Mist: A Verified Programming Language

Michiel Helvensteijn
CWI, Amsterdam, The Netherlands
LIACS, Leiden University, The Netherlands
michiel.helvensteijn@cwi.nl

Stijn de Gouw
CWI, Amsterdam, The Netherlands
LIACS, Leiden University, The Netherlands
cdegouw@cwi.nl

We introduce Mist, a new programming language designed to be used in conjunction with a verifying compiler. A language designed on such a premise has several advantages over other languages with regard to expressiveness, correctness and implementation. This paper discusses such advantages as controlled nondeterminism, more sophisticated assertions, more flexible type-systems and new opportunities for verification and optimization available to the compiler.

1 Introduction

Programming is among those processes most prone to human error. A rigorous method to ensure the absence of bugs is formal verification of desired properties. In 2003, Hoare formulated the grand challenge of the verifying compiler [6]. Such a compiler would be able to statically verify consistency of a program against its formal specification, which may include both functional and non-functional properties. Ideally, it would also be able to exploit the redundant information for optimization.

Several tools take steps towards a realisation of this goal. Static verification tools / environments such as KeY [1], VCC [3], FramaC [8] and SIMPL [9] are able to verify correctness properties of programs. However, these tools are not involved in program compilation, so they cannot assist in optimization. Nor can they use the specification to generate runtime checks in case a proof cannot be found.

As explained by Hoare [6], existing legacy code and legacy programming languages may not be the best candidates for verified compilation anyway. Much legacy code is not of sufficient quality or abstraction to benefit from rigorous specification and analysis. And the majority of existing programming languages do not natively support behavioral specification or redundant annotation. And most that do (e.g., Eiffel, Spec#, JML) are not exploiting the paradigm fully. That is, the basic language will be quite conventional, it will just include a specification language.

We propose a new programming language called Mist [4, 5]. It is designed from the ground up to be used in conjunction with a verifying compiler. While such a compiler does not yet exist, we work on the premise that continued improvement of verification software will make the concept feasible in the near future. The advantages go beyond verification of correctness. The language can afford to be more powerful and expressive, justified by extra (implicit) assertions on the program code. Such assertions may, for now, be translated to runtime checks. But as compilers evolve, more of them may be statically verified. A discussion of this follows in Section 3.1.

To address the undecidability of the program verification problem, some of the proof burden is shifted to the programmer in order to solve specific instances, i.e. if more information is required in order to complete a proof, the programmer will need to supply extra assertions in the code. Besides the prerequisite syntax for behavioral specification of functions[1], the Mist language also offers a sophisticated assertion mechanism. This results in program code that includes the information necessary to mechanically verify all steps of the required proofs. More on this in Section 2.2.

[1]We emphasize that in this text (and in Mist) we make no distinction between functions, methods and procedures. We use the word ‘function’ for all three.

Submitted to:
IPA PhD Workshop 2010

2 The Mist Language

The Mist programming language is imperative, statically scoped and strongly typed. Notably, it also follows the ‘design by contract’ paradigm, as introduced in Eiffel [7]. We plan for it to become a fully fledged application programming language, but the focus of our work is on automatic formal verification and related language constructs. The following subsections are by no means representative of the entire language, but rather highlight some relevant features.

2.1 Behavioral Specification

Function *behavioral specifications*, also called *contracts*, are a way to declaratively and formally specify the semantics of a function. It may consist of several clauses. A *precondition* specifies the conditions that are required to hold upon invoking the function. A *postcondition* specifies the conditions that are guaranteed to hold if and when the function terminates. A *termination clause* requires that the function always terminates in finite time. A *modification clause* lists the only non-local variables that the function is allowed to change. An empty modification clause, for instance, indicates a side-effect free function.

In the future, we plan to allow several other (non-functional) clauses in the specification. For example, clauses expressing worst-case time/space-complexity in some formal notation.

A contract not only *documents* but also *specifies* the behavior of a function. Its imperative code only suggests one possible runtime implementation. At the call-site, no assumptions are made of the function other than what the contract specifies. It is a powerful mechanism for controlled non-determinism.

However, we will also allow a programmer the choice of specifying a function by its code. This is reasonable for very small functions (e.g. methods only setting or getting a value) or in case the programmer wishes not to write contracts (i.e. use Mist as a normal imperative language). Note, however, that this can make formal reasoning about a function much more problematic.

2.2 Assertions

For the foreseeable future, computers cannot beat human creativity in the construction of formal proofs. In particular, many static truths about a program (e.g. that a loop terminates or maintains some invariant) cannot be found by a computer alone. Yet when presented with such information, a computer can sometimes verify it as true and then use it as a lemma in the construction of larger proofs.

Mist *assertions* are a construct for annotating code with such redundant information. They serve a dual purpose. They are used to express properties that the programmer wants verified, tested and/or documented. Secondly, assertions may guide proofs of correctness or validity of an optimization. Examples include *loop invariants* to prove correctness and *loop variants* and *recursive function variants* to prove termination. Other kinds of assertion[2] will be added in the future.

The compiler attempts to prove an assertion correct. If it is unable to, but cannot *disprove* it either, it is allowed to *assume* that it is correct and exploit it. If an assertion could not be statically verified, the compiler will always issue a warning and use the assertion to generate a runtime check (Section 3.1).

We believe that Mist-like assertions might change the way programmers write code in the future. Rather than alter the code so the compiler can understand or optimize it better (Section 3.2), decreasing its maintainability, he would add extra assertions. Assertions may be temporarily hidden by an IDE and may be removed completely if and when the compiler becomes smart enough to do without them.

[2]Our use of the term ‘assertion’ is appropriate, though somewhat untraditional in the world of programming languages. Classical assertions, such as those in C and Java, can only express a predicate over a state at some point in the code. Mist assertions can do that too, but can also express other kinds of constraints.

2.3 Type System

Most typesystems reject some programs that are correct, because when you require purely static type-checking, you have to cast a wide net. We can make the type system more selective by allowing checks at runtime (*soft typing* [2]), knowing that a verifying compiler could potentially eliminate the cost.

For instance, we might allow an expression of type T to be passed to an operation expecting some *subtype* of T. Of course, such an implicit down-cast is generally unsafe. Purely static type-checkers would not allow it, unless an explicit cast is used. In Mist, the operation would simply generate the implicit assertion that the argument will, at the call-site, evaluate to a value of the right subtype.

This scenario describes several well-known situations in programming. For instance, assigning a signed integer to an unsigned integer variable. Or dereferencing a pointer that might hold the `NULL` value. It is no different for division by an integer that might potentially be 0. Note that Mist does not actually offer a solution to these problems. It just puts them in terms of the assertion framework.

An interesting kind of user-defined subtype that requires this kind of flexibility is a base type constrained by an arbitrary unary predicate. For instance: `type eint <- int where ($ mod 2 = 0)`, the type of even integers (`$` indicates the predicate parameter).

Generally, Mist is based around values. Types are sets of values, working seamlessly with assertions and invariants. For instance, a function taking an `eint` parameter and another function taking an `int` parameter, but restricting it to the even integers in the precondition, would be behaviorally equivalent.

3 The Mist Compiler

Because a fully fledged verifying compiler does not yet exist, and because program verification is generally undecidable, the ability to statically verify all implicit and explicit assumptions of the code is always a variable. The compiler for the language would be designed with this in mind.

This section describes several compiler features in the context of this uncertainty.

3.1 Proving and Checking Assertions

Postconditions and assertions provided by the programmer (implicit and explicit) may be verified by the compiler, aided by a backend SMT solver. On request, the compiler may generate human readable proof-outlines that include the original code.

When a certain assertion could not be statically verified, it means the assertion is incorrect, the program contains a bug or the compiler is unable to construct the proof. The compiler should always issue a warning when this happens. If the compiler is able to find a counter-example, compilation should be aborted entirely. The programmer has the following options when he still believes the assertion holds:

- Provide additional assertions to guide the compiler to the proof. Possibly the compiler can prove them and use them as lemmas in order to verify the original assertion.
- Allow the compiler to generate a runtime check. This is inferior to a proof, but at least some bugs will trigger an assertion failure rather than allow the program to continue in an erroneous state.
- Explicitly suppress the runtime check. This should only be done if the check has prohibitive computational complexity and only after the programmer has used some other means of verification, such as exhaustive testing or a manual proof.

Additionally, newer versions of the compiler may include a more sophisticated theorem prover, able to verify assertions that older versions could not.

3.2 Optimization

Compared to standard compilers, a verifying compiler will have additional potentially powerful optimizations at its disposal. Examples include allocating less space for certain variables because of known constraints, dead-code elimination and code-movement.

But there are also more interesting optimizations. Consider a code-base that contains both the Merge Sort algorithm, which has low time complexity, and the Insertion Sort algorithm, which has low space complexity. If the programmer invokes Insertion Sort and the compiler is set to optimize for speed, it is free to substitute a call to Merge Sort, since Merge Sort has the same behavioral specification as Insertion Sort. It would not be allowed to substitute a call to Quick Sort, however. Being an unstable sorting algorithm, Quick Sort has a weaker postcondition than Insertion Sort.

In general, the compiler may replace any call to function f with a call to function g if g is a *refinement* of f. It implies that g has a weaker (or equivalent) precondition than f and a stronger (or equivalent) postcondition. A backend SMT solver would determine if the refinement relation holds.

4 Conclusion and Future Work

We are working on the Mist programming language, which is to be used in combination with a verifying compiler. We have already implemented a version of the Mist compiler for a very limited subset of the language. It is able to statically verify while-programs with recursive procedures [4, 5]. At the moment, it uses KeY [1] to prove generated verification conditions.

We are still working on the design of Mist and a formalization of the ideas presented in this abstract. We are also continuing work on the Mist compiler, allowing its verification framework to cover a larger subset of the language.

We hope that the existence of a language such as Mist, which could potentially offer great advantages to the software engineering industry, may motivate work on formal program verification and satisfiability solving, coming closer to a realisation of the verifying compiler as described in Hoare's grand challenge.

References

[1] Bernhard Beckert, Reiner Hähnle & Peter H. Schmitt, editors (2007): *Verification of Object-Oriented Software: The KeY Approach.* LNCS 4334. Springer-Verlag.

[2] R. Cartwright & M. Fagan (1991): *Soft typing.* In: *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, ACM, pp. 278–292.

[3] Markus Dahlweid, Michal Moskal, Thomas Santen, Stephan Tobies & Wolfram Schulte (2008): *VCC: Contract-based Modular Verification of Concurrent C.* Unpublished.

[4] Stijn de Gouw & Michiel Helvensteijn (2009): *Automatic Program Verification in Mist.* Master's thesis, Leiden University.

[5] Michiel Helvensteijn & Stijn de Gouw. *The design of and compiler for the Mist programming language.* `http://mist.googlecode.com` [online].

[6] Tony Hoare (2003): *The verifying compiler: A grand challenge for computing research.* *Lecture notes in computer science* , pp. 262–272.

[7] Bertrand Meyer (2005): *Eiffel as a Framework for Verification.* pp. 301–307.

[8] Benjamin Monate & Loïc Correnson. *Frama-C Software Analyzers.* `http://frama-c.cea.fr` [online].

[9] Norbert Schirmer (2006): *Verification of Sequential Imperative Programs in Isabelle/HOL.* Ph.D. thesis, Technische Universität München.