Science of Computer Programming ••• (••••) •••–•••

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Designing a verifying compiler: Lessons learned from developing Whiley

David J. Pearce *, Lindsay Groves

School of Engineering and Computer Science, Victoria University of Wellington, New Zealand

ARTICLE INFO

Article history:
Received 30 April 2014
Received in revised form 27 September 2015
Accepted 29 September 2015
Available online xxxx

Keywords:
Program verification
Loop invariants
Hoare logic
Verification tools

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#). We have been developing a programming language from scratch to simplify verification, called Whiley, and an accompanying verifying compiler. In this paper, we present a technical overview of the verifying compiler and document the numerous design decisions made. Indeed, many of our decisions reflect those of similar tools. However, they have often been ignored in the literature and/or spread thinly throughout. In doing this, we hope to provide a useful resource for those building verifying compilers.

1. Introduction

The idea of verifying that a program meets a given specification for all possible inputs has been studied for a long time. Hoare's Verifying Compiler Grand Challenge was an attempt to spur new efforts in this area to develop practical tools [1]. According to Hoare's vision, a verifying compiler "*uses automated mathematical and logical reasoning to check the correctness of the programs that it compiles*" [1]. Hoare's intention was that verifying compilers should fit into the existing development tool chain, "*to achieve any desired degree of confidence in the structural soundness of the system and the total correctness of its more critical components*". For example, commonly occurring errors could be automatically eliminated, such as: *division-by-zero*, *integer overflow*, *buffer overruns* and *null dereferences*.

The first systems that could be reasonably considered as verifying compilers were developed some time ago, and include that of King [2], Deutsch [3], the Gypsy Verification Environment [4] and the Stanford Pascal Verifier [5]. Following on from these was the Extended Static Checker for Modula-3 [6]. Later, this became the Extended Static Checker for Java (ESC/Java) — a widely acclaimed and influential work in this area [7]. Building on this success was the Java Modelling Language (and its associated tooling) which provided a standard notation for specifying functions in Java [8,9]. Since then a variety of other tools have blossomed in this space, including Spec# [10,11], Dafny [12,13], Why3 [14] and VeriFast [15,16].

Much of the research in verifying compilers has targeted established languages (e.g., Java, C, C#, etc.). Unfortunately, such languages were not designed for verification and contain numerous problematic features, including: fixed-width number representations [17,18], unrestricted pointers [19], arbitrary side-effects [20], closures [21] and flexible threading mod-

* Corresponding author.
E-mail addresses: djp@ecs.vuw.ac.nz (D.J. Pearce), lindsay@ecs.vuw.ac.nz (L. Groves).

http://dx.doi.org/10.1016/j.scico.2015.09.006
0167-6423/

els [22,23]. The alternative, of course, is to design programming languages from scratch specifically for this purpose. Barnett et al. argue this "*lets a language designer pick features that mesh well with verification*" [10].

In this vein, we have developed a programming language from scratch, called Whiley, and an accompanying verifying compiler [24–28]. Whiley is an imperative language designed primarily to simplify verification. The project's goals are:

1. **Ease of Use.** An important goal is to develop a system which is as accessible as possible, and which one could imagine being used in a day-to-day setting. To that end, the language was designed to superficially resemble modern imperative languages (e.g., Python). To simplify verification, Whiley employs unbounded integer and rationals rather than fixed-width arithmetic. Likewise, to further reduce programmer burden, simple loop invariants are inferred where possible.
2. **Scope.** Another important goal of the project is to enable programs to be automatically verified as: *correct with respect to their declared specifications*; and, *free from runtime error* (e.g., divide-by-zero, array index-out-of-bounds, etc.). However, more complex properties such as *termination*, *worst-case execution time*, *worst-case stack depth*, etc. are not considered (although would be interesting future work). This impacts the language design as, for example, we do not provide syntax for expressing loop variants (i.e., establishing termination is not a consideration).
3. **Demonstration.** Another important goal of the project is to demonstrate that Whiley is suitable for developing safety-critical systems. This means, for example, that Whiley programs must be executable and can be integrated into existing systems (i.e., via a foreign function interface). To this end, we are initially targeting small embedded systems (see Section 8).

Many of these goals are seemingly at odds with each other. For example, unbound arithmetic is not suitable for embedded systems. In many cases, we can work around this by exploiting function specifications as these provide rich information [29]. When compiling for an embedded device we can, for example, require all integers used within a function are bounded using appropriate invariants (i.e., that they are required to be within certain ranges, etc.). Thus, the difficulty of verification is layered. That is, users need not initially worry about bounded arithmetic and, instead, can focus on learning and understanding the verification process. Later on, if they wish to target an embedded system, they must then further refine their specifications to allow this. To these ends, we have successfully compiled Whiley programs to run on a quadcopter (see Section 8). Likewise, we have used Whiley to teach students at Victoria University of Wellington about verification (see Section 8). Finally, the Whiley verifying compiler is released under an open source license (BSD), and can be downloaded from http://whiley.org and forked at http://github.com/Whiley/. All examples in this paper have been tested against the latest release at the time of writing (version 0.3.36).

1.1. Contribution

The primary contribution of this paper lies in documenting numerous important issues faced in developing a verifying compiler and the decisions we made. Many of our choices reflect those of similar systems such as Dafny, Spec#, ESC/Java, etc. Such decisions are often left undocumented, or spread throughout numerous papers in the literature. By bringing these together in one place we hope to provide a useful resource for those designing and implementing verifying compilers. Note, however, this paper does not attempt to evaluate our language design against those goals outlined above, and this remains as future work.

Finally, an earlier version of this paper was published at the Workshop on Formal Techniques for Safety-Critical Systems (FTSCS'13) [30]. We present here a significantly revised and extended version of that paper, which includes additional discussion of our experiences, a small evaluation against benchmarks from the COST'11 [31] and VSCOMP'10 [32] verification competitions and three case studies looking at Whiley in the context of teaching, embedded systems and alternative SMT solvers.

1.2. Organisation

We provide a general introduction to verification with Whiley in Section 2, followed by an overview of the compiler architecture in Section 3. In Section 4, we provide a detailed discussion of the salient design choices made and their implications in practice. Then, in Section 5 and Section 6 we reflect on experiences gained from verifying programs with Whiley. Following this is a small evaluation of Whiley's verification capability in Section 7 and a discussion of three case studies in Section 8. Finally, related work is discussed in Section 9 before we conclude in Section 10.

2. Language overview

In this section, we introduce the Whiley language in the context of software verification through a series of examples. However, we do not provide an exhaustive examination of the language and, instead, the interested reader may find more detailed introductions elsewhere [33,34].

Please cite this article in press as: D.J. Pearce, L. Groves, Designing a verifying compiler: Lessons learned from developing Whiley, Sci. Comput. Program. (2015), http://dx.doi.org/10.1016/j.scico.2015.09.006

2.1. Example 1 — preconditions and postconditions

Whiley allows explicit *pre-* and *post-conditions* to be given for functions. For example, the following function accepts a positive integer and returns a natural number:

```
function decrement(int x) -> (int y)
// Parameter x must be greater than zero
requires x > 0
// Return must be greater or equal to zero
ensures y >= 0:
    //
    return x - 1
```

Here, the function `decrement()` includes **`requires`** and **`ensures`** clauses which correspond (respectively) to its *precondition* and *postcondition*. In this context, `y` represents the return value and may be used only within the **`ensures`** clause. The Whiley compiler statically verifies that this function meets its specification (recall that arithmetic in Whiley is unbounded and, hence, cannot underflow). We can see this for ourselves by manually constructing an appropriate *verification condition* (that is, a logical condition whose truth establishes that the implementation meets its specification). In this case, the appropriate verification condition is $\forall x \in \texttt{int}.(x > 0 \implies x - 1 \geq 0)$ — which by inspection we can see holds. In general, verification conditions are typically too large to easily verify by hand and, instead, Whiley relies on an automated theorem prover to discharge them.

2.2. Example 2 — conditionals

The Whiley compiler reasons about functions by exploring the different control-flow paths through their bodies. Furthermore, as it learns more about the variables used in the function, it automatically takes this into account. For example:

```
function abs(int x) -> (int y)
// Return value cannot be negative
ensures y >= 0:
    //
    if x >= 0:
        return x
    else:
        return -x
```

The Whiley compiler statically verifies that this function always returns a non-negative integer. To do this, the compiler must reason correctly about the implicit constraints implied by the conditional. In this case, one verification condition is generated for each execution path, giving: $\forall x \in \texttt{int}.(x \geq 0 \implies x \geq 0)$ for the true branch; and, $\forall x \in \texttt{int}.(x < 0 \implies -x \geq 0)$ for the false branch.

A similar, but slightly more complex example, is that for computing the maximum of two integers:

```
function max(int x, int y) -> (int z)
// Must return either x or y
ensures x == z || y == z
// Return must be as large as x and y
ensures x <= z && y <= z:
    //
    if x > y:
        return x
    else:
        return y
```

In this case, multiple **`ensures`** clauses are given which are conjoined together to form the function's postcondition. We have found that allowing multiple **`ensures`** clauses can help readability, and note that JML [9], Spec# [10] and Dafny [12] also permit this. Furthermore, multiple **`requires`** clauses are permitted in the same manner. Again, the Whiley compiler statically verifies this function meets its specification.

2.3. Example 3 — data type invariants

In Whiley, we may also provide invariants over data types, as the following illustrates:

Please cite this article in press as: D.J. Pearce, L. Groves, Designing a verifying compiler: Lessons learned from developing Whiley, Sci. Comput. Program. (2015), http://dx.doi.org/10.1016/j.scico.2015.09.006

```
// A natural number is an integer greater-than-or-equal-to zero
type nat is (int n) where n >= 0

// A positive number is an integer greater-than zero
type pos is (int p) where p > 0

function f(pos x) -> (nat n)
// Return must differ from parameter
ensures n != x:
    //
    return x - 1
```

Here, the **type** declaration includes a **where** clause constraining the permissible values for the type. The declared variable (e.g., `n` or `p`) is used to represent an arbitrary value of the given type. Thus, `nat` defines the type of natural numbers. Likewise, `pos` gives the type of positive integers. 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 as variables can move seamlessly between types. 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 $\mathtt{T}_2$. For example, `pos` is a subtype of `nat` as the constraint on `pos` implies that of `nat`. As another example, consider the following:

```
type anat is (int x) where x >= 0
type bnat is (int x) where 2*x >= x

function f(anat x) -> bnat:
    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 `x` from both sides). The Whiley compiler is able to reason that these types are equivalent and statically verifies that this function meets its specification.

2.4. Example 4 — bounds checking

Whiley automatically eliminates simple runtime errors by ensuring, for example, array accesses are within bounds, divisors are non-zero, etc. The following illustrates:

```
function get(int[] items, int i) -> int:
    if i >= 0 && i < |items|:
        return items[i]
    else:
        return 0
```

In this case, the access `items[i]` must be shown as within the bounds of the array `items`. The conditional guards against this and, hence, the Whiley compiler statically verifies this function cannot cause an out-of-bounds error.

2.5. Example 5 — loop invariants

Whiley supports explicit loop invariants which are necessary to prove many useful properties about programs with loops. The following illustrates:

```
function sum(int[] items) -> (int r)
// Every item in items is greater or equal to zero
requires all { i in 0..|items| | items[i] >= 0 }
// Return must be greater or equal to zero
ensures r >= 0:
    //
    int r = 0
    int i = 0
    while i < |items| where r >= 0 && i >= 0:
        r = r + items[i]
        i = i + 1
    return r
```

Please cite this article in press as: D.J. Pearce, L. Groves, Designing a verifying compiler: Lessons learned from developing Whiley, Sci. Comput. Program. (2015), http://dx.doi.org/10.1016/j.scico.2015.09.006

Here, a bounded quantifier is used to enforce that `sum()` accepts an array of natural numbers. A key constraint is that summing an array 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 (more on this later).

2.6. Example 6 — flow typing & unions

An unusual feature of Whiley is the use of a *flow typing system* (see e.g., [35,36,28,37]) coupled with *union types* (see e.g., [38,39]). To illustrate, we consider null values. These have been a significant source of error in languages like Java and C#. The issue is that, in such languages, one can treat *nullable* references as though they are *non-null* references [40] (Hoare calls this his billion dollar mistake [41]). Although many approaches have been proposed (e.g., [42–48]), Whiley's type system provides an elegant solution:

```
// Return index of first occurrence of c in str, or null if none
function indexOf(string str, char c) -> int|null:
    ...
```

Here, `indexOf()` returns the first index of a character in the string, or **`null`** if there is none. The type **`int|null`** is a union type, meaning it is either an **`int`** *or* **`null`**. The system ensures the type **`int|null`** cannot be treated as an **`int`**. The following snippet illustrates how one must first check such a type *is* an **`int`** using the **`is`** operator (similar to `instanceof` in Java):

```
...
int|null idx = indexOf(...)
if idx is int:
    ...             // idx has type int
else:
    ...             // idx has type null
```

Here, we first ensure `idx` is an **`int`** before using it in the true branch. Furthermore, Whiley's flow type system automatically retypes `idx` on the true (resp. false) branch to have type **`int`** (resp. **`null`**). The use of union types here to manage **`null`** values is closely related to the use of option types in languages like Haskell, Scala and, more recently, Java 8. The essential difference being that union types provide a general mechanism for type composition which, for example, are non-disjoint.

2.7. Example 7 — effective unions

To further illustrate union types in Whiley, we consider unions of the same kind (e.g., a union of record types, or a union of array types). These expose commonality and we refer to them as *effective unions* (e.g., an effective record type). In the case of a union of records, fields common to all records are exposed:

```
// A circle has a position and a radius
type Circle is { int x, int y, int radius }

// A rectangle has a position and dimension
type Rectangle is { int x, int y, int width, int height }

// A shape is either a circle or a rectangle
type Shape is Circle | Rectangle
```

A `Shape` is either a `Rectangle` or a `Circle` (which are both record types). Any variable of type `Shape` exposes fields `x` and `y` *because these are common to all cases*. Finally, it's interesting to note that the notion of an effective record type is similar, in some ways, to that of the *common initial sequence* found in C [49].

2.8. Example 8 — recursive structures

Whiley provides recursive types which are similar to the algebraic data types found in functional languages (e.g., Haskell, ML, etc.). However, recursive types in Whiley build on unions for greater flexibility (i.e., as they are non-disjoint and structural). The following illustrates:

Please cite this article in press as: D.J. Pearce, L. Groves, Designing a verifying compiler: Lessons learned from developing Whiley, Sci. Comput. Program. (2015), http://dx.doi.org/10.1016/j.scico.2015.09.006

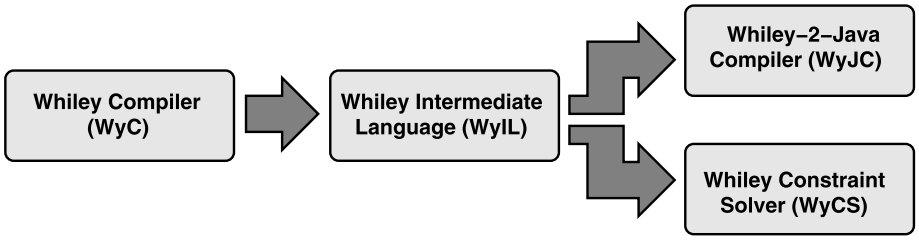


Fig. 1. Illustrating the compilation and verification pipeline.

```
public type Tree is null | Node

public type Node is { int data, Tree lhs, Tree rhs } where
    // Data in left node (if applicable) must be below this node
    (lhs is Node ==> lhs.data < data) &&
    // Data in right node (if applicable) must be above this node
    (rhs is Node ==> rhs.data > data)
```

This defines something approximating the notion of a binary search tree. As recursive types in Whiley always describe tree-like structures it is, for example, impossible for an instance of `Tree` to contain a cycle. The above also illustrates how flow typing gives an elegant solution for managing the type invariants. Specifically, on the right-hand side of the implication, `lhs` (resp. `rhs`) is automatically retyped to `Node`. Finally, the invariant given above permits, for example, `data < lhs.rhs.data` for a given tree node and, thus, is not sufficient to properly characterise binary search trees. This can be done, however, using an auxiliary predicate which returns the set of all data elements in a given tree.

To illustrate the flexibility of recursive types in Whiley compared with algebraic data types, we can define the notion of a "three tree" as follows:

```
type ThreeTree is { int data, Node lhs, Node rhs } where
    // Data in left (resp. right) node must be below (resp. above) this node
    lhs.data < data && rhs.data > data
```

Here, a `ThreeTree` is a tree which contains at least three nodes. Since types are structural in Whiley and unions are non-disjoint, it immediately follows that `ThreeTree` is a subtype of `Node` and, hence, `Tree`. Since an explicit subtyping declaration is not required for this (i.e., `extends` or `implements` in Java), it also follows that the definition of `ThreeTree` can be added subsequently by a different programmer in another file or library (though this does require the original definitions are made visible outside the enclosing file via the **`public`** modifier).

3. Compiler overview

The Whiley verifying compiler provides a full-stack verification and compilation system, including an implementation of an automated theorem prover. In this section, we give a brief technical overview of the verifying compiler (further details can be found elsewhere [25]).

3.1. Architecture

Fig. 1 provides an overview of the flow of information within the compiler. Here, `whiley` source files are converted into (binary) `wyil` files; in turn, these are converted into binary `class` files (for execution on the JVM) and `wyal` source files (for verification). The main modules are:

- **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 JVM bytecode or Microsoft CIL) along with an accompanying binary file format.
- **Whiley-2-Java Compiler (WyJC).** A back-end which converts `wyil` files into JVM `class` files.

Please cite this article in press as: D.J. Pearce, L. Groves, Designing a verifying compiler: Lessons learned from developing Whiley, Sci. Comput. Program. (2015), http://dx.doi.org/10.1016/j.scico.2015.09.006

- **Whiley Constraint Solver (WyCS).** An automated theorem prover responsible for converting `wyil` files into `wyal` files and verifying they are correct. The latter contains verification conditions written in a variant of first-order logic called the *Whiley Assertion Language (WyAL)*.

From Fig. 1, we see that the intermediate language forms the core of the compiler. This follows the approach of similar tools. For example, the authors of the widely acclaimed Spec# tool claimed that introducing an intermediate language (i.e., Boogie) was "*the best and most far-reaching single design decision we made in implementing the Spec# verifier*" [10]. Similarly, the ESC/Java tool uses an intermediate language based on Dikstra's language of guarded commands [7]. Both of these are intermediate languages specifically focused on verification, rather than compilation as well. In contrast, the Whiley Intermediate Language (WyIL) is a bytecode language designed specifically to support both tasks (more information on this is available elsewhere [50]). For example, control-flow is semi-structured to allow source-level loop statements and their invariants to be encoded uniformly using a specialised `loop` bytecode, whilst **`break`** and **`continue`** statements are handled with unconditional branching.

The verifying compiler is implemented in a modular fashion with distinct file formats for input to each module. This 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, for example, replace the WyCS theorem prover with another (e.g., Z3 [51] or Simplify [52]) by writing a wrapper which converts files in the `wyal` format into the appropriate input language of the external tool (see Section 8 for more on this).

Finally, a prototype C backend has been developed which allows Whiley programs to be translated into C and then compiled into native machine code. Furthermore, other backends for the compiler are planned for the future (e.g., LLVM, JavaScript, etc.).

3.2. Intermediate language

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

```
function abs(int x) -> (int r)
ensures r >= 0:
    //
    if x < 0:
        x = -x
    return x
```

```
function abs(int) -> int:
ensures:
  const %1 = 0
  ifge %0, %1 goto label0
  fail
.label0
  return
body:
  const %1 = 0
  ifge %0, %1 goto label1
  neg %0 = %0
.label1
  return %0
```

Here, two bytecode blocks have been generated from the Whiley source with the first representing the **`ensures`** clause, and the latter representing the function's body. Indeed, the Whiley compiler translates all preconditions, postconditions, loop invariants and type invariants into bytecode and this is what the verification condition generator operates on. A bytecode block (such as `body` above) has an unlimited register set available to it, and these are prefixed with `%` above (e.g., `%1`). As for JVM bytecode, the set of registers used in any given block is statically known and, furthermore, registers hold parameter values on entry (i.e., `%0` holds parameter `x` on entry). In the above example, the `const` bytecode loads an integer constant into register `%1`. The `neg` bytecode negates its operand and assigns to a given register. Finally, the `return` bytecode returns its operand.

3.3. Assertion language

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

Please cite this article in press as: D.J. Pearce, L. Groves, Designing a verifying compiler: Lessons learned from developing Whiley, Sci. Comput. Program. (2015), http://dx.doi.org/10.1016/j.scico.2015.09.006

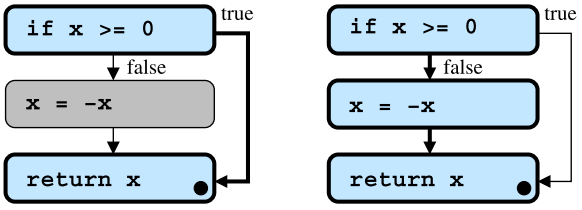
```
assert:
  forall(int x0):
    if:
      x0 >= 0
    then:
      x0 >= 0
```

```
assert:
  forall(int x0, int x1):
    if:
      x0 < 0
      x1 == -x0
    then:
      x1 >= 0
```

Here, the left assertion corresponds to the execution path through the false branch of the **if** statement in the original Whiley function; likewise, the right assertion corresponds to the path through the true branch. Note, indentation syntax is used to aid readability here, and conditions on adjacent lines with the same indentation are conjoined.

3.4. Verification condition generation

The verification condition generator traverses the control-flow graph of the function in a path-sensitive fashion. When a branch is encountered, the generator forks and proceeds independently down each path. The following illustrates:



Here, the generator explores both paths through the function and, for each, generates separate verification conditions at the **return** statement to enforce the post-condition (as indicated by the marks). In doing this, the generator accumulates knowledge about the current path taken through conditional branches and the effect statements have on program variables. Thus, reaching the **return** statement through the true branch of the conditional yields the state $x_0 \geq 0 \wedge r = x_0$ (ignore the subscripts for now, and recall r is the declared return value). In contrast, traversing the false branch yields $x_0 < 0 \wedge x_1 = -x_0 \wedge r == x_1$.

Subscripts are used in intermediate generator states to denote the current assignment of variables (which is similar, in some ways, to static single assignment form [53]). In any given state, the greatest subscript for a given variable denotes the "current" value, whilst lower subscripts denote "earlier" values. For example, x_1 represents the value of variable x after the assignment, whilst x_0 its value before that point. Observe that we cannot discard earlier values since they may contain important and relevant information (such as on the false branch above).

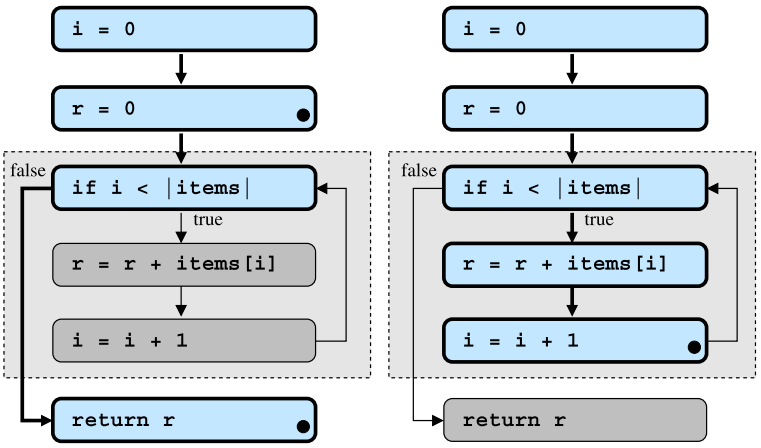
3.5. Loops

The treatment of loops warrants special attention. A critical challenge is the separation of information from before the loop from that within and after the loop. Consider the following example:

```
function sum(int[] items) -> (int r)
requires all { i in 0..|items| | items[i] >= 0 }
ensures r >= 0:
  //
  int i = 0
  int r = 0
  while i < |items| where i >= 0 && r >= 0:
      r = r + items[i]
      i = i + 1
  return r
```

In this case, the generator traverses two distinct paths. The first passes through the **while** statement without entering the body; the second enters the body but does not continue afterwards. The following illustrates, where the loop is highlighted with dashed boxes:

Please cite this article in press as: D.J. Pearce, L. Groves, Designing a verifying compiler: Lessons learned from developing Whiley, Sci. Comput. Program. (2015), http://dx.doi.org/10.1016/j.scico.2015.09.006



The generator produces three verification conditions to be discharged: the first establishes the loop invariant on entry (above left); the second establishes the post-condition assuming the loop invariant after the loop (above left); the third establishes the loop invariant at the end of the loop, assuming it held at the beginning (above right).

Following a strict interpretation of Hoare logic, information about program variables from before the loop cannot be carried forward into or after the loop. This is achieved in the generator simply by incrementing the suffix of all affected variables (which is commonly referred to as "havocing" variables). For example, consider the path through the above control-flow graph shown on the left. Upon reaching the return statement, the generator has the following state:

$$\underbrace{\forall i \in items_0.(i \geq 0)}_{\text{Precondition}} \wedge \underbrace{i_0 == 0 \wedge r_0 == 0}_{\text{From before loop}} \wedge \underbrace{i_1 >= |items_0|}_{\text{Inverted Condition}} \wedge \underbrace{i_1 >= 0 \wedge r_1 >= 0}_{\text{Loop Invariant}}$$

Here, the disconnection between information known before the loop from that known afterwards is evident by the use of disjoint variables (i.e. i_0 represents variable `i` before the loop, whilst i_1 represents it after the loop, etc.). In fact, this strategy corresponds to Floyd's rule for assignment where we have immediately skolemised the generated existentials [54].

In practice, preventing information about all program variables from being carried forward into and after the loop is unnecessarily restrictive. For example, reestablishing the above loop invariant depends on the information which held beforehand about the `items` variable. Since this variable is not modified by the loop, our generator will not send it to havoc on entry to the loop. We return to discuss this further in Section 5.2.

4. Reflections on language design decisions

In designing the Whiley language, we have attempted to make choices which simplify the verification of its programs. We now examine and reflect upon these decisions.

4.1. Paradigm

> **D1.** *Functions are pure, whilst methods maybe impure.*

Justification Our starting point in designing Whiley was to focus on an imperative-style language as this exposes the full range of verification problems (e.g., such as specifying loop invariants, handling aliasing, etc.). However, existing imperative languages (e.g., Java, C#, C, etc.) 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) [55–58]. To address this, Whiley explicitly distinguishes between pure functions and impure methods. The following illustrates:

```
function f(int x) -> int: // pure
   return x

method m(&int pX):        // impure
   *pX = 1
```

Here, `f()` is a pure function whilst `m()` is an impure method with side-effects. In particular, the type `&int` represents a *reference* to an `int`. Functions cannot accept references and/or perform I/O, and must return a value of some description (otherwise they have no effect). In contrast, methods may mutate external state through references and/or perform I/O.

Please cite this article in press as: D.J. Pearce, L. Groves, Designing a verifying compiler: Lessons learned from developing Whiley, Sci. Comput. Program. (2015), http://dx.doi.org/10.1016/j.scico.2015.09.006

Reflections This choice enables a clean distinction between what is permitted in a specification and what is not. Retaining the notion of an *impure* method also simplifies integration with existing (e.g., embedded) systems (see Section 8 for more). Similar approaches appear in most existing systems. For example, both JML and Spec# support a `pure` modifier for this purpose which provides a notion of *weak purity* [59,10]. In contrast, Dafny supports true mathematical functions like Whiley [12]. Unlike Whiley, however, Dafny also supports *framing* which allows mathematical functions to read heap locations [60]. A frame refers to the state upon which a function operates, including that reachable through parameters. In this way, for example, mathematical functions can traverse mutable object structures. As such, the treatment of references in Whiley is more limited and would benefit from such mechanisms for reasoning about mutable object structures.

4.2. Value semantics

D2. *Compound data types have value semantics.*

Given the restriction on the use of references within functions and specifications, the need to access compound data within them remains. To address this, all compound structures in Whiley (e.g., arrays 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. This latter point serves a critical purpose: to give Whiley the appearance of an *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 (recall Section 1). For example:

```
function f(int[] xs) -> int:
    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, meaning that `xs` is updated inside `f()` and this does not affect `f`'s caller. That is, `xs` is not a *reference* to an array of **`int`**; rather, it *is* an array of **`int`**s and assignments to it do not affect state visible outside of `f()`.

Reflections The use of value semantics is critical to verification in Whiley as it allows collections to be used in specifications and functions. This differs from most existing systems, such as JML and Spec#, where collection types are not first-class and, hence, are typically impure. Indeed, without a fundamental immutable collection type it is very difficult to specify such collections themselves [8]. As such, JML supports mathematical collection types as models which can only be used within specifications. In contrast, Dafny also provides first-class support for immutable sets, lists and recursive datatypes and these can be used in specifications and mathematical functions.

Finally, we note that the choice to support value semantics in Whiley impacts upon the language's potentially efficiency, particularly within the context of existing (e.g., embedded) systems. However, there are techniques which can be applied here, such as the use of reference counting to minimise unnecessary cloning of compound structures [61–63].

4.3. Arithmetic

D3. *Arithmetic is unbounded.*

Justification Modern languages typically provide fixed-width numeric types, such as 32-bit twos-complement integers, or 64-bit IEEE 754 floating point numbers. Such data types are notoriously difficult for an automated theorem prover to reason about [64]. 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.

Reflections Specifying programs in the presence of arithmetic which may overflow is surprisingly difficult. Indeed, when using VeriFast it is repeatedly recommended to disable overflow checking [65], whilst systems like JML and Spec# assume (unsoundly) that numeric types do not overflow or suffer rounding. Again, Dafny makes the same decision as Whiley to support unbounded arithmetic (although efficient implementation does not appear to be a concern). In contrast, efficient implementation is a concern for Whiley. For example, compiling Whiley programs for embedded systems would be infeasible without the ability to generate fixed-width integer types. In principle, this can be achieved through the use of specifications. That is, by specifying finite bounds on data types which the compiler can exploit. At this stage, however, it remains to implement an integer range analysis for this purpose (see e.g., [66]). Furthermore, it is unclear how to constrain Whiley's **`real`** type (which represents arbitrary rationals) to enable the use of fixed-width representations and further work is needed to explore existing techniques [67].

Please cite this article in press as: D.J. Pearce, L. Groves, Designing a verifying compiler: Lessons learned from developing Whiley, Sci. Comput. Program. (2015), http://dx.doi.org/10.1016/j.scico.2015.09.006

4.4. Runtime verification

D4. *All specifications are checkable at runtime.*

Justification Undecidability and other practical constraints (e.g., timeouts) present a challenge for any verification system, and we must anticipate that the theorem prover may not always reach a conclusion. To deal with this, systems like JML and Spec# permit the use of *runtime checking* as an alternative mechanism [8,68]. This is a useful fall-back when the theorem prover cannot make strong conclusions about the validity of a statement.

To use runtime checking in this manner, one must ensure that all specifications are *executable* (i.e., can be turned into runtime checks). Whiley provides a strong guarantee every specification can be turned into a runtime check. In particular, all quantification must be *bounded* (i.e., over a set which is finite at runtime). This is because bounded quantification can be easily translated into a **`for`**-loop over its elements.

Reflections The choice to restrict Whiley to executable specifications means the language is strictly less expressive than, for example, JML or Dafny [68]. This is because the use of unbound quantification (i.e., quantification over an infinite set) must be prohibited [69,70]. The following illustrates (using Whiley-like syntax):

```
function f(int x, int y) -> (int r):
    return x + y

assert all { x in int, y in int | f(x,y) == f(y,x) }
```

The **`assert`** statement requires that the function `f(int,int)` is *commutative*. This example cannot be expressed in Whiley because it is quantifying over the infinite set of all integers. As such, the assertion *cannot be converted into a runtime check* because it would require an infinite amount of time to check.[1] Finally, we note that at this stage it remains unclear to us how much of an issue this would be in practice.

4.5. Verification vs typing

D5. *Type checking is separate from verification.*

Justification Type checking is typically implemented as an efficient and decidable process within a compiler. In contrast, verification via an automated theorem prover is typically undecidable. To that end, the process of type checking is strictly separate from that of verification in Whiley. This means, for example, verification can be turned off whilst retaining type safety. It also means that the verifier can assume programs are well-typed during verification.

Reflection From the user's perspective, the distinction between typing and verification in Whiley is blurred and this exposes some difficult edge cases. To understand this, consider the following:

```
function f(int|null x) -> bool|null:
  //
  if x is int && x >= 0:
     return true
  else if x is int && x < 0:
     return false
  else:
     return x
```

This function appears on the surface to be a valid Whiley program. Unfortunately, it is not and the Whiley compiler correctly rejects this program, reporting a message of the form "`expected type null|bool, found int|null`". This is potentially confusing from a usability perspective! To understand why this program is not valid, one must properly understand the distinction between typing and verification. Specifically, that typing is separate from and occurs before verification. Thus, the type checker is unable to reason that `x` cannot hold an **`int`** in the **`else`** block.

[1] Whilst it is true that integers in most languages have fixed-width representations, this means only that quantification over integers is implicitly bounded in such languages. However, for other data types (e.g., collections), this is not the case.

Please cite this article in press as: D.J. Pearce, L. Groves, Designing a verifying compiler: Lessons learned from developing Whiley, Sci. Comput. Program. (2015), http://dx.doi.org/10.1016/j.scico.2015.09.006

The above problem motivates the need for a mechanism to enable some interaction between typing and verification. Consider trying to resolve the above error as follows:

```
...
else if x is int && x < 0:
   return false
else if x is null:
   return x
else:
   return ???
```

Here, the user is attempting to work around the type checker by splitting the final case in two. This is sensible, but exposes the need to return a dummy value in the last case (i.e., because it is actually dead code). This is somewhat frustrating, and has motivated the subsequent inclusion of a special **`fail`** statement into the language:

```
...
else if x is null:
   return x
else:
   fail
```

The **`fail`** statement is recognised by the type checker as indicating unreachable code and, hence, that it need not continue down this branch. Conversely, the **`fail`** statement is recognised by the verifier as a fact which must be established (i.e., that this branch is indeed unreachable). Note that substituting "**`assert`** `false`" does not work here because the type checker must assume control may continue after this statement.

Finally, we note that Spec# is the only other system we are aware of supporting something similar to the **`fail`** statement. Specifically, the statement "**`assert`** `false;`" is given special treatment by the type checker so that the following compiles:

```
bool f(int y)
ensures result ==> y > 0;
ensures !result ==> y <= 0;
{
   if(y > 0) {
     return true;
   } else if(y <= 0) {
     return false;
   } else {
     assert false;
   }
}
```

However, perhaps unexpectedly, this no longer compiles if one removes the **`else`** branch altogether, or replaces "**`assert`** `false;`" with "**`assert`** `!true;`". Thus, "**`assert`** `false`" is effectively a special statement equivalent to our **`fail`** statement. Finally, we note that Dafny does not ensure execution paths are terminated with return statements for methods with declared return types (hence, in our example above, one can simply omit the else branch).

5. Reflections on loop invariants

Hoare logic was the starting point for developing our verifying compiler, and is the general approach taken by most comparable tools [71]. However, whilst Hoare logic provides an excellent foundation for reasoning about programs, there remain a number of hurdles to overcome in developing a practical tool. In this section, we examine issues relating to loop invariants using examples based on those we encountered in practice.

The issue of specifying loop invariants is widely examined in the literature (see e.g., [7,72–75]) and was well known to us. Unfortunately, we were still surprised by how much of a problem this is in practice. Indeed, Flanagan and Qadeer comment that *"Our experience with ESC/Java indicates that the burden of specifying loop invariants is substantial"* [76]. Our original plan was to avoid automatically inferring loop invariants as, in general, this is a hard algorithmic challenge (see e.g., [77, 72,78,74,79]). However, we now believe that synthesising invariants (where possible) is crucial for achieving a tool which is accessible to the non-specialist.

5.1. Simple synthesis

As mentioned above, generating loop invariants in the general case is hard. However, there are situations where loop invariants can easily be determined. The following illustrates an interesting example:

Please cite this article in press as: D.J. Pearce, L. Groves, Designing a verifying compiler: Lessons learned from developing Whiley, Sci. Comput. Program. (2015), http://dx.doi.org/10.1016/j.scico.2015.09.006

```
function sum(int[] xs) -> (int r)
// All elements of parameter xs are greater-or-equal to zero
requires all { i in 0..|xs| | xs[i] >= 0 }
// Return value must be greater-or-equal to zero
ensures r >= 0:
    //
    int i = 0
    int r = 0
    //
    while i < |xs| where r >= 0:
        r = r + xs[i]
        i = i + 1
    //
    return r
```

This function computes the sum of an array of natural numbers, and returns a natural number. We can see that an explicit loop invariant has been given through a `where` clause. The question to consider is: *did the programmer specify the loop invariant properly?* Unfortunately, the answer is: *no*. In fact, the loop invariant needs to be extended as follows:

```
    ...
    while i < |xs| where i >= 0 && r >= 0:
        r = r + xs[i]
        i = i + 1
    //
    return r
```

The need to include `i >= 0` in the loop invariant is frustrating as, intuitively, it is trivial to see that it holds. In the future, we plan to automatically synthesise simple loop invariants such as this using static analysis, or similar (see e.g., [78,80]). Finally, we note Dafny has limited support for inferring invariants (for example, it infers `i >= 0` above but not `r >= 0`).

Constrained types As discussed in Section 2, Whiley also supports the notion of constrained types. In fact, by exploiting constrained types with variable declarations we can provide an alternative solution to the above issue with `sum()`:

```
function sum(int[] xs) -> (int r)
// All elements of parameter xs are greater-or-equal to zero
requires all { i in 0..|xs| | xs[i] >= 0 }
// Return value must be greater-or-equal to zero
ensures r >= 0:
    //
    nat i = 0
    nat r = 0
    //
    while i < |xs|:
        r = r + xs[i]
        i = i + 1
    //
    return r
```

Here, variable declarations are used to restrict the permitted values of variables `i` and `r` throughout the function. In order to give Whiley the look-and-feel of a dynamically typed language, it originally did not support variable declarations. However, these have now been added specifically to ease the burden of writing loop invariants. Perhaps surprisingly, Why3 is the only comparable tool we are aware of which supports constrained types in this fashion (though class invariants in some languages are similar, but cannot be used on primitive types in this way) [81].

5.2. Loop invariant variables

As discussed in Section 3.5, a strict interpretation of Hoare logic requires that all information about variables in a loop be given through the loop invariant. From a usability perspective this is undesirable as it requires the programmer to completely specify the loop invariant even in cases where this should be unnecessary. For example, consider the following Whiley program:

Please cite this article in press as: D.J. Pearce, L. Groves, Designing a verifying compiler: Lessons learned from developing Whiley, Sci. Comput. Program. (2015), http://dx.doi.org/10.1016/j.scico.2015.09.006

```
function f(int x) -> (int r)
// Parameter x must be greater than zero
requires x > 0
// Return value must be greater-or-equal to ten
ensures r >= 10:
    //
    int i = 0
    //
    while i < 10 where i >= 0:
        i = i + x
    //
    return i
```

Intuitively, we can see this program satisfies its specification. Unfortunately, under a strict interpretation of Hoare logic, it cannot be shown as correct because the loop invariant is insufficient. To understand this, consider Hoare's classical rule for loops:

$$\frac{\left\{Q \wedge B\right\}\ \mathtt{S}\ \left\{Q\right\}}{\left\{Q\right\}\ \mathtt{while(B)}\ \mathtt{S}\ \left\{\neg B \wedge Q\right\}} \quad \text{(H-WHILE)}$$

Here, Q corresponds to the loop invariant and this, along with the loop condition, is the only information carried through the loop body. A strict interpretation of this rule would suggest that Q corresponds to the explicit loop invariant given in the Whiley program. However, this would mean that necessary properties of all variables must be explicitly stated in the loop invariant and, in practice, we want to relax this where possible to reduce programmer burden.

In the above Whiley program, we refer to variable `x` as being *loop invariant*. That is, the variable is not assigned within the body of the loop and, hence, its value is an invariant of the loop. The Whiley verifier allows information about such variables to be carried forward into and after the loop, thereby allowing the above program to verify without further changes. This is done easily by not havocing such variables on entry to the loop (recall Section 3.5). Finally, the work of Beckert et al. provides some theoretical foundations which underpin this [82]. Similarly Barnett and Leino refer to variables updated in the loop body as *loop targets* and, again, only havoc these variables [83]. In fact, almost all comparable tools support loop invariant variables in this way (e.g. Dafny, Spec#, ESC/Java, etc.). Indeed, without explicit support for loop invariant variables, *there would be no way in such tools to specify that a variable is not modified by a loop* and many programs would simply be unverifiable (more on this later).

5.3. Loop invariant properties

Whilst our verifying compiler easily handles loop invariant variables, there remain situations when invariants need to be needlessly respecified. Consider the following:

```
function add(int[] v1, int[] v2) -> (int[] v3)
// Input lists must have same length
requires |v1| == |v2|
// Return must have same length as parameters
ensures |v3| == |v1|:
    //
    int i = 0
    while i < |v1| where i >= 0:
        v1[i] = v1[i] + v2[i]
        i = i + 1
    return v1
```

This example adds two vectors of equal size. Unfortunately, this again does not verify under the rule H-WHILE because the loop invariant is too weak. The key problem is that `v1` is modified in the loop and, hence, our above solution for loop invariant variables does not apply. Following rule H-WHILE, the verifying compiler can only reason about what is specified in the loop condition and invariant. Hence, it knows nothing about the size of `v1` after the loop and, hence, cannot establish that `|v1| == |v2|` holds after the loop. Likewise (and more importantly in this case), it cannot establish that the size of `v1` is unchanged by the loop (which we refer to as a *loop-invariant property*). Thus, it cannot establish that the size of the returned vector equals that held in `v1` on entry, and reports the function does not meet its postcondition. Adding explicit support for loop-invariant properties of variables (such as the size of an array) would seem to be helpful. Perhaps surprisingly, we are not aware of any comparable tools which do so and, in the future, we aim to extend Whiley with support for this.

Please cite this article in press as: D.J. Pearce, L. Groves, Designing a verifying compiler: Lessons learned from developing Whiley, Sci. Comput. Program. (2015), http://dx.doi.org/10.1016/j.scico.2015.09.006

An interesting question here is how one should verify the above program. Unfortunately, in the Whiley language developed thus far (and most comparable tools) it is *impossible to directly specify that the size of `v1` is unchanged by the loop*. This is because there is no direct syntax for expressing the state of a variable at the beginning of a loop iteration, versus at the end. For example, if `v1'` denoted the value of `v1` at the end of a loop iteration, then `|v1| == |v1'|` would express the required property. Note, it is unclear what such an invariant would mean on entry to the loop and we would probably need additional syntax to distinguish the normal invariant (e.g., which must hold on entry) from that which establishes relationships between values in successive iterations.[2]

Returning to our example we can, in fact, exploit a simple trick which allows the given function to verify. Since `v2` is not modified by the loop, and `|v1| == |v2|` held on entry, we can use `i >= 0 && |v1| == |v2|` as the loop invariant.

Ghost variables Our solution to the problem of verifying `add()` suggests a general pattern for handling loop-invariant properties. That is, in situations where no variable with the desired property already exists (like `v2` above), we can *introduce* one (commonly called a *ghost variable* [7,13,84]). The following illustrates:

```
function add(int[] v1, int x) -> (int[] v2)
// Return must have same length as parameter
ensures |v1| == |v2|:
    //
    int tmp = |v1| // ghost
    int i = 0
    //
    while i < |v1| where i >= 0 && |v1| == tmp:
        v1[i] = v1[i] + x
        i = i + 1
    //
    return v1
```

Here a temporary variable, `tmp`, has been introduced specifically to ensure the above function can be verified. It is a loop invariant variable and, hence, whatever was known before the loop is known after. Specifically, in this case, that it matches the length of `v1` on entry to the function. Thus, we can exploit this temporary variable to specify an appropriate loop invariant.

5.4. Break/continue

Another interesting question faced when extending Hoare logic to a practical system, such as Whiley, is the handling of **`break`** and **`continue`** statements. These are an important concern because of their prevalence in practice. For example, Chalin examined the verification of real-world loops with unstructured control-flow (e.g., break/continue statements) and found that "*40% of the 1500 loops in the Eclipse JDT (an industrial grade open source Java compiler) have conditions with side-effects, and/or bodies containing breaks, continue or return statements*" [85].

Whilst **`continue`** statements are relatively straightforward, there are two differing approaches to handling **`break`** statements. The first requires the loop invariant be restored at the point of the break, whilst the alternative approach does not. Instead, the latter departs from rule H-WHILE above in allowing the loop invariant and the loop's postcondition to differ more significantly. Specifically, the loop's postcondition is the disjunction of the loop invariant and those properties known at any **`break`** statements within the loop. This is the approach taken in Whiley, and the following illustrates:

```
function indexOf(int[] items, int item) -> (int r)
ensures r == |items| || items[r] == item:
    //
    int i = 0
    //
    while i < |items| where i >= 0 && i <= |items|:
        if items[i] == item:
            break
        i = i + 1
    return i
```

Verifying this program requires that the verifier allows for greater differences between the loop invariant and loop postcondition. Recall from Section 3.5 that our verification condition generator explores the simple (i.e., acyclic) paths of a

[2] In fact, JML supports the use of `\old(i)` within a loop invariant to refer to the value of `i` which held on entry to the loop (rather than, for example, the value which held on the previous iteration).

Please cite this article in press as: D.J. Pearce, L. Groves, Designing a verifying compiler: Lessons learned from developing Whiley, Sci. Comput. Program. (2015), http://dx.doi.org/10.1016/j.scico.2015.09.006

function. Thus, handling **break** statements is relatively straightforward within the Whiley compiler as they simply generate alternative paths out of the loop.

5.5. Do-while loops

Another practical concern is the handling of do-while loops. Again, there have been some differences of opinion in how to do this. Essentially, the question is whether or not the loop invariant needs to hold on entry to the loop. Consider the following:

```
function lastIndexOf(int[] xs, int x) -> (int r)
requires |xs| > 0
ensures r == -1 || xs[r] == x:
  //
  int i = |xs|
  do:
    i = i - 1
  while i >= 0 && xs[i] != x where i >= -1 && i < |xs|
  //
  return i
```

The Whiley compiler verifies this function meets its specification. To do so requires treating the first iteration of the loop separately from the remainder. Specifically, the loop invariant does not need to hold on entry to the loop. Instead, it must hold at the end of the loop before the condition. This can be understood by unrolling the loop as follows:

```
function lastIndexOf(int[] xs, int x) -> (int r)
requires |xs| > 0
ensures r == -1 || xs[r] == x:
  //
  int i = |xs|
  i = i - 1
  while i >= 0 && xs[i] != x where i < |xs| && i >= -1:
    //
    i = i - 1
  //
  return i
```

Hoare and Wirth provided a rule in the style of Hoare logic for handling do-while loops which operates in essentially this manner [86], whilst Alagic and Arbib proved its equivalence to the while loop unrolling [87].

From a usability perspective, our treatment of do-while loops is perhaps a little concerning. In particular, the treatment of while versus do-while loops is inconsistent (i.e., the loop invariant is/is not required to hold on entry). Furthermore, understanding the rule for do-while loops is conceptually more challenging because it requires reasoning about the first iteration distinctly from the remainder. Not all tools have adopted this approach to handling do-while loops. For example, the authors of VCC [84] made an explicit decision that the invariant must hold on entry [88], because otherwise:

> "*It introduces a disjunction at the top of the loop: you know that either this is the first time through the loop or the invariant holds. Now, when some assertion fails in the middle of the loop body, there is confusion about whether it was on the first iteration*"

Similarly, Spec# [89], ESC/Java [76] and GPUVerify [90] all require the loop invariant holds on entry, whilst Dafny does not currently support do-while loops. Frama-C is perhaps the exception amongst comparable tools in allowing flexibility here [91]. The syntax used in, for example, Spec# is illustrative. Consider the following:

```
int f(int x) {
  //
  int i = 0;
  do invariant i > 0; {
    i = i + 1;
  } while(i < x);
  //
  return i;
}
```

This Spec# program does not verify because the loop invariant does not hold on entry. The positioning of the `invariant` expression is interesting as this clearly indicates the invariant must hold on entry. Thus, the above can be easily corrected

Please cite this article in press as: D.J. Pearce, L. Groves, Designing a verifying compiler: Lessons learned from developing Whiley, Sci. Comput. Program. (2015), http://dx.doi.org/10.1016/j.scico.2015.09.006

by updating the invariant to `i >= 0`. Despite this, it remains unclear to us which approach to do-while loops should be favoured. In particular, we note that requiring the loop invariant to hold on entry does prohibit some correct loops from being verified as is (i.e., without being first transformed into while loops). For example, do-while loops which initialise variables on the first iteration are problematic with this approach, as the loop invariant cannot use such variables.

6. Further reflections

In the previous section we reflected on the design of the Whiley language and its support for loop invariants. In this section, we continue our reflections by examining some further problematic issues.

6.1. Predicates as uninterpreted functions

Whiley permits the use of (pure) functions within specifications to increase their expressiveness. This can be useful as a mechanism for improving code reuse, or as an aid to readability. Unfortunately, this exposes subtle pitfalls for the inexperienced user. To understand this, consider the following (albeit contrived) example:

```
function f(int x, int y) -> int
requires x > y:
   //…
```

Let us imagine the user wishes to factor out the invariant between `x` and `y` for reuse in other places. There are two approaches here. Firstly, one can define a new record type with two fields and an appropriate invariant:

```
type greaterThan is { int x, int y } where x > y
```

This type can now be reused across multiple methods/functions and when defining other data types. This approach is sensible and it works. However, an alternative approach might be to create a *predicate function* as follows:

```
function greaterThan(int x, int y) -> bool:
    return x > y

function f(int x, int y) -> int
requires greaterThan(x,y):
   //…
```

Unfortunately, this approach does not work as expected and can cause considerable confusion. Like most tools of this nature, Whiley treats functions in specifications as *uninterpreted functions* [92]. This means that only the specifications of such functions are considered during verification (rather than their implementations as well). As such, the following program will fail to verify:

```
function g(int x, int y) -> int|null:
   if x > y:
      return f(x,y)
   else:
      return null
```

Whiley's verifying compiler complains that the precondition is not met for the call `f(x,y)`. This is somewhat counter-intuitive, since the conditional appears to be doing exactly this. In essence, the problem is that the body of `greaterThan()` is not considered by the verifier. One way to modify the above function so that it compiles is as follows:

```
function g(int x, int y) -> int|null:
   if greaterThan(x,y):
      return f(x,y)
   else:
      return null
```

Adopting this approach can require whole-scale changes to a code base as, to call `f(x,y)`, we now require `greaterThan(x,y)` rather than just `x > y`. An alternative is to encode the predicate's implementation within its specification as follows:

Please cite this article in press as: D.J. Pearce, L. Groves, Designing a verifying compiler: Lessons learned from developing Whiley, Sci. Comput. Program. (2015), http://dx.doi.org/10.1016/j.scico.2015.09.006

```
function greaterThan(int x, int y) -> (bool r)
ensures r <==> x > y:
    return x > y
```

This provides a general pattern for extracting conditions as predicates. However, it also seems somewhat wasteful due to the necessary redundancy between the implementation and specification. Although not implemented in Whiley, one option is to support `predicate` declarations for exactly this purpose. An alternative would be to support macros as an alternative to uninterpreted functions.

6.2. Inductive reasoning

The inability of modern automated theorem provers (e.g., Simplify [52], Z3 [51], etc.) to reason inductively about functions in specifications is a well-known issue [93]. To understand this, consider specifying the `sum()` function as follows:

```
function sum(int[] xs, int i) -> (int r)
requires |xs| > 0 && i >= 0 && i <= |xs|
// Base case: empty subarray
ensures i == |xs| ==> r == 0
// General case: at least one element in subarray
ensures i < |xs| ==> r == xs[i] + sum(xs,i+1):
    // ...
```

Whilst this specification appears sensible, exploiting known mathematical properties in subsequent specifications is essentially impossible. For example, consider the following specification for reversing an array:

```
// Rotate head item to back of array
function reverse(int[] xs) -> (int[] ys)
// Permutation does not change sum
ensures |xs| > 0 ==> sum(xs,0) == sum(ys,0):
   // ...
```

Unfortunately, one cannot automatically verify the seemingly straightforward property that the sum is preserved by this function (i.e., regardless of its implementation). Providing some mechanism to interact with the automated theorem prover seems critical to dealing with situations such as this. One can do this using an **assume** statement in Whiley to override the verifier. Such a statement provides a way to instruct the verifier to blindly assume something holds. For this example, we have:

```
   assume |xs| > 0 ==> sum(xs,0) == sum(ys,0)
```

Placing this before the **return** statement allows the verifier to pass `reverse()`. Of course, the use of **assume** statements is potentially unsafe and relies on correct judgement. In the future, we intend to support interactive proofs for such cases.

Finally, we note that some systems provide other mechanisms for interacting with the verifier. For example, Dafny and VeriFast provide support *lemma methods*. These are ghost functions which exploit loops to establish inductive properties that can then be used elsewhere. Although Whiley does not directly support lemma methods, a similar effect can be achieved simply by writing additional functions to establish required properties. Another approach is to provide explicit `fold` and `unfold` primitives [94].

6.3. Triggers

The use of quantification is a known challenge for automated theorem provers [95–99]. Unfortunately, it is a trap for both experienced and inexperienced user alike. For example, consider the following (albeit contrived) function:

```
function to100() -> (int r)
ensures r == 100:
   //
   int r = 0
   while r < 100
       where 0 <= r && r <= 100
       where some { k in 0 .. 50 | 2 * k == r }:
       //
       r = r + 2
   return r
```

Please cite this article in press as: D.J. Pearce, L. Groves, Designing a verifying compiler: Lessons learned from developing Whiley, Sci. Comput. Program. (2015), http://dx.doi.org/10.1016/j.scico.2015.09.006

Table 1
Overview of benchmarks.

Benchmark	Description	Status	Notes
`01_sumax` (VSCOMP'10)	Compute the sum and maximum of an array of N natural numbers, and prove the postcondition $\texttt{sum} \leq N \cdot \texttt{max}$	10/11	The easiest benchmark to specify. However, our verifier's limited ability to reason about non-linear arithmetic prevents it from discharging one verification condition.
`02_invert` (VSCOMP'10)	Invert an injective array `A`, and prove the postcondition that output array `B` is injective and that `B[A[i]]==i` for all elements.	8/12	The most challenging benchmark for the verifier. It remains unclear whether the function is specified in a way which our verifier could accept. Leino commented previously on the need for a supporting Lemma for this benchmark and the lack of this in our version may be the issue [32].
`03_maxarray` (COST'11)	Find the maximum element of an array using Kaldewaij's "Search by Elimination" method [100], and prove the postcondition that the element returned is indeed maximum.	12/12	Verifier discharged all verification conditions without intervention.
`04_duplets` (COST'11)	Given an array of natural numbers which contains at least two identical elements, return any two such indices and prove the postcondition that their elements match.	13/13	Verifier discharged all verification conditions. One verification condition required manual intervention to prove.

Although this function is (in some sense) correctly specified, the Whiley verifying compiler will reject it. This is because its automated theorem prover (like others, such as Simplify and Z3) will not *trigger* quantifier instantiation based on arithmetic expressions. Specifically, a trigger is a pattern which the automated theorem prover uses to drive quantifier instantiation. Systems like Dafny, Spec#, and Whiley all use a predefined set of triggers which limits the ways in which quantifiers can be used successfully. Without prior knowledge of this, the inexperienced user will not understand the reason that this is failing and may be unable to proceed.

7. Benchmark evaluation

In this section, we report on an evaluation of Whiley's verification capability using a set of small benchmarks taken from previous verification competitions. Specifically, we attempted to verify four of the seven benchmarks from the VSCOMP'10 and COST'11 verification competitions [32,31]. Given the work-in-progress nature of Whiley's verifier, we were not expecting to be able to verify all benchmarks. Rather, the goal was simply to assess the current state of Whiley's verification capability.

Table 1 identifies the benchmarks considered and provides some notes on them. Benchmarks from the two competitions which operated on recursive data structures (e.g., linked lists, trees, etc.) were ignored. This is because the Whiley verifier does not currently support recursive data types, despite the fact that such structures and their invariants can be expressed in Whiley itself. Furthermore, the `nqueens` benchmark from VSCOMP'10 was omitted simply because of its size.

Table 1 includes the evaluation results, and the Whiley code for all benchmarks is given in the Appendix A. The "status" column in the table identifies the proportion of generated verification conditions which the verifier has discharged.

7.1. Discussion

These benchmarks were certainly more challenging for the verifier than the majority of simple array functions explored previously (e.g., reversing an array, enlarging an array, etc.). In general, discharging them with the verifier required considerable knowledge of the tool and the various parameter options available. For example, some benchmarks required increasing the maximum number of inference steps the verifier was permitted to take. Furthermore, discharging some verification conditions required manual intervention. This was done by exploiting the tool's ability to write generated verification conditions to disk in human readable form (recall Section 3.3). Then, premises which were not required to discharge the verification condition were commented out by hand. The following illustrates a cutdown verification condition from the `04_duplets` benchmark:

```
assert "loop invariant not restored":
    forall (int[] A, int start, int i):
        if:
            requires0(A, start)
            requires1(A, start)
            (start + 1) != |A|
            loopinvariant0(A, start, i)
            loopinvariant1(A, start, i)
            i < |A|
            A[start] != A[i]
        then:
            loopinvariant1(A, start, i + 1)
```

Please cite this article in press as: D.J. Pearce, L. Groves, Designing a verifying compiler: Lessons learned from developing Whiley, Sci. Comput. Program. (2015), http://dx.doi.org/10.1016/j.scico.2015.09.006

The `requiresX` and `loopinvariantX` macros represent the individual clauses of the precondition and loop invariant, and are numbered according to their order of occurrence. For example, `loopinvariant0` represents the first **`where`** clause of the loop invariant (line 22 of `04_duplets` in the Appendix A).

The above verification condition reestablishes one clause of the loop invariant and represents the path through the loop body taking the conditional's false branch. By inspection we determined that premises `requires0`, `requires1` and `loopinvariant0` are not needed to reestablish `loopinvariant1`. We manually commented out these three lines and this enabled the verifier to discharge the verification condition when previously it could not. Specifically, with these clauses included, the verifier would exhaust available memory. In general, eliminating clauses from verification conditions (if done safely) can reduce the verifier's search space and increases the chances of success.

Finally, another strategy used during the evaluation was to restructure programs in a way which helps the verifier. Consider the following loop invariant from the `03_maxarray` benchmark:

```
where l >= 0 && l <= u && u < |A|
// All up to lower bound below either lower or upper bound
&& all { k in 0..l | A[k] <= A[l] || A[k] <= A[u] }
// All from upper bound below either lower or upper bound
&& all { k in u+1..|A| | A[k] <= A[l] || A[k] <= A[u] }:
```

The loop invariant for this benchmark is written differently in the Appendix A, where each line of the invariant is a separate **`where`** clause rather than a separate conjunct. We discovered this seemingly innocuous difference can have a large effect in practice. To understand why, we note that separate verification conditions are generated for distinct **`where`** clauses rather than, say, distinct conjuncts. Thus, the above variation leads to a single (large) verification condition being generated, whilst the version given in the Appendix A results in three (smaller) conditions. As before, smaller verification conditions reduce the search space for the verifier, increasing its chances of success.

8. Case studies

In this section, we report on various efforts to evaluate Whiley in different settings. In particular, we have: 1) used Whiley for teaching formal methods to students at Victoria University; 2) investigated the use of Whiley in a small embedded system (Quadcopter); 3) investigated the use of Z3 as an alternative theorem prover. We now report the main findings from each of these.

8.1. Whiley for teaching

In 2014, Whiley was used as part of the assessment for a 2nd year course taught at Victoria University of Wellington. The course label and title was "*SWEN224: Formal Foundations of Programming*" and, in 2014, there were roughly 110 students enrolled. The course primarily focuses on the use of Hoare logic as a means to ensure functions meet their specifications. Students were given assignment problems which, amongst other things, included short Whiley programs to be completed. For example, in some cases, they needed to give an appropriate specification in Whiley for a function based on an English description. In other cases, they needed to find the strongest post-condition for a given function in Whiley. Students interacted with Whiley through the online tool.[3]

Overall, the use of Whiley in teaching SWEN224 was considered a success. Anecdotally, students found using Whiley interesting and helpful (compared with using just pen and paper), although a proper user experiment is needed before any concrete conclusions can be drawn. From our perspective here, the main conclusion drawn from this experience was the need for better error messages. In particular, those related to verification failures. For example, when a post-condition is not met, the compiler simply reports "`postcondition not satisfied`". Unfortunately, this does not help students understand why it is failing. Producing counterexamples which exhibit the failure would be helpful, but is currently beyond the scope of our automated theorem prover (indeed, we note that existing tools such as Z3 can produce counterexamples, though this is often limited [101]). Furthermore, verification failures do not always indicate a function's implementation is incorrect, only that the strict requirements needed for verification to succeed are not met. For example, in a correct function containing a loop with an incorrectly specified loop invariant, it's unclear whether a counterexample would actually be helpful.

A particular challenge students encountered when dealing with poor error messages was the need to identify exactly what was failing. For example, which part of a post-condition was not met. Initially, the Whiley compiler did not provide any contextual information with error messages and this made it hard to debug verification failures. To address this, support was added for reporting limited contextual information. Specifically, when a verification failure occurs indicating some part of the specification is not met, the specific `ensures` clause involved is highlighted in addition to the line containing the offending `return` statement. This gives a mechanism for systematically determining which part is failing. We note that

[3] See http://whiley.org/play.

Please cite this article in press as: D.J. Pearce, L. Groves, Designing a verifying compiler: Lessons learned from developing Whiley, Sci. Comput. Program. (2015), http://dx.doi.org/10.1016/j.scico.2015.09.006

Dafny provides similar contextual information and that further research in this area would be extremely beneficial for practical tools.

8.2. Whiley on an embedded system

We have successfully compiled and executed Whiley code on the BitCraze Crazyflie Nano Quadcopter. A detailed report of this can be found elsewhere, and we here discuss the primary conclusions and observations [102]. The Crazyflie Nano Quadcopter is based around a 32-bit ARM-Cortex microcontroller running at 72 MHz with 128 KB of Flash and 20 KB of RAM. The Crazyflie employs the widely used FreeRTOS operating system to manage on-board systems [103]. The goal of the project was to replace one of the main components in the system with equivalent code written in Whiley (as attempting to rewrite the entire system was considered out-of-scope). The stabilisation component — whose purpose is to manage stabilisation of the Quadcopter in flight — was selected for this. To further manage the project's scope, several simplifications were made:

1. **Arithmetic was treated unsoundly.** The lack of an integer range analysis within the Whiley compiler made it impossible to determine finite bounds for all integer variables. Therefore to avoid implementing this as part of the project, a simplification was made whereby Whiley's (unbounded) `int` types were simply compiled into (fixed-width) C `int` types. Whilst this is clearly unsound in the general case, it allowed us to proceed with the project.
2. **Verification was ignored.** To further reduce scope, the project focused only on the compilation and execution of Whiley code on the target platform, rather than including the additional concern of trying to verify this code. Thus, verification of the QuadCopter code developed is left as future work.

Whilst these points do limit the project's outcomes, there were nevertheless still important conclusions to be drawn. In particular, that the Whiley language currently suffers some inherent problems which make operating in such an environment difficult or impossible. These all relate to memory management, which is a specific concern on a memory-constrained device such as the Crazyflie. Specifically:

1. **No support for Stack References**. To reduce the need for dynamically allocated memory, the existing Crazyflie code base makes extensive use of stack-allocated data, with references to it being passed to called functions. Unfortunately, although Whiley supports references, there is currently no way to take the address of a stack-allocated variable. Instead, such data must be heap-allocated.
2. **No support for Deallocation**. Whiley supports heap-allocated data through the `new` operator. Unfortunately, there is currently no mechanism for explicit deallocation of heap allocated data. This presents a particular problem for a memory-constrained environment.
3. **No support for Global Variables**. The existing Crazyflie code base makes use of global variables in a number of situations. In many cases, these can be refactored away. However, in a few situations, global variables are used as a communication mechanism between FreeRTOS tasks. This is safe because such variables are only written once by the master task, whilst other tasks only ever read them. Unfortunately, Whiley has no support for global variables.

Despite these issues, the project did successfully replace the stabilisation component with an equivalent written in Whiley, and then compile and execute this code on the Crazyflie. Flight tests were conducted to ensure operation appeared identical in all respects. Whiley's *Foreign Function Interface (FFI)* was critical to this success as it provided a bridge between the existing C code and code written in Whiley and, furthermore, enabled certain workarounds for the above problems. Some comments on how the above issues were resolved:

1. **Stack Allocation**. To enable stack allocation of compound data types, the `new` operator was reinterpreted. Instead of dynamically allocating a structure on the heap, it was repurposed to return the address of stack allocated variables. This meant that allocating data on the heap required calls to the explicit memory management functions provided by FreeRTOS. Furthermore, use of the repurposed `new` operator was inherently unsafe, as Whiley's existing type system does not prevent returned references from escaping their enclosing function. As such, all uses of this operator had to be checked by hand.
2. **Global Variables**. To support global variables in a limited sense, the FFI was utilised to allow global variables to be declared in C and then accessed via impure Whiley methods.
3. **Datatype Wrangling**. The choice to compile all integer types in Whiley to C's native `int` type was outlined above. However, there were some additional problems as the existing Crazyflie code made extensive use of the full-range of data types available in C, including `unsigned int` and `short`. This presents a problem because the Whiley compiler would not generate stubs for functions which accepted other integer datatypes. Instead, C functions were written by hand to marshall and unmarshall data between code written in Whiley, versus that written in C.

The primary conclusion from this project was the need for Whiley to better support memory management. In particular, to permit references to stack-allocated data and to statically guarantee such references cannot escape their enclosing func-

Please cite this article in press as: D.J. Pearce, L. Groves, Designing a verifying compiler: Lessons learned from developing Whiley, Sci. Comput. Program. (2015), http://dx.doi.org/10.1016/j.scico.2015.09.006

tion (i.e., something similar to the notion of *object lifetimes* in Rust [104]). Likewise, there is a clear need to fully support bounded data types. In particular, although such datatypes are inherently supported through the use of type invariants, additional mechanisms are required within the compiler to ensure that compilation back-ends have easy access to such information.

8.3. Whiley verification using Z3

In an effort to improve Whiley's verification capabilities, we have also investigated the use of Z3 as an alternative automated theorem prover [51]. A complete report of this can be found elsewhere, and we here discuss the primary conclusions and observations [105]. The hope was that using Z3 might increase the range of programs which can be verified and/or increase overall performance of verification. Unfortunately, this did not turn out to be the case. The primary problems were:

1. **Impedance Mismatch.** The Whiley language supports a rich array of data types, including the use of unions, intersections, negations and (structurally defined) recursive data types. Encoding these within the context of Z3/SMT-LIB proved to be essentially impossible. In particular, there is no clear way to encode union types within either the SMT-LIB language or Z3's extensions to that language. Furthermore, although Z3 adds support for recursive data types, these are defined nominally rather than structurally and, hence, cannot accurately encode Whiley's recursive types.
2. **Fundamental Collection Types**. A further mismatch occurs between the Whiley Constraint Solver (WyCS) and Z3 regarding the fundamental unit of abstraction for representing compound data (e.g., arrays). Specifically, in WyCS, the fundamental units of abstraction are sets and tuples, whilst in Z3 it is maps. For example, in WyCS a list of `T` can be represented as a set of pairs, `(int,T)`, with appropriate constraints (i.e., that the first position of all pairs is greater-or-equal than zero, that they are all consecutive, etc.). However, in Z3, it would be represented using a map from `int` to `T` with appropriate constraints. Whilst this distinction appears small, it presents a significant hurdle when automatically translating verification conditions in WyCS to their equivalents in Z3. This is because it results in a *double encoding* of the data types. That is, sets in WyCS are encoded using maps in Z3 and, likewise, tuples in WyCS are encoded using maps in Z3. Therefore, a Whiley array is encoded as a set of pairs in the WyCS format and then encoded again as a map of maps for Z3 (roughly speaking).

We note that point (2) above could be overcome by reimplementing our verification condition generator to target Z3 directly, rather than targeting WyCS which is then translated into Z3. Finally, we also note that the proposed extension of Krönig et al. would be an extremely helpful addition to the SMT-LIB language [106].

9. Related work

We begin our discussion of related work by examining in more detail a selection of verifying compilers and associated tools which are the most comparable with Whiley. Following this, we look at related work on the topic of verification condition generation.

9.1. Verifying compilers

ESC/Java & JML The Extended Static Checker for Java (ESC/Java) is perhaps one of the most influential tools in the area of verifying compilers [7]. The ECS/Java tool was itself based on earlier work which developed the ESC/Modula-3 tool [6]. The tool essentially provides a verifying compiler for Java programs whose specifications are given as annotations in a subset of JML [68,8]. The following illustrates a simple method in JML which ESC/Java verifies as correct:

```
/*@ requires n >= 0;
  @ ensures \result >= 0;
  @*/
public static int method(int n) {
   int i = 0;
   /*@ maintaining i >= \old(i); */
   while(i < n) { i = i + 1; }
   return i;
}
```

Here, we can see pre- and post-conditions are given for the method, along with an appropriate loop invariant. Recalling our discussion from Section 5.3, `\old(i)` refers to `i` on entry to the loop and, hence, we have `\old(i)==0` holds in this case.

As briefly mentioned earlier, the ESC/Java tool makes some unsound assumptions when verifying programs. In particular, arithmetic overflow is ignored and loops are treated unsoundly by simply unrolling them for a fixed number of iterations. The tool also provides limited support for reasoning about dynamic memory through the use of ownership annotations and

Please cite this article in press as: D.J. Pearce, L. Groves, Designing a verifying compiler: Lessons learned from developing Whiley, Sci. Comput. Program. (2015), http://dx.doi.org/10.1016/j.scico.2015.09.006

`assignable` clauses for expressing frame conditions. ESC/Java has been demonstrated in some real-world settings. For example, Cataño and Huisman used it to check specifications given for an independently developed implementation of an electronic purse [107].

Unfortunately the development of JML and its associated tooling has stagnated over the last decade, although has more recently picked up again through the OpenJML initiative [108–110].

Spec# The Spec# system followed ESC/Java and benefited from many of the insights gained in that project. Spec# added proper support for handling loop invariants [10], for handling safe object initialisation [42] and allowing temporary violations of object invariants through the `expose` keyword [89]. The latter is necessary to address the so-called *packing problem* which was essentially ignored by ESC/Java [11]. Another departure from ESC/Java was the use of the BOOGIE intermediate language for verification (as opposed to guarded commands) [73], and the Z3 automated theorem prover (as opposed to Simplify) [51]. Both of these mean that Spec# is capable of verifying a wider range of programs than ESC/Java.

Although the Spec# project has now wrapped up, the authors did provide some invaluable reflections on their experiences with the project [10]. Amongst many other things, they commented that:

> "*Of the unsound features in ESC/Java, many were known to have solutions. But two open areas were how to verify object invariants in the presence of subclassing and dynamically dispatched methods (…) as well as method framing.*"

A particular concern was the issue of method re-entrancy, which is particularly challenging to model correctly. Another interesting insight given was that:

> *"If we were to do the Spec# research project again, it is not clear that extending an existing language would be the best strategy."*

The primary reason for this was the presence of constructs that are difficult for a verifier to reason about, and also the challenge for a small research group in maintaining compatibility with a large and evolving language.

Finally, the Spec# project lives on in various guises. For example, VCC verifies concurrent C code and was developed by reusing much of the tool chain from Spec# [84]. VCC has been successfully used to verify Microsoft's Hyper-V hypervisor. Likewise, Microsoft recently introduced a *Code Contracts* library in .NET 4.0 which was inspired by the Spec# project (though mostly focuses on runtime checking).

Dafny Dafny is perhaps the most comparable related work to Whiley, and was developed independently at roughly the same time [12,13]. That said, the goals of the Dafny project are somewhat different. In particular, the primary goal of Dafny to provide a proof-assistant for verifying algorithms rather than, for example, generating efficient executable code. In contrast, Whiley aims to generate code suitable for embedded systems [102,29]. Dafny is an imperative language with simple support for objects and classes without inheritance. Like Whiley, Dafny employs unbound arithmetic and distinguishes between pure and impure functions. Dafny provides algebraic data types (which are similar to Whiley's recursive data types) and supports immutable collection types with value semantics that are primarily used for ghost fields to enable specification of pointer-based programs. Dynamic memory allocation is possible in Dafny, but no explicit deallocation mechanism is given and presumably any implementation would require a garbage collector.

Unlike Whiley, Dafny also supports generic types and dynamic frames [60]. As discussed in Section 4, the latter provides a suitable mechanism for reasoning about pointer-based programs. For example, Dafny has been used successfully to verify the Schorr–Waite algorithm for marking reachable nodes in an object graph [12]. Finally, Dafny has been used to successfully verify benchmarks from the VSTTE'08 [111], VSCOMP'10 [32], VerifyThis'12 [112] challenges (and more).

VeriFast VeriFast is a modular program verifier for concurrent and sequential programs written in C and Java, and employs separation logic and fractional permissions to ensure memory safety [15,16]. The tool is unusual in eschewing the use of quantifiers within specifications. Instead, inductive predicates are provided to model properties that would otherwise be expressed using quantified formulae. VeriFast supports algebraic data types to allow specifications to reason about locations contained in linked structures. Finally, VeriFast has been used to reason about memory safety in JavaCard programs and Linux device drivers [16].

Why3 The Why3 verification system provides a specification language, called WhyML, for modelling programs written in general purpose languages [14]. This is somewhat different from the approach taken in most verification systems where a general purpose language is equipped with a specification language, typically constructed from a pure subset of the original language. Indeed, the authors of Why3 state [113]:

> *"The Why3 platform can be used by itself, as some kind of standalone "meta" theorem prover, but the main purpose of Why3 is to be used as an intermediate language."*

The Why3 system is intended to enable a range of different theorem provers to be used in proving correctness, depending on the nature of the program being verified. For example, a short but extremely intricate C program for solving the N-Queens

Please cite this article in press as: D.J. Pearce, L. Groves, Designing a verifying compiler: Lessons learned from developing Whiley, Sci. Comput. Program. (2015), http://dx.doi.org/10.1016/j.scico.2015.09.006

program has been fully verified with the aid of Why3 [114]. This was achieved by abstracting the original program into WhyML, and the proof required the use of three distinct theorem provers to discharge 41 verification conditions. Of these, 35 were discharged automatically by Alt-ERGO [115] or CVC3 [116], whilst the remainder were discharged manually using Coq [117].

Finally, the Why3 system is substantially different from the original Why platform, which was specifically designed for verification of Java or C [118]. In the former case, specifications were given in JML and in the latter case a JML-inspired specification language for C.

Frama-C Frama-C provides a set of sound software analyses for the industrial analysis of ISO C99 source code [91]. The system uses the ACSL specification language as a platform on which different solver plugins can operate. For example, different plugins may use different approaches to checking functions meet their specifications, such as abstract interpretation or deductive verification. The ACSL specification language is based loosely upon JML and supports a variant of separation logic through the `\separated` command.

Frama-C has been applied to a range of real-world problems. For example, it has been used the context of Air Traffic Management systems to reason about floating point operations and establish bounds on rounding errors [119]. Similarly, AirBus has investigated the use of Frama-C within the context of the DO-178B standard for software in airborne systems and equipment [120].

9.2. Verification condition generation

Hoare provided the foundation for formalising work in this area with his seminal paper introducing *Hoare Logic* [71]. This provides a framework for proving that a sequence of statements meets its postcondition given its precondition. Unfortunately Hoare logic does not tell us how to *construct* such a proof; rather, it gives a mechanism for *checking* a proof is correct. Therefore, to actually verify a program is correct, we need to construct proofs which satisfy the rules of Hoare logic.

The most common way to automate the process of verifying a program is with a verification condition generator. Such algorithms propagate information in either a forwards or backwards direction. However, the rules of Hoare logic lend themselves more naturally to the latter [121]. Perhaps for this reason, many tools choose to use the weakest precondition transformer. For example, the widely acclaimed ESC/Java tool computes weakest preconditions [7], as does the Why platform [118], Spec# [10], LOOP [122], JACK [123] and SnuggleBug [124]. This is surprising given that it leads to fewer verification conditions and, hence, makes it harder to generate useful error messages. To work around this, Burdy et al. embed path information in verification conditions to improve error reporting [123]. A similar approach is taken in ESC/Java, but requires support from the underlying automated theorem prover [52]. Denney and Fischer extend Hoare logic to formalise the embedding of information within verification conditions [125]. Again, their objective is to provide useful error messages.

A common technique for generating verification conditions is to transform the input program into *passive form* [6,7,83,126]. Here, the control-flow graph of each function is converted using standard techniques into a reducible (albeit potentially larger) graph. This is then further reduced by eliminating loops to leave an acyclic graph, before a final transformation into *Static Single Assignment form (SSA)* [53,127]. The main advantage is that, after this transformation, generating verification conditions becomes straightforward. Furthermore, the technique works well for unstructured control flow and can be tweaked to produced compact verification conditions [128–131].

Dijkstra's *Guarded Command Language* provides an alternative approach to the generation of verification conditions [132]. In this case, the language is far removed from the simple imperative language of Hoare logic and, for example, contains only the sequence and non-deterministic choice constructs for handling control-flow. There is a rich history of using guarded commands as an intermediate language for verification, which began with the ESC/Modula-3 tool [6]. This was continued in ESC/Java and, during the later development of Spec#, a richer version (BOOGIE) was developed [73]. Such tools use guarded commands as a way to represent programs that are in passive form (discussed above) in a human-readable manner. As these programs are acyclic, the looping constructs of Dijkstra's original language are typically ignored.

Finally, it is worth noting that Frade and Pinto provide an excellent survey of verification condition generation for simple WHILE programs [121]. They primarily focus on Hoare Logic and various extensions, but also explore Dijkstra's Guarded Command Language. They consider an extended version of Hoare's While Language which includes user-provided loop invariants. They also present an algorithm for generating verification conditions based on the weakest precondition transformer.

10. Conclusion

In this paper, we have documented several important issues faced in developing the Whiley language and its accompanying verifying compiler. We have also reflected on how our choices compare with other tools, of which Dafny is particularly notable as sharing many similarities with Whiley. Curiously, we note that Whiley and Dafny were developed independently of each other, roughly around the same time. We have also reported on efforts to verify benchmarks from two verification competitions and to demonstrate Whiley in the context of teaching and embedded systems, and reflected on the issues this has raised. Finally, we hope that by bringing our experiences together in one place will provide a useful resource for those developing verifying compilers in the future.

Please cite this article in press as: D.J. Pearce, L. Groves, Designing a verifying compiler: Lessons learned from developing Whiley, Sci. Comput. Program. (2015), http://dx.doi.org/10.1016/j.scico.2015.09.006

Acknowledgements

This work is supported by the Marsden Fund, administered by the Royal Society of New Zealand, Grant number VUW1105.

Appendix A

A.1. 01_sumax

```
type nat is (int x) where x >= 0

function sumax(int[] items) -> (int sum, int max)
requires |items| > 0
requires all { i in 0 .. |items| | items[i] >= 0 }
ensures sum <= |items| * max:
    //
    int sum = 0
    int max = 0
    int i = 0
    //
    while i < |items|
        //
        where i >= 0 && i <= |items| && sum >= 0 && max >= 0
        //
        where sum <= i * max:
        //
        sum = sum + items[i]
        if max <= items[i]:
            max = items[i]
        i = i + 1
    //
    return sum,max
```

A.2. 02_invert

```
function invert(int[] A) -> (int[] B)
// A must be injection
requires
  all { i in 0..|A|, j in 0..|A| | i != j ==> A[i] != A[j] }
// all elements of A must be within bounds
requires all { i in 0..|A| | A[i] >= 0 && A[i] < |A| }
// A and B have same size
ensures |A| == |B|
// all elements of B are withing bounds
ensures all { i in 0..|B| | 0 <= B[i] && B[i] < |B| }
// B is really inversion of A
ensures all { i in 0..|B| | B[A[i]] == i }
// B is injective
ensures
  all { i in 0..|B|, j in 0..|B| | i != j ==> B[i] != B[j] }:
    //
    int[] C = A
    int i = 0
    //
    while i < |A|
        // i must be positive
        where i >= 0 && |C| == |A| && i <= |A|
        // is inversion so far
        where all { k in 0..i | C[A[k]] == k }:
        C[A[i]] = i
        i = i + 1
    //
    return C
```

Please cite this article in press as: D.J. Pearce, L. Groves, Designing a verifying compiler: Lessons learned from developing Whiley, Sci. Comput. Program. (2015), http://dx.doi.org/10.1016/j.scico.2015.09.006

A.3. 03_maxarray

```
function max(int[] A) -> (int r)
// A non-empty array is required
requires |A| > 0
// Element index returned identifies real maximum
ensures all { i in 0..|A| | A[i] <= A[r] }:
  //
  int l = 0
  int u = |A|-1
  while l < u
    where l >= 0 && l <= u && u < |A|
    // All up to lower bound below either lower or upper bound
    where all { k in 0..l | A[k] <= A[l] || A[k] <= A[u] }
    // All from upper bound below either lower or upper bound
    where all { k in u+1..|A| | A[k] <= A[l] || A[k] <= A[u] }:
    //
    if A[l] <= A[u]:
        l = l + 1
    else:
        u = u - 1
```

A.4. 04_duplets

```
// This function finds a duplet in the given array. It is assumed
// that such a duplet exists.
function findDuplet(int[] A, int start) -> (int r1, int r2)
// Need at least two elements in the subregion
requires start >= 0 && start < |A|
// Some duplet exists in the initial array
requires some {
    i in start..|A|, j in start..|A| | i != j && A[i] == A[j]
}
// The indices returned are indeed a duplet
ensures r1 != r2 && A[r1] == A[r2]:
  //
  if start + 1 == |A|:
    // This is the base case where the postcondition follows
    // immediately from the precondition.
    return 0,1
  else:
    // This is the (potentially) recursive case. The goal here is
    // to check whether A[start] is part of a duplet or not.
    int i = start + 1
    while i < |A|
      where i > start && i <= |A|
      // A[start] doesn't match any subsequent element (thus far)
      where all { k in start+1 .. i | A[start] != A[k] }:
      //
      if A[start] == A[i]:
        return start,i
      i = i + 1
    //
    // Precondition for recursive call follows as we have ruled out
    // A[start] as being part of a duplet. Hence, at least one duplet
    // must exist in subregion beginning at start+1.
    return findDuplet(A,start+1)
```

Please cite this article in press as: D.J. Pearce, L. Groves, Designing a verifying compiler: Lessons learned from developing Whiley, Sci. Comput. Program. (2015), http://dx.doi.org/10.1016/j.scico.2015.09.006

References

[1] C. Hoare, The verifying compiler: a grand challenge for computing research, J. ACM 50 (1) (2003) 63–69.
[2] S. King, A program verifier, Ph.D. thesis, Carnegie-Mellon University, 1969.
[3] L.P. Deutsch, An interactive program verifier, Ph.D., 1973.
[4] D.I. Good, Mechanical proofs about computer programs, in: Mathematical Logic and Programming Languages, 1985, pp. 55–75.
[5] D. Luckham, S. German, F. von Henke, R. Karp, P. Milne, D. Oppen, W. Polak, W. Scherlis, Stanford Pascal verifier user manual, Technical report CS-TR-79-731, Stanford University, Department of Computer Science, 1979.
[6] D.L. Detlefs, K.R.M. Leino, G. Nelson, J.B. Saxe, Extended static checking, SRC research report 159, Compaq Systems Research Center, 1998.
[7] C. Flanagan, K. Leino, M. Lillibridge, G. Nelson, J.B. Saxe, R. Stata, Extended static checking for Java, in: Proceedings of the ACM Conference on Programming Language Design and Implementation, PLDI, 2002, pp. 234–245.
[8] G.T. Leavens, Y. Cheon, C. Clifton, C. Ruby, D.R. Cok, How the design of JML accommodates both runtime assertion checking and formal verification, Sci. Comput. Program. 55 (1–3) (2005) 185–208.
[9] D.R. Cok, J. Kiniry, ESC/Java2: uniting ESC/Java and JML, in: Proceedings of the Conference on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices, CASSIS, in: LNCS, vol. 3362, Springer-Verlag, 2005, pp. 108–128.
[10] M. Barnett, M. Fähndrich, K.R.M. Leino, P. Müller, W. Schulte, H. Venter, Specification and verification: the Spec# experience, Commun. ACM 54 (6) (2011) 81–91.
[11] M. Barnett, R. DeLine, M. Fähndrich, K.R.M. Leino, W. Schulte, Verification of object-oriented programs with invariants, J. Object Technol. 3 (6) (2004) 27–56.
[12] K.R.M. Leino, Dafny: an automatic program verifier for functional correctness, in: Proceedings of the Conference on Logic for Programming, Artificial Intelligence, and Reasoning, LPAR, in: LNCS, vol. 6355, Springer-Verlag, 2010, pp. 348–370.
[13] K.R.M. Leino, Developing verified programs with Dafny, in: Proceedings of the Conference on Verified Software: Theories, Tools, Experiments, VSTTE, in: LNCS, vol. 7152, Springer-Verlag, 2012, p. 82.
[14] J. Filliâtre, A. Paskevich, Why3 — where programs meet provers, in: Proceedings of the European Symposium on Programming, ESOP, Springer-Verlag, 2013, pp. 125–128.
[15] B. Jacobs, J. Smans, F. Piessens, A quick tour of the VeriFast program verifier, in: Proceedings of the Asian Symposium on Programming Languages and Systems, APLAS, 2010, pp. 304–311.
[16] B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, F. Piessens, VeriFast: a powerful, sound, predictable, fast verifier for C and Java, in: Proceedings of the NASA Formal Methods Symposium, Springer-Verlag, 2011, pp. 41–55.
[17] D. Monniaux, The pitfalls of verifying floating-point computations, ACM Trans. Program. Lang. Syst. 30 (3) (2008) 12.
[18] K.R.M. Leino, Extended static checking: a ten-year perspective, in: Informatics — 10 Years Back, 10 Years Ahead, in: LNCS, vol. 2000, 2001, pp. 157–175.
[19] G. Kulczycki, H. Keown, M. Sitaraman, B.W. Weide, Abstracting pointers for a verifying compiler, in: Proceedings of the Software Engineering Workshop, SEW, IEEE Computer Society Press, 2007, pp. 204–213.
[20] K.R.M. Leino, P. Müller, Object invariants in dynamic contexts, in: Proceedings of the European Conference on Object-Oriented Programming, ECOOP, 2004, pp. 491–516.
[21] B. Cook, A. Podelski, A. Rybalchenko, Proving program termination, Commun. ACM (2011) 88–98.
[22] J. Smans, B. Jacobs, F. Piessens, VeriCool: an automatic verifier for a concurrent object-oriented language, in: Formal Methods for Open Object-Based Distributed Systems, FMOODS, 2008, pp. 220–239.
[23] E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, S. Tobies, VCC: a practical system for verifying concurrent C, in: Proceedings of the Conference on Theorem Proving in Higher Order Logics, TPHOL, 2009, pp. 23–42.
[24] The Whiley programming language, http://whiley.org.
[25] D.J. Pearce, L. Groves, Whiley: a platform for research in software verification, in: Proceedings of the Conference on Software Language Engineering, SLE, 2013, pp. 238–248.
[26] D.J. Pearce, The Whiley Rewrite Language (WyRL), in: Proceedings of the Conference on Software Language Engineering, SLE, 2015 (in press).
[27] D.J. Pearce, J. Noble, Implementing a language with flow-sensitive and structural typing on the JVM, Electron. Notes Theor. Comput. Sci. 279 (1) (2011) 47–59.
[28] D.J. Pearce, Sound and complete flow typing with unions, intersections and negations, in: Proceedings of the Conference on Verification, Model Checking, and Abstract Interpretation, VMCAI, 2013, pp. 335–354.
[29] D.J. Pearce, Integer range analysis for Whiley on embedded systems, in: Proceedings of the IEEE/IFIP Workshop on Software Technologies for Future Embedded and Ubiquitous Systems, 2015, pp. 26–33.
[30] C. Dymnikov, D.J. Pearce, A. Potanin, OwnKit: inferring modularly checkable ownership annotations for Java, in: Proceedings of the Australasian Software Engineering Conference, ASWEC, 2013, pp. 181–190.
[31] T. Bormer, M. Brockschmidt, D. Distefano, G. Ernst, J.-C. Filliâtre, R. Grigore, M. Huisman, V. Klebanov, C. Marché, R. Monahan, W. Mostowski, N. Polikarpova, C. Scheben, G. Schellhorn, B. Tofan, J. Tschannen, M. Ulbrich, The COST IC0701 verification competition 2011, in: Proceedings of the Workshop on Formal Verification of Object-Oriented Software, FoVeOOS, in: LNCS, vol. 7421, Springer-Verlag, 2011, pp. 3–21.
[32] V. Klebanov, P. Müller, N. Shankar, G.T. Leavens, V. Wüstholz, E. Alkassar, R. Arthan, D. Bronish, R. Chapman, E. Cohen, M. Hillebrand, B. Jacobs, K.R.M. Leino, R. Monahan, F. Piessens, N. Polikarpova, T. Ridge, J. Smans, S. Tobies, T. Tuerk, M. Ulbrich, B. Weiß, The 1st verified software competition: experience report (VSComp), in: Proceedings of the Symposium on Formal Methods, FM, in: LNCS, Springer-Verlag, 2011.
[33] D.J. Pearce, The Whiley language specification, Updated, 2014.
[34] D.J. Pearce, Getting started with Whiley, Last updated, 2014.
[35] S. Tobin-Hochstadt, M. Felleisen, Logical types for untyped languages, in: Proceedings of the ACM International Conference on Functional Programming, ICFP, 2010, pp. 117–128.
[36] A. Guha, C. Saftoiu, S. Krishnamurthi, Typing local control and state using flow analysis, in: Proceedings of the European Symposium on Programming, ESOP, 2011, pp. 256–275.
[37] D.J. Pearce, A calculus for constraint-based flow typing, in: Proceedings of the Workshop on Formal Techniques for Java-Like Programs, FTFJP, 2013, p. 7.
[38] F. Barbanera, M.D.-C. Caglini, Intersection and union types, in: Proceedings of the Conference on Theoretical Aspects of Computer Software, TACS, 1991, pp. 651–674.
[39] A. Igarashi, H. Nagira, Union types for object-oriented programming, J. Object Technol. 6 (2) (2007) 47–68.
[40] B.C. Pierce, Types and Programming Languages, MIT Press, 2002.
[41] T.Hoare, Null references: the billion dollar mistake, in: QCon, 2009, presentation.
[42] M. Fähndrich, K.R.M. Leino, Declaring and checking non-null types in an object-oriented language, in: Proceedings of the ACM conference on Object-Oriented Programming, Systems, Languages and Applications, OOPSLA, ACM Press, 2003, pp. 302–312.
[43] T. Ekman, G. Hedin, Pluggable checking and inferencing of non-null types for Java, J. Object Technol. 6 (9) (2007) 455–475.

Please cite this article in press as: D.J. Pearce, L. Groves, Designing a verifying compiler: Lessons learned from developing Whiley, Sci. Comput. Program. (2015), http://dx.doi.org/10.1016/j.scico.2015.09.006

[44] P. Chalin, P.R. James, Non-null references by default in Java: alleviating the nullity annotation burden, in: Proceedings of the European Conference on Object-Oriented Programming, ECOOP, 2007, pp. 227–247.
[45] C. Male, D. Pearce, A. Potanin, C. Dymnikov, Java bytecode verification for @NonNull types, in: Proceedings of the conference on Compiler Construction, CC, 2008, pp. 229–244.
[46] L. Hubert, A non-null annotation inferencer for Java bytecode, in: Proceedings of the Workshop on Program Analysis for Software Tools and Engineering, PASTE, ACM Press, 2008, pp. 36–42.
[47] M. Cielecki, J. Fulara, K. Jakubczyk, L. Jancewicz, Propagation of JML non-null annotations in Java programs, in: Proceedings of the Conference on Principles and Practices of Programming in Java, PPPJ, 2006, pp. 135–140.
[48] L. Hubert, T. Jensen, D. Pichardie, Semantic foundations and inference of non-null annotations, in: Proceedings of the International Conference on Formal Methods for Open Object-Based Distributed Systems, FMOODS, Springer-Verlag, 2008, pp. 132–149.
[49] ISO/IEC, International standard ISO/IEC 9899, programming languages — C, 1990.
[50] D.J. Pearce, Practical verification condition generation for a bytecode language, Tech. rep. ECSTR14-07, Victoria University of Wellington, 2014.
[51] L. de Moura, N. Bjørner, Z3: an efficient SMT solver, in: Proceedings of the conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS, 2008, pp. 337–340.
[52] D. Detlefs, G. Nelson, J.B. Saxe, Simplify: a theorem prover for program checking, J. ACM 52 (3) (2005) 365–473.
[53] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, F.K. Zadeck, An efficient method of computing static single assignment form, in: Proceedings of the ACM symposium on the Principles Of Programming Languages, POPL, 1989, pp. 25–35.
[54] R.W. Floyd, Assigning meaning to programs, in: Proceedings of Symposia in Applied Mathematics, vol. 19, American Mathematical Society, 1967, pp. 19–31.
[55] D.J. Pearce, JPure: a modular purity system for Java, in: Proceedings of the conference on Compiler Construction, CC, 2011, pp. 104–123.
[56] A. Rountev, Precise identification of side-effect-free methods in Java, in: Proceedings of the International Conference on Software Maintenance, ICSM, IEEE Computer Society, 2004, pp. 82–91.
[57] A. Salcianu, M. Rinard, Purity and side effect analysis for Java programs, in: Proceedings of the Conference on Verification, Model Checking, and Abstract Interpretation, VMCAI, 2005, pp. 199–215.
[58] A. Milanova, A. Rountev, B.G. Ryder, Parameterized object sensitivity for points-to and side-effect analyses for Java, in: Proceedings of the International Symposium on Software Testing and Analysis, ISSTA, ACM, 2002, pp. 1–11.
[59] D. Cok, G.T. Leavens, Extensions of the theory of observational purity and a practical design for JML, in: Workshop on Specification and Verification of Component-Based Systems, SAVCBS, 2008, pp. 43–50.
[60] I.T. Kassios, Dynamic frames: support for framing, dependencies and sharing without restrictions, in: Proceedings of the Symposium on Formal Methods, FM, in: LNCS, vol. 4085, Springer-Verlag, 2006, pp. 268–283.
[61] N. Lameed, L.J. Hendren, Staged static techniques to efficiently implement array copy semantics in a MATLAB JIT compiler, in: Proceedings of the Conference on Compiler Construction, CC, 2011, pp. 22–41.
[62] N. Shankar, Static analysis for safe destructive updates in a functional language, in: Proceedings of the Symposium Logic-Based Program Synthesis and Transformation, LOPSTR, 2001, pp. 1–24.
[63] M. Odersky, How to make destructive updates less destructive, in: Proceedings of the ACM symposium on the Principles Of Programming Languages, POPL, 1991, pp. 25–36.
[64] R.E. Bryant, D. Kroening, J. Ouaknine, S.A. Seshia, O. Strichman, B.A. Brady, Deciding bit-vector arithmetic with abstraction, in: Proceedings of the conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS, 2007, pp. 358–372.
[65] B. Jacobs, J. Smans, F. Piessens, The VeriFast program verifier: a tutorial, Tech. rep., 2014.
[66] F. Nielson, H.R. Nielson, C.L. Hankin, Principles of Program Analysis, Springer-Verlag, 1999.
[67] E. Darulova, V. Kuncak, Sound compilation of reals, in: Proceedings of the ACM symposium on the Principles Of Programming Languages, POPL, ACM Press, 2014, pp. 235–248.
[68] P. Chalin, F. Rioux, JML runtime assertion checking: improved error reporting and efficiency using strong validity, in: Proceedings of the Symposium on Formal Methods, FM, in: LNCS, vol. 5014, Springer-Verlag, 2008, pp. 246–261.
[69] S. Jefferson, S.N. Kamin, Executable specifications with quantifiers in the FASE system, in: Proceedings of the ACM symposium on the Principles Of Programming Languages, POPL, ACM, 1986, pp. 318–326.
[70] N.E. Fuchs, Specifications are (preferably) executable, Softw. Eng. J. 7 (5) (1992) 323–334.
[71] C. Hoare, An axiomatic basis for computer programming, Commun. ACM 12 (10) (1969) 576–583.
[72] A. Ireland, B.J. Ellis, T. Ingulfsen, Invariant patterns for program reasoning, in: Proceedings of the Mexican International Conference on Artificial Intelligence, MICAI, in: LNCS, vol. 2972, Springer-Verlag, 2004, pp. 190–201.
[73] M. Barnett, B.E. Chang, R. DeLine, B. Jacobs, K.R.M. Leino, Boogie: a modular reusable verifier for object-oriented programs, in: Proceedings of the Formal Methods for Components and Objects, FMCO, 2006, pp. 364–387.
[74] C.A. Furia, B. Meyer, Inferring loop invariants using postconditions, CoRR, arXiv:0909.0884.
[75] N. Polikarpova, I. Ciupa, B. Meyer, A comparative study of programmer-written and automatically inferred contracts, in: Proceedings of the International Symposium on Software Testing and Analysis, ISSTA, ACM Press, 2009, pp. 93–104.
[76] C. Flanagan, S. Qadeer, Predicate abstraction for software verification, in: Proceedings of the ACM Symposium on the Principles Of Programming Languages, POPL, ACM Press, 2002, pp. 191–202.
[77] R. Chadha, D.A. Plaisted, On the mechanical derivation of loop invariants, J. Symb. Comput. 15 (5 & 6) (1993) 705–744.
[78] K.R.M. Leino, F. Logozzo, Loop invariants on demand, in: Proceedings of the Asian Symposium on Programming Languages and Systems, APLAS, in: LNCS, vol. 3780, Springer-Verlag, 2005, pp. 119–134.
[79] M.-V. Aponte, P. Courtieu, Y. Moy, M. Sango, Maximal and compositional pattern-based loop invariants, in: Proceedings of the Symposium on Formal Methods, FM, in: LNCS, vol. 7436, Springer-Verlag, 2012, pp. 37–51.
[80] D. Cachera, T.P. Jensen, A. Jobin, F. Kirchner, Inference of polynomial invariants for imperative programs: a farewell to Gröbner bases, in: Proceedings of the Static Analysis Symposium, SAS, in: LNCS, vol. 7460, Springer-Verlag, 2012, pp. 58–74.
[81] F. Bobot, J.-C. Filliâtre, C. Marché, G. Melquiond, A. Paskevich, The Why3 platform, v0.85, 2014.
[82] B. Beckert, S. Schlager, P.H. Schmitt, An improved rule for while loops in deductive program verification, in: Proceedings of the International Conference on Formal Engineering Methods, ICFEM, 2005, pp. 315–329.
[83] M. Barnett, K.R.M. Leino, Weakest-precondition of unstructured programs, in: Proceedings of the Workshop on Program Analysis for Software Tools and Engineering, PASTE, ACM Press, 2005, pp. 82–87.
[84] E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, S. Tobies, VCC: a practical system for verifying concurrent C, in: Proceedings of the Conference on Theorem Proving in Higher Order Logics, TPHOL, in: LNCS, vol. 5674, Springer-Verlag, 2009, pp. 23–42.
[85] P. Chalin, Adjusted verification rules for loops are more complete and give better diagnostics for less, in: Proceedings of the Conference on Software Engineering and Formal Methods, SEFM, IEEE Computer Society Press, 2009, pp. 317–324.
[86] C.A.R. Hoare, N. Wirth, An axiomatic definition of the programming language PASCAL, Acta Inform. 2 (4) (1973) 335–355.

Please cite this article in press as: D.J. Pearce, L. Groves, Designing a verifying compiler: Lessons learned from developing Whiley, Sci. Comput. Program. (2015), http://dx.doi.org/10.1016/j.scico.2015.09.006

[87] S. Alagic, M.A. Arbib, The Design of Well-Structured and Correct Programs, Springer-Verlag, 1978.
[88] E. Cohen, Move invariants to end of do-while loop?, http://vcc.codeplex.com/discussions/271749.
[89] K.R.M. Leino, P. Müller, Using the Spec# language, methodology, and tools to write bug-free programs, in: LASER Summer School, in: LNCS, vol. 6029, Springer-Verlag, 2008, pp. 91–139.
[90] A. Betts, N. Chong, A.F. Donaldson, S. Qadeer, P. Thomson, GPUVerify: a verifier for GPU kernels, in: Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages and Applications, OOPSLA, ACM Press, 2012, pp. 113–132.
[91] P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, B. Yakobowski, Frama-C: a software analysis perspective, in: Proceedings of the Conference on Software Engineering and Formal Methods, SEFM, in: LNCS, vol. 7504, Springer-Verlag, 2012, pp. 233–247.
[92] S. Gulwani, A. Tiwari, Assertion checking over combined abstraction of linear arithmetic and uninterpreted functions, in: Proceedings of the European Symposium on Programming, ESOP, in: LNCS, vol. 3924, 2006, pp. 279–293.
[93] W. Sonnex, S. Drossopoulou, S. Eisenbach, Zeno: an automated prover for properties of recursive data structures, in: Proceedings of the Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS, in: LNCS, vol. 7214, Springer-Verlag, 2012, pp. 407–421.
[94] M. Schwerhoff, A.J. Summers, Lightweight support for magic wands in an automatic verifier, in: Proceedings of the European Conference on Object-Oriented Programming, ECOOP, in: LIPIcs, vol. 37, 2015, pp. 614–638.
[95] Y. Ge, C. Barrett, C. Tinelli, Solving quantified verification conditions using satisfiability modulo theories, in: Proceedings of the Conference on Automated Deduction, CADE, in: LNCS, vol. 4603, Springer-Verlag, 2007, pp. 167–182.
[96] K.R.M. Leino, R. Monahan, Automatic verification of textbook programs that use comprehensions, in: Proceedings of the Workshop on Formal Techniques for Java-like Programs, FTFJP, 2007.
[97] S.K. Lahiri, S. Qadeer, Back to the future: revisiting precise program verification using SMT solvers, in: Proceedings of the ACM Symposium on the Principles Of Programming Languages, POPL, 2008, pp. 171–182.
[98] N. Bjørner, L. de Moura, Z3[10]: applications, enablers, challenges and directions, in: Proceedings of the Workshop on Constraints in Formal Verification, 2009.
[99] M. Moskal, Programming with triggers, in: Proceedings of the International Workshop on Satisfiability Modulo Theories, SMT, ACM Press, 2009, pp. 20–29.
[100] A. Kaldewau, Programming: The Derivation of Algorithms, International Series in Computer Science, Prentice–Hall, 1990.
[101] C. Gladisch, Satisfiability solving and model generation for quantified first-order logic formulas, in: Proceedings of the Conference on Formal Verification of Object-Oriented Software, in: LNCS, vol. 6528, Springer-Verlag, 2010, pp. 76–91.
[102] M. Stevens, Demonstrating Whiley on an embedded system, Tech. rep, School of Engineering and Computer Science, Victoria University of Wellington, 2014, http://www.ecs.vuw.ac.nz/~djp/files/MattStevensENGR489.pdf.
[103] FreeRTOS homepage, http://freertos.org/.
[104] R. Poss, Rust for functional programmers, CoRR, arXiv:1407.5670.
[105] H. Wylde, Verifying Whiley programs using an off-the-shelf SMT solver, Tech. rep, School of Engineering and Computer Science, Victoria University of Wellington, 2014, http://ecs.victoria.ac.nz/~djp/files/HenryWyldeENGR489.pdf.
[106] D. Kröning, P. Rümmer, G. Weissenbacher, A proposal for a theory of finite sets, lists, and maps for the SMT-lib standard, in: Proceedings of the Conference on Automated Deduction, CADE, 2009.
[107] N. Cataño, M. Huisman, Formal specification and static checking of Gemplus' electronic purse using ESC/Java, in: Proceedings of the Symposium on Formal Methods Europe, FME, in: LNCS, vol. 2391, Springer-Verlag, 2002, pp. 272–289.
[108] D.R. Cok, OpenJML: software verification for Java 7 using JML, OpenJDK, and eclipse, in: Proceedings of the Workshop on Formal Integrated Development Environment, F-IDE, in: LNCS, vol. 149, 2014, pp. 79–92.
[109] J. Sánchez, G.T. Leavens, Static verification of PtolemyRely programs using OpenJML, in: Proceedings of the Workshop on Foundations of Aspect-Oriented Languages, FOAL, ACM Press, 2014, pp. 13–18.
[110] D.R. Cok, OpenJML: JML for Java 7 by extending OpenJDK, in: Proceedings of the NASA Formal Methods Symposium, in: LNCS, vol. 6617, Springer-Verlag, 2011, pp. 472–479.
[111] K.R.M. Leino, R. Monahan, Dafny meets the verification benchmarks challenge, in: Proceedings of the Conference on Verified Software: Theories, Tools, Experiments, VSTTE, in: LNCS, vol. 6217, Springer-Verlag, 2010, pp. 112–126.
[112] M.Huisman, V.Klebanov, R.Monahan, VerifyThis verification competition 2012, Organizer's report, 2013.
[113] F. Bobot, J.-C. Filliâtre, C. Marché, A. Paskevich, Why3: shepherd your herd of provers, in: Proceedings of the Workshop on Intermediate Verification Languages, BOOGIE, 2011.
[114] J.-C. Filliâtre, Verifying two lines of C with Why3: an exercise in program verification, in: Proceedings of the Conference on Verified Software: Theories, Tools, Experiments, VSTTE, in: LNCS, vol. 7152, Springer-Verlag, 2012, pp. 83–97.
[115] The Alt-Ergo automated theorem prover, http://alt-ergo.lri.fr/.
[116] C. Barrett, C. Tinelli, CVC3, in: Proceedings of Conference on Computer Aided Verification, CAV, 2007, pp. 298–302.
[117] Y. Bertot, P. Castéran, Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions, Texts in Theoretical Computer Science, Springer-Verlag, 2004.
[118] J.-C. Filliâtre, C. Marché, The Why/Krakatoa/Caduceus platform for deductive program verification, in: Proceedings of Conference on Computer Aided Verification, CAV, in: LNCS, vol. 4590, Springer-Verlag, 2007, pp. 173–177.
[119] A.E. Goodloe, C. Muñoz, F. Kirchner, L. Correnson, Verification of numerical programs: from real numbers to floating point numbers, in: Proceedings of the NASA Formal Methods Symposium, in: LNCS, vol. 7871, Springer-Verlag, 2013, pp. 441–446.
[120] J. Souyris, V. Wiels, D. Delmas, H. Delseny, Formal verification of avionics software products, in: Proceedings of the Symposium on Formal Methods, FM, in: LNCS, vol. 5850, Springer-Verlag, 2009, pp. 532–546.
[121] M.J. Frade, J.S. Pinto, Verification conditions for source-level imperative programs, Comput. Sci. Rev. 5 (3) (2011) 252–277.
[122] B. Jacobs, Weakest pre-condition reasoning for Java programs with JML annotations, J. Log. Algebr. Program. 58 (1–2) (2004) 61–88.
[123] L. Burdy, A. Requet, J.-L. Lanet, Java applet correctness: a developer-oriented approach, in: Proceedings of the Symposium on Formal Methods Europe, FME, in: LNCS, vol. 2805, Springer-Verlag, 2003, pp. 422–439.
[124] S. Chandra, S.J. Fink, M. Sridharan, Snugglebug: a powerful approach to weakest preconditions, in: Proceedings of the ACM conference on Programming Language Design and Implementation, PLDI, ACM Press, 2009, pp. 363–374.
[125] E. Denney, B. Fischer, Explaining verification conditions, in: Proceedings of the Conference on Algebraic Methodology and Software Technology, AMAST, in: LNCS, vol. 5140, Springer, 2008, pp. 145–159.
[126] R. Grigore, J. Charles, F. Fairmichael, J. Kiniry, Strongest postcondition of unstructured programs, in: Proceedings of the Workshop on Formal Techniques for Java-like Programs, FTFJP, ACM Press, 2009, pp. 6:1–6:7.
[127] R. Cytron, J. Ferrante, B.K. Rosen, M.N. Wegman, F.K. Zadeck, Efficiently computing static single assignment form and the control dependence graph, ACM Trans. Program. Lang. Syst. 13 (4) (1991) 451–490.
[128] C. Flanagan, J.B. Saxe, Avoiding exponential explosion: generating compact verification conditions, in: Proceedings of the ACM symposium on the Principles Of Programming Languages, POPL, ACM Press, 2001, pp. 193–205.

Please cite this article in press as: D.J. Pearce, L. Groves, Designing a verifying compiler: Lessons learned from developing Whiley, Sci. Comput. Program. (2015), http://dx.doi.org/10.1016/j.scico.2015.09.006

[129] K.R.M. Leino, Efficient weakest preconditions, Inf. Process. Lett. 93 (6) (2005) 281–288.
[130] D. Babic, A.J. Hu, Structural abstraction of software verification conditions, in: Proceedings of Conference on Computer Aided Verification, CAV, in: LNCS, vol. 4590, Springer-Verlag, 2007, pp. 366–378.
[131] D. Babic, A.J. Hu, Exploiting shared structure in software verification conditions, in: Haifa Verification Conference, in: LNCS, vol. 4899, Springer, 2007, pp. 169–184.
[132] E.W. Dijkstra, Guarded commands, nondeterminancy and formal derivation of programs, Commun. ACM 18 (1975) 453–457.

Please cite this article in press as: D.J. Pearce, L. Groves, Designing a verifying compiler: Lessons learned from developing Whiley, Sci. Comput. Program. (2015), http://dx.doi.org/10.1016/j.scico.2015.09.006