Whiley: a Platform for Research in Software Verification

David J. Pearce and Lindsay Groves

Victoria University of Wellington
Wellington, New Zealand
{djp,lindsay}@ecs.vuw.ac.nz

Abstract. An ongoing challenge for computer science is the development of a tool which automatically verifies programs meet their specifications, and are free from runtime errors such as divide-by-zero, array out-of-bounds and null dereferences. Several impressive systems have been developed to this end, such as ESC/Java and Spec#, which build on existing programming languages (e.g. Java, C#). However, there remains a need for an open research platform in this area. We have developed the Whiley programming language, and its accompanying verifying compiler, as an open platform for research. Whiley has been designed from the ground up to simplify the verification process. In this paper, we introduce the Whiley language and it accompanying verifying compiler tool.

1 Introduction

Prof. Sir Tony Hoare (ACM Turing Award Winner, FRS) proposed the creation of a *verifying compiler* as a grand challenge for computer science [1]. A verifying compiler "*uses automated mathematical and logical reasoning to check the correctness of the programs that it compiles.*" There have been numerous attempts to construct a verifying compiler system, although none has yet made it into the mainstream. Early examples include that of King [2], Deutsch [3], the Gypsy Verification Environment [4] and the Stanford Pascal Verifier [5]. More recently, the Extended Static Checker for Modula-3 [6] which became the Extended Static Checker for Java (ESC/Java) — a widely acclaimed and influential work [7]. Building on this success was JML and its associated tooling which provided a standard notation for specifying functions in Java [8]. Finally, Microsoft developed the Spec# system which is built on top of C# [9].

Both ESC/Java and Spec# build on existing object-oriented languages (i.e. Java and C#) but, as a result, suffer numerous limitations. The problem is that such languages were not designed for use with verifying compilers. Ireland, in his survey on the history of verifying compilers, noted the following [10]:

> *"The choice of programming language(s) targeted by the verifying compiler will have a significant effect on the chances of success."*

Likewise, a report on future directions in verifying compilers, put together by several researchers in this area, makes a similar comment [11]:

> *"Programming language design can reduce the cost of specification and verification by keeping the language simple, by automating more of the work, and by eliminating common errors."*

This paper introduces Whiley, a programming language designed from scratch in conjunction with a verifying compiler. The intention of this is to provide an open framework for research in automated software verification. The initial goal is to automatically eliminate common errors, such as *null dereferences*, *array-out-of-bounds*, *divide-by-zero* and more. In the future, the intention is to consider more complex issues, such as termination, proof-carrying code and user-supplied proofs. Finally, several works have already been published which focus primarily on Whiley's type system [12–14].

The Tool. The main tool underlying Whiley is the verifying compiler. This is been in development for over three years, and has become a large (and relatively mature) code base. Numerous student projects have been conducted already based on this compiler, and the hope is to use it for teaching next year. The compiler is released under an open source license, can be downloaded from `http://whiley.org` and forked at `http://github.com/DavePearce/Whiley/`. Some interesting statistics are available from `http://http://www.ohloh.net/p/whiley` and a fun demonstration on writing loop invariants is available here: `http://www.youtube.com/watch?v=WwnxHugabrw`. Finally, a prototype Eclipse plugin is available and can be installed via the update site: `http://whiley.org/eclipse`.

2 Language Core

We begin by exploring the Whiley language and highlight some of the choices made in its design. For now, we stick to the basic issues of syntax, semantics and typing and, in the following section, we will focus more specifically on using Whiley for verification. Perhaps one of our most important goals was to make the system as accessible as possible. To that end, the language was designed to superficially resemble modern imperative languages (e.g. Python), and this decision has significantly affected our choices.

Overview. Languages like Java and C# permit arbitrary side-effects within methods and statements. This presents a challenge when such methods may be used within specifications. Systems like JML and Spec# require that methods used in specifications are *pure* (i.e. side-effect free). An important challenge here is the process of checking that a function is indeed pure. A significant body of research exists on checking functional purity in object-oriented languages (e.g. [15, 16]). Much of this relies on interprocedural analysis, which is too costly for a verifying compiler. To address this, Whiley is a hybrid object-oriented and functional language which divides into *a functional core* and an *imperative outer layer*. Everything in the functional core can be modularly checked as being side-effect free. To make this possible, Whiley incorporates first-class sets, lists and maps which are *values* (rather than mutable objects) and, hence, allow call-by-value semantics (more on this later).

Flow Typing. An unusual feature of Whiley is the use of a *flow typing system* (see e.g. [17, 18, 13, 14]). This gives Whiley the look-and-feel of a dynamically typed language (e.g. Python). Furthermore, automatic variable retyping through conditionals is supported using the **`is`** operator (similar to `instanceof` in Java) as follows:

```
define Circle as {int x, int y, int radius}
define Rect as {int x, int y, int width, int height}
define Shape as Circle | Rect

real area(Shape s):
    if s is Circle:
        return PI * s.radius * s.radius
    else:
        return s.width * s.height
```

A `Shape` is either a `Rect` or a `Circle` (which are both record types). The type test "`s is Circle`" determines whether `s` is a `Circle` or not. Unlike Java, Whiley automatically retypes `s` to have type `Circle` (resp. `Rect`) on the true (resp. false) branches of the **`if`** statement. There is no need to explicitly cast variable `s` to the appropriate `Shape` before accessing its fields.

Union Types. Another unusual feature of Whiley is the use of *union types* (see e.g. [19, 20]), which complement the flow type system. Consider the following example:

```
null|int indexOf(string str, char c):
   ...

[string] split(string str, char c):
   idx = indexOf(str,c)
   // idx has type null|int
   if idx is int:
       // idx now has type int
       below = str[0..idx]
       above = str[idx..]
       return [below,above]
   else:
       // idx now has type null
       return [str]
```

Here, `indexOf()` returns the first index of a character in the string, or **`null`** if there is none. The type **`null`**`|`**`int`** is a union type, meaning it is either an **`int`** *or* **`null`**. The system seamlessly ensures **`null`** is never dereferenced because the type **`null`**`|`**`int`** cannot be treated as an **`int`**. Instead, one must first check it *is* an **`int`** using e.g. "`idx is int`".

Recursive Data Types. Whiley provides recursive types which are similar to the abstract data types found in functional languages (e.g. Haskell, ML, etc). For example:

```
define LinkedList as null | {int data, LinkedList next}

int length(LinkedList l):
  if l is null:
    return 0 // l now has type null
  else:
    return 1 + length(l.next) // l now has type {int data, LinkedList next}
```

Here, we again see how flow typing gives an elegant solution. More specifically, on the false branch of the type test "`l is null`", variable `l` is automatically retyped to `{int data, LinkedList next}` — thus ensuring the subsequent dereference of `l.next` is safe. No casts are required as would be needed for a conventional imperative language (e.g. Java). Finally, like all compound structures, the semantics of Whiley dictates that recursive data types are passed by value (or, at least, appear to be from the programmer's perspective).

Value Semantics. The prevalence of pointers — or references — in modern programming languages (e.g. Java, C++, C#) has been a major hindrance in the development of verifying compilers. Indeed, Mycroft recently argued that (unrestricted) pointers should be "considered harmful" in the same way that Dijkstra considered goto harmful [21]. To address this, all compound structures in Whiley (e.g. lists, sets, and records) have *value semantics*. This means they are passed and returned by-value (as in Pascal, MATLAB or most functional languages). But, unlike functional languages (and like Pascal), values of compound types can be updated in place. Whilst this latter point may seem unimportant, it serves a critical purpose: to give Whiley the appearance of a modern *imperative* language when, in fact, the functional core of Whiley is pure. This goes towards our goal of making the language as accessible as possible.

Value semantics implies that updates to a variable only affect that variable, and that information can only flow out of a function through its return value. Consider:

```
int f([int] xs):
    ys = xs
    xs[0] = 1
    ...
```

The semantics of Whiley dictate that, having assigned `xs` to `ys` as above, the subsequent update to `xs` does not affect `ys`. Arguments are also passed by value, hence `xs` is updated inside `f()` and this does not affect `f`'s caller. That is, `xs` is not a *reference* to a list of **`int`**; rather, it *is* a list of **`int`**s and assignments to it do not affect state visible outside of `f()`.

Unbound Arithmetic. Modern languages typically provide fixed-width numeric types, such as 32bit twos-compliment integers, or 64-bit IEEE 754 floating point numbers. Such data types are notoriously difficult for an automated theorem prover to reason about [22]. Systems like JML and Spec# assume (unsoundly) that numeric types do not overflow or suffer from rounding. To address this, Whiley employs *unbounded integers* and *rationals* in place of their fixed-width alternatives and, hence, does not suffer the limitations of soundness discussed above.

Performance. Many of our choices (e.g. value semantics and unbound arithmetic) have a potentially detrimental effect on performance. Whilst this is a trade-off we accept, there are existing techniques which can help. For example, using reference counting to minimise unnecessary cloning of compound structures (see e.g. [23]); and, integer range analysis (see e.g. [24]) to place variables into native data types where possible.

3 Verification

The key goal of the Whiley project is to develop an open framework for research in automated software verification. As such, we now explore verification in Whiley.

Example 1 — Constrained Types. The following Whiley code defines a function accepting a positive integer and returning a non-negative integer (i.e. natural number):

```
int f(int x) requires x > 0, ensures $ >= 0 && $ != x:
    return x-1
```

Here, the function `f()` includes a **`requires`** and **`ensures`** clause which correspond (respectively) to its *pre-condition* and *post-condition*. In this context, $ represents the return value, and must be used in the **`ensures`** clause. The Whiley compiler statically verifies that this function meets its specification.

The above illustrates a function specification given through explicit pre- and post-conditions. However, we may also employ *constrained types* to simplify it as follows:

```
define nat as int where $ >= 0
define pos as int where $ > 0

nat f(pos x) ensures $ != x:
    return x-1
```

Here, the **`define`** statement includes a **`where`** clause constraining the permissible values for the type ($ represents the variable whose type this will be). Thus, `nat` defines the type of non-negative integers (i.e. the natural numbers). Likewise, `pos` gives the type of positive integers and is implicitly a subtype of `nat` (since the constraint on `pos` implies that of `nat`). We consider that good use of constrained types is critical to ensuring that function specifications remain as readable as possible.

The notion of type in Whiley is more fluid than found in typical languages. In particular, if two types $\mathtt{T}_1$ and $\mathtt{T}_2$ have the same *underlying* type, then $\mathtt{T}_1$ is a subtype of $\mathtt{T}_2$ iff the constraint on $\mathtt{T}_1$ implies that of $\mathtt{T}_2$. Consider the following:

```
define anat as int where $ >= 0
define bnat as int where 2*$ >= $

bnat f(anat x):
    return x
```

In this case, we have two alternate (and completely equivalent) definitions for a natural number (we can see that `bnat` is equivalent to `anat` by subtracting $ from both sides). The Whiley compiler is able to reason that these types are equivalent and statically verifies that this function is correct.

Example 2 — Implicit Retyping. Variables in Whiley are described by their underlying type and those constraints which are shown to hold. As the automated theorem prover learns more about a variable, it automatically takes this into consideration when checking constraints are satisfied. For example:

```
define nat as int where $ >= 0

nat abs(int x):
    if x >= 0:
        return x
    else:
        return -x
```

The Whiley compiler statically verifies that this function always returns a non-negative integer. This relies on the compiler to reason correctly about the implicit constraints implied by the conditional. A similar, but slightly more complex example is that for computing the maximum of two integers:

```
int max(int x, int y) ensures $ >= x && $ >= y
                                 && ($==x || $==y):
    if x > y:
        return x
    else:
        return y
```

Again, the Whiley compiler statically verifies this function meets its specification. Here, the body of the function is almost completely determined by the specification — however, in general, this not the case.

Example 3 — Bounds Checking. An interesting example which tests the automated theorem prover more thoroughly is the following:

```
null|int indexOf(string str, char c):
    for i in 0..|str|:
        if str[i] == c:
            return i
    return null
```

In this case, the access `str[i]` must be shown as within the bounds of the list `str`. The Whiley compiler statically verifies this is true and, hence, that `indexOf()` cannot cause an out-of-bounds error.

Example 4 — Loop Invariants. Another example illustrates the use of *loop invariants* in Whiley:

```
define natlist as [int] where all { x in $ | x >= 0 }

int sum(natlist list) ensures $>=0:
    r = 0
    for v in list where r >= 0:
        r = r + v
    return r
```

Here, bounded quantifiers are used to define a list of natural numbers which is accepted by the `sum()` function. Equivalently, we could have used `[nat]` (with `nat` defined as before) — and these two alternative definitions of the same concept are, in a strong sense, identical.

A key constraint is that summing a list of natural numbers yields a natural number (recall arithmetic is unbounded and does not overflow in Whiley). The Whiley compiler statically verifies that `sum()` does indeed meet this specification. The loop invariant is necessary to help the compiler generate a sufficiently powerful verification condition to prove the function meets the post condition. In the future, we hope to automatically synthesize simple loop invariants such as this.

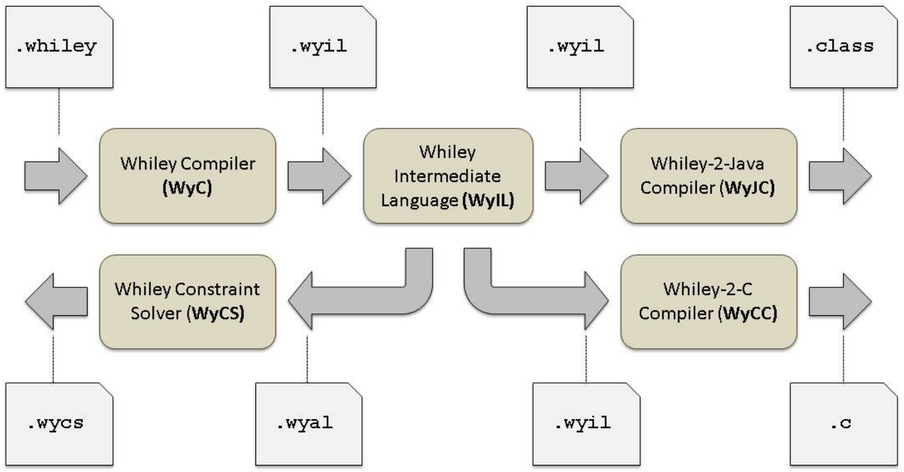
.whiley
.wyil
.wyil
.class
Whiley Compiler (WyC)
Whiley Intermediate Language (WyIL)
Whiley-2-Java Compiler (WyJC)
Whiley Constraint Solver (WyCS)
Whiley-2-C Compiler (WyCC)
.wycs
.wyal
.wyil
.c

Fig. 1. Illustrating the compilation and verification pipeline.

4 Compiler Architecture

The Whiley verifying compiler is structured as a number of distinct modules. This has proved invaluable for keeping a clear separation of concerns between the major components, and for testing and debugging — since many modules can be tested in isolation from others. The main modules of the verifying compiler are:

- **Whiley Build System (WyBS).** Responsible for information flow throughout the compiler, managing source and binary roots, and determining compilation orders.
- **Whiley Compiler (WyC).** Responsible for parsing and type checking `whiley` source files, and compiling them into binary `wyil` files.
- **Whiley Intermediate Language (WyIL).** A register-based intermediate language similar to Java bytecode along with an accompanying binary file format.
- **Whiley-2-Java Compiler (WyJC).** A back-end which converts `wyil` files into JVM `class` files.
- **Whiley-2-C Compiler (WyCC).** An experimental back-end which converts `wyil` files into C source files.
- **Whiley Constraint Solver (WyCS).** An automated theorem prover responsible for accepting input files in a variant of first-order logic called the *Whiley Assertion Language (WyAL)* and verifying they are correct.

Figure 1 provides an overview of the flow of information within the compiler. Here we see that `whiley` source files are converted into (binary) `wyil` files; in turn, these are converted into binary `class` files (for execution) and `wyal` source files (for verification). The latter is, in turn, converted into the more concise binary `wycs` form. Note that, in the general course of events, not all of these files are physically produced. For example, when compiling a `whiley` source file into a `class` file, no other files are written to disk (unless specifically requested).

From Figure 1, we see that a strong emphasis has been placed on the use of different file formats. Whilst this may seem overly complex, it helps the testing and debugging process significantly. For example, consider diagnosing a bug presenting as a `whiley` source file that incorrectly verifies. There are numerous places within the compiler which could be causing the problem. For example, it could be a problem with the translation of the `whiley` source to the `wyil` file. Likewise, it could be a problem with the *verification condition generator* which generates `wyal` files from `wyil` files. In debugging this, one can generate each of these files and inspect them individually to identify the misbehaving module; furthermore, one can modify any of these files and push them back into the pipeline to see the effect. Likewise, each module can be tested in isolation of others by providing tests written in its given input format.

Another advantage of the modularisation in the verifying compiler, is that it enables interesting possibilities for reuse. For example, other researchers could build a front-end for a different language and compile down to our intermediate language — thereby gaining the ability to verify their programs for free. Likewise, other researchers developing their own verifying compiler with a different intermediate representation might still generate verification conditions in the `wyal` format and reuse our theorem prover. Similarly, we can e.g. replace the WyCS theorem prover with another (e.g. Z3 [25] or Simplify [26]) by writing a wrapper which converts files in the `wyal` format into the appropriate input language of the external tool[1].

4.1 Intermediate Language

The *Whiley Intermediate Language (WyIL)* is a register-based intermediate language which resembles Java Bytecode. The following illustrates a Whiley function (left) and the corresponding WyIL code (right):

```
int abs(int x) ensures $ >= 0:
    if x >= 0:
        return x
    else:
        return -x
```

```
int abs(int):
ensures:
  const %3 = 0 : int
  assertge %0,%3 "..." : int
body:
  const %2 = 0 : int
  iflt %0,%2 goto label0 : int
  return %0 : int
.label0
  neg %5 = %0 : int
  return %5 : int
```

As can be seen from above, every WyIL bytecode is associated with a type. Furthermore, registers are prefixed with `%` (e.g. `%3`); the `const` bytecode loads a constant value into a register; the `iflt` bytecode branches to a label if its first operand is less than its second; the `neg` bytecode negates its operand and assigns to a given register; finally, the `return` bytecode returns its operand.

[1] And, indeed, a student project run this year has been investigating doing exactly this.

4.2 Assertion Language

The *Whiley Assertion Language* is a dialect of first-order logic with various additional theories (e.g. for arithmetic, sets, lists, etc). A given `wyil` file will generate a single `wyal` file that may contain numerous assertions. The following illustrates the two assertions generated from our example above (written side-by-side to conserve space):

```
assert "...":                  | assert "...":
    forall(int r0):            |     forall(int r0):
        if:                    |         if:
            r0 >= 0            |             r0 < 0
        then:                  |         then:
            r0 >= 0            |             -r0 >= 0
```

Here, the left assertion corresponds to the execution path through the true branch of the **`if`** statement in the original Whiley function; likewise, the right assertion corresponds to the path through the false branch. Finally, each **`assert`** statement is given a message to report if it is found to be invalid (note, these are elided above for brevity).

4.3 Build System

The *Whiley Build System (WyBS)* controls the overall flow of information within the compiler. Every build operates over a *project* which contains one or more *source roots*, and one or more corresponding *binary roots*. A source root gives the root location of a Whiley package (e.g. a file system directory). A binary root indicates where binary files should be located (e.g. a file system directory, a `jar` file, etc). Observe that some binary files (e.g. `wyil`) are written during compilation, but may also be read (e.g. from the standard library). A key design feature is that roots may be *virtual* — meaning they are not written physically to disk. A command-line option can then determine whether or not a given root should be virtual (i.e. whether or not a given set of files need to be physically generated). A further advantage of this approach is that it aids integration with other tools (e.g. `ant`, `eclipse`, etc). For example, `eclipse` maintains its own filesystem representation and, hence, integrating our compiler requires integrating with this. In fact, this was straightforward: we simply created a range of `root` classes which interface with `eclipse` and replace those used by the stand-alone compiler.

5 Conclusion

In this paper, we have presented the Whiley language and its accompanying verifying compiler tool. Our goal is to provide an open framework for research in automated software verification, and work continues on this front.

References

1. Tony Hoare. The verifying compiler: A grand challenge for computing research. *Journal of the ACM*, 50(1):63–69, 2003.
2. S. King. *A Program Verifier*. PhD thesis, Carnegie-Mellon University, 1969.

3. L. Peter Deutsch. *An interactive program verifier*. Ph.d., 1973.
4. D. I. Good. Mechanical proofs about computer programs. In *Mathematical logic and programming languages*, pages 55–75, 1985.
5. D. C. Luckham, S. M. German, F. W. von Henke, R. A. Karp, P. W. Milne, D. C. Oppen, W. Polak, and W. L. Scherlis. Stanford pascal verifier user manual. Technical Report CS-TR-79-731, Stanford University, Department of Computer Science, 1979.
6. David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. SRC Research Report 159, Compaq Systems Research Center, 1998.
7. Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *Proc. PLDI*, pages 234–245, 2002.
8. G. T. Leavens, Y. Cheon, C. Clifton, C. Ruby, and D. R. Cok. How the design of JML accommodates both runtime assertion checking and formal verification. *Science of Computer Programming*, 55(1-3):185–208, March 2005.
9. Mike Barnett, K. Rustan, M. Leino, and Wolfram Schulte. The spec# programming system: An overview. Technical report, Microsoft Research, 2004.
10. A. Ireland. A Practical Perspective on the Verifying Compiler Proposal. In *Proceedings of the Grand Challenges in Computing Research Conference*, 2004.
11. G. T. Leavens, J. Abrial, D. Batory, M. Butler, A. Coglio, K. Fisler, E. Hehner, C. Jones, D. Miller, S. Peyton-Jones, M. Sitaraman, D. R. Smith, and A. Stump. Roadmap for enhanced languages and methods to aid verification. In *Proc. of GPCE*, pages 221–235, 2006.
12. D. Pearce and J. Noble. Implementing a language with flow-sensitive and structural typing on the JVM. *Electronic Notes in Computer Science*, 279(1):47–59, 2011.
13. D. J. Pearce. Sound and complete flow typing with unions, intersections and negations. In *Proc. VMCAI*, pages 335–354, 2013.
14. D. J. Pearce. A calculus for constraint-based flow typing. In *Proc. FTFJP*, page (to appear), 2013.
15. Atanas Rountev. Precise identification of side-effect-free methods in java. In *Proc. ICSM*, pages 82–91. IEEE Computer Society, 2004.
16. A. Salcianu and M. Rinard. Purity and side effect analysis for Java programs. In *Proc. VMCAI*, pages 199–215, 2005.
17. Sam Tobin-Hochstadt and Matthias Felleisen. Logical types for untyped languages. In *Proc. ICFP*, pages 117–128, 2010.
18. A. Guha, C. Saftoiu, and S. Krishnamurthi. Typing local control and state using flow analysis. In *Proc. ESOP*, pages 256–275, 2011.
19. Franco Barbanera and Mariangiola Dezani-Cian Caglini. Intersection and union types. In *In Proc. TACS*, pages 651–674, 1991.
20. Atsushi Igarashi and Hideshi Nagira. Union types for object-oriented programming. *Journal of Object Technology*, 6(2), 2007.
21. Alan Mycroft. Programming language design and analysis motivated by hardware evolution. In *Proc. SAS*, pages 18–13. Springer-Verlag, 2007.
22. R. E. Bryant, D. Kroening, J. Ouaknine, S. A. Seshia, O. Strichman, and B. A. Brady. Deciding bit-vector arithmetic with abjc10straction. In *Proc. TACAS*, pages 358–372, 2007.
23. Nurudeen Lameed and Laurie J. Hendren. Staged static techniques to efficiently implement array copy semantics in a MATLAB JIT compiler. In *Proc. CC*, pages 22–41, 2011.
24. Flemming Nielson, Hanne R. Nielson, and Chris L. Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.
25. Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Proc. TACAS*, pages 337–340, 2008.
26. Detlefs, Nelson, and Saxe. Simplify: A theorem prover for program checking. *JACM*, 52, 2005.