublished, 2008.
- [10] Stijn de Gouw and Michiel Helvensteijn. Mist – The Programming Language. Unpublished, 2008.
- [11] RObert DeLine and K. Rustan M. Leino. BoogiePL: A typed procedural language for checking object-oriented programs. Technical report, Microsoft, 2005.
- [12] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.
- [13] W. Drabent and J. Maluszynski. Inductive assertion method for logic programs. *TCS*, 59(1):133–155, 1988.
- [14] R.B. Findler and M. Felleisen. Contract soundness for object-oriented languages. *ACM SIGPLAN Notices*, 36(11):1–15, 2001.
- [15] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
- [16] James J. Gibson. The Theory of Affordances. *Perceiving, Acting, and Knowing*, 1977.
- [17] D. Harel, D. Kozen, J. Tiuryn, and I.F.R. Concepts. Dynamic logic.

- [18] CAR Hoare. An axiomatic basis for computer programming. 1969.
- [19] CAR Hoare. Procedures and parameters: An axiomatic approach. In *Symposium on Semantics of Algorithmic Languages*, volume 188, pages 102–116. Springer, 1971.
- [20] T. Hoare. The verifying compiler: A grand challenge for computing research. *Lecture notes in computer science*, pages 262–272, 2003.
- [21] International Organization for Standardization. *ISO/IEC 14882:2003*, second edition, 2003.
- [22] James H. Morris. Comments on "procedures and parameters". Undated and Unpublished.
- [23] C.B. Jones. *Systematic software development using VDM*. Prentice-Hall Englewood Cliffs, NJ, 1986.
- [24] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, March 2004.
- [25] Laura Kovcs. Reasoning Algebraically About P-Solvable Loops. In *Tools and Algorithms for the Construction and Analysis of Systems*, 2008.
- [26] E.R. Olderog. On the notion of expressiveness and the rule of adaptation. *Theoretical Computer Science*, 24(3):337–347, 1983.
- [27] B. Serpette, J. Vuillemin, and J.C. Herve. BigNum: a portable and efficient package for arbitrary-precision arithmetic. *Research report*, 2.
- [28] Thomas Kleymann. *Hoare logic and VDM: Machine-checked soundness and completeness proofs*. PhD thesis, University of Edinburgh, 1998.