Integer Range Analysis for Whiley on Embedded Systems

David J. Pearce
School of Engineering and Computer Science
Victoria University of Wellington, New Zealand
Email: djp@ecs.vuw.ac.nz

***Abstract*—Programs written in the Whiley programming language are verified at compile-time to ensure all function specifications are met. The purpose of doing this is to eliminate as many software bugs as possible and, thus, Whiley is ideally suited for use in safety-critical systems. The language was designed from scratch to simplify verification as much as possible. To that end, arithmetic types in Whiley consist of unbounded integers and rationals and this poses a problem for use in memory constrained embedded devices. However, function specifications in Whiley provide a rich source of information from which finite bounds for integer variables can be determined. In this paper, we present a technique for range analysis of integer variables in Whiley. Previous work is typically based on dataflow analysis which requires a fixed-point computation and necessitates the use of imprecise widenings to ensure termination. However, the presence of loop and data type invariants in Whiley means that loops can be handled quickly and precisely.**

***Keywords*-Integer range analysis; embedded systems; compilers**

I. Introduction

The Whiley programming language has been developed from the ground up to enable compile-time verification of its programs [1], [2], [3], [4]. The Whiley Compiler (WyC) attempts to ensure that all functions in a program meet their specifications. When it succeeds in this endeavour, we know that: 1) all function post-conditions are met (assuming their pre-conditions held on entry); 2) all invocations meet their respective function's pre-condition; 3) runtime errors such as divide-by-zero, out-of-bounds accesses and null-pointer dereferences are impossible. Note, however, that such programs may still loop indefinitely and/or exhaust available resources (e.g. RAM).

Whiley's verification system makes it ideally suited for use with safety-critical systems. However, there remain several hurdles related to the compilation of Whiley programs for memory-constrained embedded environments. This paper addresses one of these problems, namely that of identifying finite bounds for all integer variables. This problem arises because of the decision to make all arithmetic in Whiley unbounded. From a verification perspective this decision makes sense as, for example, 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 [5], whilst systems like ESC/Java and Spec# assume (unsoundly) that numeric types do not overflow or suffer rounding [6], [7]. In contrast, Dafny — a comparable tool to Whiley — makes the same decision to support unbounded arithmetic, although appears unconcerned with efficient execution.

The presence of unbound arithmetic in Whiley presents a challenge, namely: *how to determine finite ranges for all variables in a program?* If this can be done then, in principle at least, one can safely compile the program for a memory-constrained embedded system. In this paper, we focus only on integer variables and, hence, this is the well-known problem of *integer range analysis* [8]. The majority of existing work on this problem has focused on determining finite ranges for programs written in traditional languages, such as C or Java [9], [10], [11], [12], [13]. As such, the problem is quite different from that studied here. In particular, such approaches assume no information is available regarding the ranges of variables and, hence, everything must be computed from scratch which requires an interprocedural data-flow analysis. This is both costly and suffers from the inherent problem of ensuring termination in the presence of the (essentially infinite) lattice of integer ranges [8]. Typically, for example, a widening operator is triggered after a certain number of iterations to ensure a coarse over-approximation is achieved.

Whiley programs offer several advantages when performing integer range analysis, compared with traditional languages. The presence of function specifications eliminates the need for a costly interprocedural analysis. Furthermore, the presence of loop and data type invariants eliminates the need for a fixed-point computation and the associated termination problem. Nevertheless, there remain challenges to overcome. In particular, specifications in Whiley may be arbitrarily complex and do not necessarily correspond to finite ranges. Instead, we must conservatively extract finite ranges from specifications for all variables. In such case that this is impossible, then the program cannot be successfully compiled for a memory constrained embedded system. This can happen, for example, if the programmer forgets to ensure that one or more variables are finitely bounded. However, this can always be resolved by updating the relevant specifications to ensure this is so.

The contributions of this paper are:

1) We present an algorithm for computing ranges for variables of integer type, including list bounds. This exploits information of function specifications and type invariants available in the Whiley programming language.
2) We have developed a prototype implementation as a module for the Whiley compiler. This is released under an open source license (BSD) and is available to download from http://github.com/Whiley/Whiley2EmbeddedC.

Finally, all examples in this paper have been tested against the latest release at the time of writing ($v0.3.32$). Furthermore, they can be verified and executed in the browser at http://whiley.org/play/.

II. Overview

In this section, we progressively introduce the problem of determining finite ranges for integer variables in Whiley. In doing this, we also provide some introduction to the language itself, however this is necessarily brief and the interested reader may find more detailed introductions elsewhere [14].

A. Type Invariants

The simplest form of specification in Whiley is provided through support for *data type invariants*. These allow the user to define data types which are further constrained in some way. The following illustrates two examples:

```
// The type of natural numbers
type nat is (int x) where x >= 0

// The type of 16bit unsigned integers
type u16 is (int x) where x >= 0 && x <= 65535
```

Here, the **type** declaration includes a **where** clause constraining the permissible values for the type. The declared variable (i.e. `x`) is used to represent an arbitrary value of the given type.

The purpose of an integer range analysis is to conservatively determine the possible range of values for integer variables. For our examples above, it should be clear that the range of values for `nat` is $[0, \infty]$ and, for `u16`, it is $[0, 65535]$. The latter is of most interest to us in this paper as it represents a finite range which can be encoded within a 16bit word. We introduce the following notation for describing integer ranges:

Definition 1 (Integer Range)**.** *Let* $\text{int}[l, u]$ *denote a range of integer values where* l *and* u *are either integer constants or* $\pm\infty$*, and where* $\text{int}[l, u] = \{x \mid x \in \mathbb{Z} \wedge l \leq x \wedge x \leq u\}$.

Although it is easy enough to see the connection between the Whiley data types defined above and their integer ranges, this is not always the case. In particular, Whiley data types may have finite bounds of arbitrary granularity as the following illustrates:

```
// The type of even numbers
type even is (int x) where x % 2 == 0

// A type consisting of the split finite ranges [-20,-10] and [10,20]
type split is (int x) where (x >= 10 && x <= 20)
                         || (x >= -20 && x <= -10)
```

The smallest integer range which includes the type `even` is $\text{int}[-\infty, \infty]$, whilst for `split` it is $\text{int}[-20, 20]$. Both of these ranges are conservative (i.e. sound) as they include all values of the types in question, but they are necessarily imprecise. From the perspective of this paper, this loss of precision is not particularly of concern as the goal is simply to determine whether a variable can be implemented in a finite number of bits (and, if so, how many bits are required).

B. Preconditions and Postconditions

Whiley allows explicit *pre-* and *post-conditions* to be given for functions. For example, the following function determines the maximum of two 16bit unsigned integers:

```
function max(int x, int y) -> (u16 r)
// restrict ranges of parameters
requires 0 <= x && x <= 65535
requires 0 <= y && y <= 65535
// ensure specification is met
ensures r == x || r == y
ensures r >= x && r >= y:
    //
    if x >= y:
        return x
    else:
        return y
```

Here, the function `max()` includes **requires** and **ensures** clauses which correspond (respectively) to its *precondition* and *postcondition*. In this case, multiple **requires** (resp. **ensures**) clauses are given which are conjoined together to form the function's precondition (resp. postcondition). Furthermore, `r` represents the return value and may be used only within the **ensures** clause(s). The Whiley compiler verifies at compile-time that this function meets its specification.

In practice, we would have normally declared parameters `x` and `y` to have type `u16` above. However, by not doing so, we illustrate the connection between type invariants and function specifications — namely, that the former can be encoded directly in the latter. Thus, for completeness, we cannot just generate ranges by analysing type invariants; rather, we must analyse pre- and post-conditions as well.

C. Arrays

Arrays of data are represented in Whiley using arbitrary length lists. The following illustrates a simple example:

```
// The type of extended 16bit unsigned integers
type u16e is (int x) where x >= -1 && x <= 65535

function indexOf([u16] xs, u16 x) -> u16e
// Restrict size of input array
requires |xs| <= 65535:
    //
    u16 i = 0
    //
    while i < |xs|:
        if xs[i] == x:
            return i
        i = i + 1
    //
    return -1 // not found
```

The Whiley compiler determines at compile time that this function meets its specification. Furthermore, the example illustrates a number of interesting points about Whiley:

- **Bounds Checking.** Aside from ensuring that the above function meets its specification, the Whiley compiler also ensures the access `xs[i]` is within the bounds of list `xs`.

- **Extended Types.** The type `u16e` has been used for the return value to allow every valid list index to be returned, along with an additional value (`-1`) to signal no element was found. This type requires a minimum of 17bits to encode. Furthermore, code which invokes this function is forced to handle the error case as, for example, assigning the return value to a variable of type `u16` generates a compile-time error.

- **Finite Lists.** The `xs` list is declared to contain elements of type `u16`. We have further constrained it to ensure that the list itself is finite as, without this, we could not constrain the range for the return type.

Reasoning about the size of a Whiley list requires considering both the size of its elements, and the maximum length of the list itself. Therefore, we introduce the following notation for this purpose:

Definition 2 (List Range)**.** *Let* $\text{list}\langle T\rangle[n]$ *denote a list type where* T *is a range type and* n *a non-negative integer constant or* ∞*, and where* $\text{list}\langle T\rangle[n] = \{\ x_0 \times ... \times x_m \mid \forall_k.(x_k \in T) \wedge m < n\}$.

As an aside, we should comment that compound data types in Whiley (e.g. lists) have *value semantics*. This means they are passed and returned by-value (as in Pascal, MATLAB or most functional languages). Whiley also supports reference types and, as in C/C++, these must be explicitly demarcated. For example, `&[int]` is a *reference* to a list of integers. However, for the purposes of this paper these types, along with a plethora of other interesting types in Whiley (e.g. tuples, records and unions), are ignored.

D. Loop Invariants

Whiley supports explicit loop invariants which are necessary to prove many useful properties about programs with loops. As with function specifications, these can contain information relevant to the ranges of variables within a function. Consider again our example from above, but this time with an alternate declaration for variable `i`:

```
function indexOf([u16] xs, u16 x) -> u16e
// Restrict size of input array
requires |xs| <= 65535:
    //
    int i = 0
    //
    while i < |xs| where i >= 0:
        if xs[i] == x:
            return i
        i = i + 1
    //
    return -1 // not found
```

Here, the loop invariant is required to aid the Whiley compiler in reasoning about variable `i` within the loop. However, it also serves a double purpose, from the perspective of this paper, in that it encodes information about the variable's range. Specifically, by combining the knowledge that `i < |xs|` and `i >= 0` we can determine a range of $\mathrm{int}[0, 65535]$ for `i` within the loop body.

III. Formalisation of Range Types

In this section, we provide a formalisation of range types and operations upon them. We consider here only a small subset of types available in Whiley, although extensions to the remainder are mostly straightforward. Furthermore, our presentation of integer ranges is mostly straightforward and similar to previous work (e.g. [9], [15], [16], [17]). However, some differences do exist. Firstly, we can assume that the programs being analysed have already been verified and, hence, that division-by-zero and out-of-bounds accesses are impossible. Secondly, we additionally consider the representation of lists and their length bounds.

A. Range Types

The language of range types under consideration in this paper is as follows:

$$\begin{array}{rcl} T & ::= & \mathrm{int}[c^-, c^+] \mid \mathrm{list}\langle T\rangle[c^+] \\ c^+ & ::= & +\infty \mid 0 \mid 1 \mid 2 \mid 3 \mid \ldots \\ c^- & ::= & -\infty \mid -1 \mid -2 \mid -3 \mid \ldots \end{array}$$

Recall the integer and list ranges defined previously in Definitions 1 + 2.

B. Range Arithmetic

We now consider the basic arithmetic operators over integer ranges. The starting point for our endeavour is to extend the primitive operators to handle infinity in the expected fashion. Figure 1 provides a case analysis for the three main operators, with subtraction omitted for brevity. Note, the fact that $c^{\pm} \mathbin{\hat{\div}} 0$ is undefined is not a specific concern as we will ensure below that this case is not encountered.

The arithmetic operators for addition, subtraction and multiplication on integer ranges can then be defined in the usual manner as follows:

$$\begin{array}{rcl} \mathrm{int}[l_1, u_2] \;+\; \mathrm{int}[l_2, u_2] & \doteq & \mathrm{int}[l_1 \hat{+} l_2, u_1 \hat{+} u_2] \\ \mathrm{int}[l_1, u_2] \;-\; \mathrm{int}[l_2, u_2] & \doteq & \mathrm{int}[l_1 \hat{-} u_2, u_1 \hat{-} l_2] \\ \mathrm{int}[l_1, u_2] \;\times\; \mathrm{int}[l_2, u_2] & \doteq & \mathrm{int}[\min(xs), \max(xs)] \end{array}$$

$$\text{where } xs = \{l_1 \mathbin{\hat{\times}} l_2, l_1 \mathbin{\hat{\times}} u_2, u_1 \mathbin{\hat{\times}} l_2, u_1 \mathbin{\hat{\times}} u_2\}$$

In addition to the above operators, we can also define division on integer ranges for all cases. This is an unusual feature of our system as, normally, division of intervals is limited to those cases where the divisor's range does not include zero. However, as highlighted already, we can exploit the fact that the Whiley compiler has already verified division-by-zero to be impossible. To do this, it uses of an automated theorem prover which can reason about numeric values much more precisely than we can here with our relatively coarse integer ranges.

$\hat{+}$	c_2^+	c_2^-	∞	$-\infty$
c_1^+	$c_1 + c_2$	$c_1 - c_2$	∞	$-\infty$
c_1^-	$c_2 - c_1$	$-(c_1 + c_2)$	∞	$-\infty$
∞	∞	∞	∞	$\pm\infty$
$-\infty$	$-\infty$	$-\infty$	$\pm\infty$	$-\infty$

$\hat{\times}$	c_2^+	c_2^-	∞	$-\infty$
c_1^+	$c_1 \times c_2$	$-(c_1 \times c_2)$	∞	$-\infty$
c_1^-	$-(c_1 \times c_2)$	$c_1 \times c_2$	$-\infty$	∞
∞	∞	$-\infty$	∞	$-\infty$
$-\infty$	$-\infty$	∞	$-\infty$	∞

$\hat{\div}$	c_2^+	c_2^-	∞	$-\infty$
c_1^+	$c_1 \div c_2$	$-(c_1 \div c_2)$	0	0
c_1^-	$-(c_1 \div c_2)$	$c_1 \div c_2$	0	0
∞	∞	$-\infty$	∞	$-\infty$
$-\infty$	$-\infty$	∞	$-\infty$	∞

Fig. 1. Case analysis for extension of arithmetic operators to $\pm\infty$. The subtraction operator is omitted for brevity, as it is similar to addition. Also, $c^{\pm} \div 0$ is undefined and, hence, $c^{\pm} \mathbin{\hat{\div}} 0$ is undefined. Finally, the resulting sign for e.g. $-\infty \mathbin{\hat{+}} \infty$ is determined by position in the constructed integer range (e.g. if in lowerbound position then negative sign, etc)

Division of integer ranges is defined as follows where the underlying operator, $\div$, corresponds to *integer division* which rounds towards zero (i.e. as commonly found in programming languages such as Java and Whiley):

$$\mathrm{int}[l_1, u_1] \;\div\; \mathrm{int}[l_2, u_2] \quad \doteq \quad \mathrm{int}[\min(xs), \max(xs)]$$

$$\text{where} \quad xs = \{l_1 \mathbin{\hat{\div}} l_3, l_1 \mathbin{\hat{\div}} u_3, u_1 \mathbin{\hat{\div}} l_3, u_1 \mathbin{\hat{\div}} u_3\}$$

$$l_3 = \begin{cases} l_2 & \text{if } l_2 \neq 0 \\ 1 & \text{if } l_2 = 0 \end{cases} \qquad u_3 = \begin{cases} u_2 & \text{if } l_2 \neq 0 \\ -1 & \text{if } u_2 = 0 \end{cases}$$

The key here is that in the special case that either the lower or upper bound of the divisor's range is zero, then we can pick the next available value in the range. For the lower bound this is 1, whilst for the upper bound this is –1.

C. Range Lattice

In order to manipulate ranges within our algorithm, we need the ability to union ranges at control-flow meet points and intersect them at control-flow branch points. To do this, we define the least-upper bound ($\sqcup$) and greatest-lower bound ($\sqcap$). For integer ranges, this is straightforward:

$$\begin{array}{rcl} \mathrm{int}[l_1, u_1] \sqcup \mathrm{int}[l_2, u_2] & \doteq & \mathrm{int}[\min(l_1, l_2), \max(u_1, u_2)] \\ \mathrm{int}[l_1, u_1] \sqcap \mathrm{int}[l_2, u_2] & \doteq & \mathrm{int}[\max(l_1, l_2), \min(u_1, u_2)] \end{array}$$

For the purposes of integer ranges, we can think of $\sqcup$ and $\sqcap$ as set union and set intersection. Thus, $\mathrm{int}[0, 255] \sqcup \mathrm{int}[-128, 127]$ gives $\mathrm{int}[-128, 255]$, etc. For list ranges, things are slightly more involved:

$$\begin{array}{rcl} \mathrm{list}\langle T_1\rangle[n_1] \sqcup \mathrm{list}\langle T_2\rangle[n_2] & \doteq & \mathrm{list}\langle T_1 \sqcup T_2\rangle[\max(n_1, n_2)] \\ \mathrm{list}\langle T_1\rangle[n_1] \sqcap \mathrm{list}\langle T_2\rangle[n_2] & \doteq & \mathrm{list}\langle T_1 \sqcap T_2\rangle[\min(n_1, n_2)] \end{array}$$

Here, we recursively use $\sqcup$ (resp. $\sqcap$) to determine the ranges of all elements. For example, $\text{list}\langle\text{int}[0,255]\rangle[64] \sqcup \text{list}\langle\text{int}[-128,127]\rangle[32]$ gives $\text{list}\langle\text{int}[-128,255]\rangle[64]$, etc.

Finally, we also provide unique top ($\top$) and bottom ($\bot$) elements to ensure a complete lattice. Thus, for example, $\text{list}\langle T\rangle[n] \sqcup \text{int}[l,u] = \top$, $\text{int}[l,u] \sqcup \bot = \text{int}[l,u]$, etc.

D. Range Comparators

In addition to the operators discussed thus far, we will also need some for comparing ranges. These are needed to properly account for the effect that conditional branches have. For example, suppose variable `x` has type `u16` and, hence, its base range is $\text{int}[0,65535]$. Then, on the true branch of a conditional **`if`**`(x <= 255)` we can further refine this to $\text{int}[0,255]$, whilst on the false branch it becomes $\text{int}[256,65535]$.

Range comparators can affect the known ranges of all variables involved. For example, suppose variable `x` has range $\text{int}[-128,127]$ and variable `y` has range $\text{int}[0,255]$. Then, on the true branch of the conditional **`if`**`(x==y)` we can refine both ranges by computing their intersection $\text{int}[-128,127] \sqcap \text{int}[0,255] = \text{int}[0,127]$. In this case, however, on the false branch we cannot refine either variable's range.

As will become more apparent in the following section, we can limit our range comparators to those acting on two variables as this follows the form of our intermediate language. Thus, each range comparator accepts two ranges and produces two ranges which are used to update the variables in question. To begin with, we define the following comparators for general range types:

$$T_1 = T_2 \quad \doteq \quad \Big(T_1 \sqcap T_2, T_1 \sqcap T_2\Big)$$

$$T_1 \neq T_2 \quad \doteq \quad \Big(T_1 - T_2, T_2 - T_1\Big)$$

To handle the latter case ($\neq$) we have used the *difference* operator which can be thought of as set difference, and is defined as follows:

$$\text{int}[l_1,u_1] - \text{int}[l_2,u_2] \quad \doteq \quad \begin{cases} \text{int}[l_1+1,u_1], \text{if } l_1 = l_2 = u_2 \\ \text{int}[l_1,u_1-1], \text{if } u_1 = l_2 = u_2 \\ \text{int}[l_1,u_1], \text{otherwise} \end{cases}$$

$$\text{list}\langle T_1\rangle[n_1] - \text{list}\langle T_2\rangle[n_2] \quad \doteq \quad \text{list}\langle T_1\rangle[n_1]$$

For calculating differences of integer types, we can only extract information when the right-hand side is a constant and, furthermore, when it matches either the lower or upper bound of the left-hand side (i.e. as we cannot precisely represent the range without that element). Unfortunately, for calculating differences of list types, there is no additional information that can be extracted. For integer ranges, we also define the following inequality comparators:

$$\text{int}[l_1,u_1] \leq \text{int}[l_2,u_2] \quad \doteq \quad \Big(\text{int}[l_1,\min(u_1,u_2)], \text{int}[\max(l_1,l_2),u_2]\Big)$$

$$\text{int}[l_1,u_1] < \text{int}[l_2,u_2] \quad \doteq \quad \Big(\text{int}[l_1,\min(u_1,u_2-1)], \text{int}[\max(l_1+1,l_2),u_2]\Big)$$

The definitions of these two comparators may seem a little odd, but they are designed to extract the maximum information possible. For example, consider a conditional **`if`**`(x <= y)` where variable `x` has range $\text{int}[0,255]$ and variable `y` has range $\text{int}[-128,127]$. For the true branch, we know that no value for `x` can be greater than the largest value for `y`. Hence, for example, variable `x` cannot hold `255` on the true branch. Likewise, we know that no value for `y` can be below the smallest value for `x`. Hence, for example, variable `y` cannot hold `-128` on the true branch, etc.

E. Widening

Finally, we briefly discuss the notion of *widening* as this is used in the description of our algorithm. Essentially, this refers to the process of determining the largest possible range for a given variable based on its declared type. Thus, widening a variable of declared type `u16` means assigning it the range $\text{int}[0,65535]$. In contrast, widening a variable of declared type **`int`** means assigning it the range $\text{int}[-\infty,\infty]$. Widening extends to list types as well in the obvious manner. For example, a variable of declared type `[u16]` is widened to the range $\text{list}\langle\text{int}[0,65535]\rangle[\infty]$.

Widening is a common concept found in the literature on program analysis [8]. In the context of range analysis, it would generally be taken to represent the process of finding a range for a given variable (typically, a loop induction variable) which ensures the fixed-point computation terminates. This does not necessarily mean the worst-case scenario of $\text{int}[-\infty,\infty]$ is the best option, but it typically is. Compared with this, we have an advantage as data type invariants mean we can widen variables more precisely without needing complicated search heuristics to find appropriate widenings.

IV. Range Analysis

We now examine how our algorithm for range analysis of Whiley programs works. A prototype implementation is also available from http://github.com/Whiley/Whiley2EmbeddedC.

A. Forward Propagation

Our range analysis does not operate directly on Whiley source code. Instead, it operates on the *Whiley Intermediate Language (WyIL)*. This 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):

```
// Define signed bytes
type i8 is (int x)
where -128<=x && x<=127

// Increment with wrap-round
function f(i8 x)->i8:
  //
  if x == 127:
    x = -128
  else:
    x = x + 1
  //
  return x
```

```
function f(i8)->i8:
body:
  const %1 = 127
  ifne %0, %1 goto l2
  const %0 = -128
  goto l3
.l2
  const %1 = 1
  add %0 = %0, %1
.l3
  return %0
```

Here, the function's body is represented as a bytecode block which has access to an unlimited register set (as needed). Registers 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 `add` bytecode adds its operands and assigns to a given target register. Finally, the `return` bytecode returns its operand.

The core of our range analysis is a forward propagation algorithm which determines the possible range for each register at each program point. Figure 2 illustrates the general idea for our example

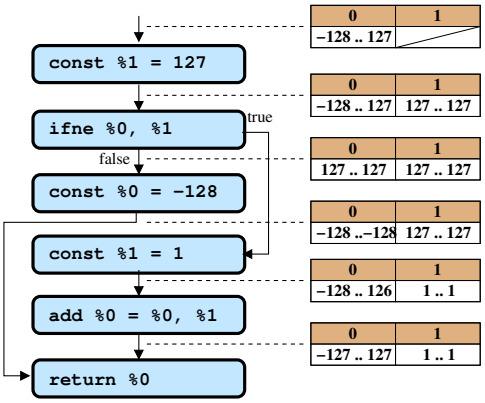

Fig. 2. Illustrating the ranges generated by forward propagation through the body of the function `f()`. The computed ranges for each transition are shown on the right. For example, after the first bytecode is executed register %1 has range $\mathrm{int}[1,1]$. Note, no range is given for register %1 before this bytecode as it was undefined at that point.

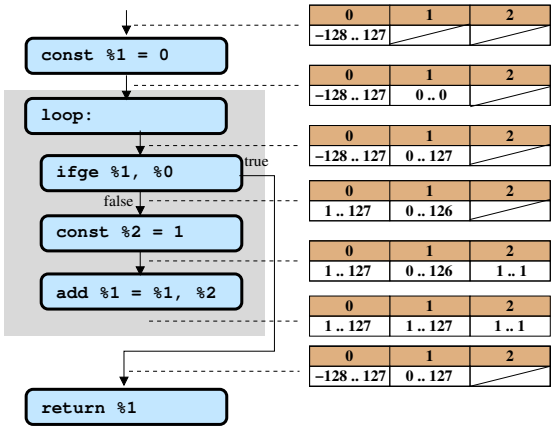

Fig. 3. Illustrating the ranges generated by forward propagation through the body of the function `g()`. The `loop` bytecode is included to clarify the process of widening variables on entry. Specifically, on entry to the loop, the range for register %1 is immediately widened to its declared maximum.

function `f()` from before. In the figure, we can see that the range of register %0 is updated by the conditional bytecode on both the true and false branches. On the false branch, this is done by intersecting ($\sqcap$) the ranges of the two operand registers. On the true branch, this is done by taking the difference of the left operand from the right operand.

Loops. Conditional and unconditional branching instructions are restricted to forward branching only. This means they cannot be used to form loops. Instead, loops are implemented in WyIL using special loop bytecodes. The primary reason for this design decision was the desire for a compact bytecode which supports a wide range of source-level statements but which also simplifies verification. This decision turns out to be useful here, as it eliminates the need to identify loops in the control-flow graph. The following illustrates:

```
// 7bit unsigned integers
type u7 is (int x)
where 0<=x && x<=127

// Count upto n
function g(i8 n) -> u7:
  u7 i = 0
  //
  while i < n:
      i = i + 1
  //
  return i
```

```
function g(i8)->u7:
body:
 const %1 = 0
 loop modifies %1:
   ifge %1,%0 goto l0
   const %2 = 1
   add %1 = %1, %2
.l0
 return %1
```

Here, the `loop` bytecode denotes a loop block containing one or more bytecodes. By itself, the loop bytecode says nothing about how the loop is exited. In this example, a conditional branch is placed at the beginning of the loop body to simulate the **`while`** loop. Equally, the condition could be placed at the end of the loop body to implement a **`do`**-**`while`** loop.

The loop bytecode always includes a *modifies* clause which indicates those registers which may be modified by the loop (in this case, register %1 maybe modified). It is an error for any variable declared outside of the loop body to be modified within the loop without being present in this clause. Note that only loop-carried registers need be included in the modifies clause and, hence, register %2 is omitted even though it is technically modified in the loop. Again, the purpose of the modifies clause was to aid verification. However, it turns out to have a double use here, as we can exploit the modifies clause in determining which variables to widen (more on this shortly).

Previous systems for range analysis often require a fixed-point computation and fall back on an imprecise widening operator to ensure termination in the presence of loops (e.g. [9], [17], [18]). Our algorithm does not require a fixed-point computation and, hence, termination is trivially guaranteed. This is because we can exploit information given in loop and data type invariants to (effectively) generate a precise widening immediately. Figure 3 illustrates the solution generated for function `g()` above. Observe that the range of register %1 is $\mathrm{int}[0,0]$ going into the loop bytecode, but is immediately widened to $\mathrm{int}[0,127]$ at the start of the loop body. This widening exploits the declared type of the variable it represents. Furthermore, all registers which are modified within a loop are widened on entry based on their respective declared type.

Invariants. Type and loop invariants are also encoded into WyIL bytecodes. For example, the type `i8` used in function `f()` is represented as follows:

```
type i8 : int
invariant:
  const %1 = -128
  ifgt %1, %0 goto lab0
  const %1 = 127
  ifgt %0, %1 goto lab0
  return
.lab0
  fail
```

Here, the `invariant` block implements the invariant logic using conditional branches, where register %0 represents an arbitrary value of type `i8`. Execution paths which reach a **`fail`** bytecode represent paths which the Whiley compiler must ensure are unrealisable.

To determine the range for variables of type `i8`, we reuse our forward propagation algorithm. In this case, we are only interested in the possible range of values for register %0 at the **`return`** statement. This is because `invariant` blocks, such as above, are never actually implemented on the target machine — they are merely abstract encodings used for verification.

Finally, loop invariants are handled in much the same way to restrict the ranges of modified registers within a loop. They are applied before processing the loop body after all such variables

are widened. Thus, for example, variables which are widened to $\mathrm{int}[-\infty, \infty]$ (i.e. because they have no declared type invariant) can still be narrowed through the loop invariant.

B. Register Allocation

Having computed ranges for all variables using the forward propagation algorithm, the next question is how to efficiently allocate variables to machine registers. A simple approach is to allocate each variable based on the maximum number of bits it requires at any point. Unfortunately, this approach is far from optimal for two reasons: firstly, it is not space efficient as a variable may be allocated to a larger register than necessary for portions of its life; secondly, it is not time efficient as it can result in unnecessary coercions. To understand the latter, consider the following example:

```
function h(i8 x) -> (u16 r):
  //
  int y
  if x > 0:
     y = 2*x // x: 1..127, y: 1..254
  else:
     y = -x  // x: -128..0, y: 0..128
             // x: -128..0, y: 0..128
  x = x + y  //
             // x: 0..381, y: 0..254
  return x
```

Now, the question is: *what size machine registers do we allocate for variables `x` and `y`?* Let us imagine a target architecture that supports both 8bit and 16bit machine registers and that we allocate registers using the simple approach above. Therefore, variable `x` is allocated to a 16bit register and variable `y` to an 8bit register. The expression `x + y` now presents a problem as our machine instructions (as is common) require operands of the same width. Therefore, we must now coerce the contents of variable `y` into a temporary 16bit register. However, had we allocated variable `y` to a 16bit register in the first place, this coercion would have been unnecessary.

At this juncture we will not discuss the register allocation problem any further, other than to note that our prototype implementation adopts the simple approach outlined above. Therefore, it remains as interesting future work to address this problem in more detail.

V. Discussion

In this section, we discuss several salient aspects of our system which warrant further attention.

A. Motivation

An interesting question is why a forward analysis is needed at all for our system. That is, *why is it not enough to simply extract ranges from declared types and allocate machine registers based on that?* There are various different ways of looking at this question. Certainly, in order to extract ranges from type invariants we require a forward analysis as invariants are themselves encoded using WyIL bytecodes. However, whilst this motivates the need for such an algorithm, it does not explain the need to run it on anything other than type invariants. One of the problems we face is that WyIL registers don't necessarily correspond to program variables. For example, they can be synthetic and introduced as part of the compilation process. Such registers will not have declared type invariants (i.e. as the compiler would not know what invariant to choose). Therefore, we need to propagate information from registers representing declared variables to those which do not. Finally, we also wish to support as many Whiley programs as possible and, as discussed earlier, it is perfectly acceptable in Whiley to bound variables using only pre-/post-conditions and loop invariants.

B. Symbolic Analysis

Our algorithm does not employ any form of symbolic analysis and this can limit its precision in places. Consider the following:

```
function f(int x, int y) -> int
requires 0 <= x && x <= y && y <= 255:
   ...
```

Intuitively, it is clear that both variables `x` and `y` can be safely allocated to 8bit machine registers. However, the bytecode for this precondition is as follows:

```
requires:
  const %2 = 0 : int
  ifgt %2, %0 goto lab_0
  // x: 0..inf, y: -inf..inf
  ifgt %0, %1 goto lab_0
  // x: 0..inf, y: 0..inf
  const %3 = 255 : int
  ifgt %1, %3 goto lab_0
  // x: 0..inf, y: 0..255
  return
.lab_0
  fail
```

The problem here is that the propagation algorithm has not correctly identified that variable `x` has range $\mathrm{int}[0, 255]$. This has happened because of the specific ordering of the conditional branches in the bytecode. To overcome this problem, one needs to use a symbolic analysis which retains the relationship between variables in order to ensure effects on one are propagated to another (i.e. in the above, since `x <= y`, then variable `x` is updated when `y <= 255` is encountered).

We note that this limitation with our analysis is offset by the fact that most precision comes from data type invariants which, by construction, are only defined in terms of a single variable. As such, when determining bounds for them, the above issue does not arise.

C. Theorem Prover

For the purposes of verification, the Whiley compiler employs an automated theorem prover which is capable of reasoning about numerical values far more precisely than can be done with ranges alone. This raises the obvious question: *why not use this to help to determine bitwidth?* Certainly, this is something we want to explore in the future. However, such tools do not lend themselves naturally to this problem. Specifically, automated theorem provers are good at answering questions of the form: *is something true?* For example, one can easily answer the question: *is variable x greater than 0?* But, questions of the form "*what is variable x greater than?*" are much less amenable for them. In contrast, our integer range analysis handles exactly questions of this form quite well.

In order to utilise an automated theorem prover, one could develop specific strategies. For example, supposing a machine with 8bit and 16bit registers. Then, one can ask direct questions of the form "*is variable x below 256?*" to help determine which register to use. However, if there are many register sizes or registers have arbitrary width (e.g. on an FPGA) this may be prohibitively expensive.

VI. Related Work

A. Bounds Checking

The problem of determining whether an array or pointer access is within bounds has been long studied in the literature. Typically, the goal is eliminate the need for runtime bounds checks where possible and/or to identify potential security vulnerabilities. Furthermore, there is usually an assumption that programs are analysed as-is (i.e. without annotations) and, hence, many works employ costly interprocedural analyses. For languages like C and Java, such analyses will also often include some mechanism for tracking pointers and aliases.

The work of Yong and Horwitz is a good example which specifically targets the elimination of runtime bounds checks and identification of potential vulnerabilities [9]. Their system performs static analysis of C programs in the presence of arbitrary pointer arithmetic and type casting (so-called *dusty deck* programs). Of particular relevance here is their use of integer ranges to bound integer variables, although their presentation does not consider multiplication or division of ranges. As with many algorithms of this nature, termination is an issue as the lattice of ranges has infinite ascending chains and, hence, a widening operator is required to ensure termination.

The earlier work of Wilson and Lam provides a similar example to that of Yong and Horwitz [10]. The bulk of their algorithm is concerned with interprocedural pointer analysis, and they make many choices which favour precision over performance. In particular, their algorithm is context- and flow-sensitive and makes use of function summaries to reduce the cost of repeated calls to the same function. Unfortunately, these choices mean that their algorithm suffers from an exponential blow up which renders it impractical on all but relatively small programs. However, of interest here is their representation of integer ranges which, unlike our approach, also includes a notion of *stride*. For example, all even integers between the lower and upper bound can be precisely characterised.

The above examples follow the ideas of classical dataflow analysis whereby ranges are propagated throughout the control-flow graph of a function and the computation iterates until a fixed-point is reached. A common and widely successful alternative is to generate a constraint graph from the program and then solve that separately. In the context of range analysis, such constraints are normally linear inequalities in some form or another. The ABCD system of Bodik *et al.* provides an excellent example of this approach [11]. Unlike most of the other works discussed here, they are concerned with analysing Java programs within the context of a virtual machine. They generate constraints known as *difference* or *potential* constraints which have the specific form $x - y \geq c$, where x, y are variables and c is a constant. The key is that such constraints lend themselves naturally to a weighted, directed-graph representation which can be solved using algorithms akin to transitive closure. Of course, such constraints cannot encode all possible expressions encountered in practice and, in such cases, coarse approximations are used. A key challenge is that positive-weight cycles must be identified and broken to ensure termination (this is roughly similar to the notion of widening used in dataflow analysis). The experimental results generated with this algorithm demonstrated it to remove 45% of all bounds checks over five benchmarks from the SPECjvm98 benchmark suite.

A similar approach to that of ABCD was also used by Wagner *et al.* to identify buffer overrun vulnerabilities in C programs [12]. They focused specifically on string manipulations, rather than general arrays and/or pointer arithmetic. Despite this, their system identified several previously unknown vulnerabilities in the widely-used `sendmail` application, despite this having been previously audited by hand. In their system, strings are modelled as pairs consisting of the number of allocated bytes and the number of bytes used. This is in some ways similar to the approach we have taken to modelling lists in Whiley. Again, to ensure termination of their algorithm, cycles in the constraint graph must be identified and broken.

Finally, another work following in this line is that of Rugina and Rinard who are concerned with bounds analysis for pointers and arrays in C [13]. A key difference from the earlier works of Bodik *et al.* and Wagner *et al.* is that their algorithm is interprocedural and precisely tracks pointer aliasing information. Furthermore, they generate linear programs which are more precise and can be offloaded to a special-purpose linear constraint solver. As such termination is not a concern as the linear constraint solver handles this.

B. Bitwidth Analysis

Of direct relevance to this paper is the literature on bitwidth analysis. As with this paper, the goal here is to determine the number of bits required to represent program variables. This is useful in the context of compiler optimisations, but also when compiling to silicone or Field-Programmable Gate Arrays (FPGAs).

A good example here is the work of Budiu *et al* who claim that "*on average 14% of the computed bytes in programs from SpecINT95 and Mediabench are useless*" [19]. Their approach focuses on identifying unused and constant bits in dusty-deck C programs to improve overall efficiency, particularly on architectures that support non-standard or arbitrary register widths (e.g. FPGAs). Their algorithm treats integers simply as bit strings, which are sequences of *known*, *unknown* or *don't care* bits. Examples of the latter arise when bits are effectively discarded, such as when when variables of larger width are cast to those of smaller width, or when masks are applied through bitwise operations, etc. Their general approach appears to be in the style of classical dataflow analysis, although there is little attention given to the issue of handling loops.

In a similar vein, Stephenson *et al.* are interested in optimising bitwidths for the purpose of compiling to silicon or FPGAs [15]. They perform a dataflow analysis using ranges which are comparable to that presented here. Value ranges are propagated in both a forwards and backwards direction, where the latter helps to further narrow ranges for discoveries made downstream. For example, consider a variable which is used to index into an array. If the range of the variable is greater than the known bounds of the array, one can clip the variable's range to match the array bounds (i.e. by assuming that no actual error exists). In such case, we want to propagate this discovery back through the control-flow graph to further prune any variables flowing into this. Loops are handled through pattern matching and, when this fails, falling back to a standard iterative approach with widenings.

Another interesting example here is the work of Ergin *et al* which is concerned with efficient packing of variables into registers [20]. More specifically, their aim is to pack multiple variables into a given register where possible to alleviate pressure on register allocation. Their work is also quite different from other works discussed here, as it is not a form of static analysis. Rather, their approach is intended to be implemented within a microprocessor itself and uses a concept of bitwidth prediction, which is analogous to branch prediction. Their claim is that 40% of all values can be represented in 16bits, with a further 45% requiring only 32bits. Thus, in modern 64bit systems there is ample opportunity to improve register utilisation.

Finally, it is interesting to note that the majority of previous work on bitwidth analysis focuses on integer variables. In constrast, the work of Gaffar *et al* focus on the bitwidth analysis of floating pointer variables for compilation onto FPGAs [21]. Their approach is also unusual in that it focuses on the sensitivity of output variables, and correlates higher sensitivity with a need for increased bitwidth.

C. Abstract Domains

In this paper, we focused on representing the set of possible values for a given variable using a simple interval. However, as we've already seen with the work of Wilson and Lam, other representations are possible which can characterise sets more precisely. The most obvious extension to the classical interval is to include some notion of stride, and this is commonly referred to as the *modulo interval*. Here, the interval is represented as a tuple (l, u, m, n) which describe the set $\{km + n \mid l \leq k \leq u\}$. Nakanishi *et al.* provide a formal examination of modulo intervals and their mathematical properties [16].

The very recent work of Gange *et al.* consider the domain of intervals which correctly handle wrap-around as caused by overflow and underflow [17]. Like others discussed earlier, their approach treats all integers uniformly as bit strings. They also focus on supporting the abstract machine instructions provided by LLVM. In particular, this includes support for both signed and unsigned division, and the corresponding remainder operators. In addition, they present a complete range analysis in the style of a classical dataflow analysis which employs widening to ensure termination.

Finally, we should note that there are other more complex numerical domains which have been considered in the program analysis literature. For example, the octagon domain generalises the notion of difference inequality constraints discussed earlier [22]. Likewise, the widely used polyhedra domain utilises much more expressive linear inequalities but, compared with simple intervals, is far more computationally expensive [23].

D. Other

Finally, we pick out some other works of note with respect to range analysis. Healy *et al* are concerned with bounding loop iterations to calculate Worst-Case Execution Time (WCET) [24]. They employ integer ranges in an unusual fashion to bound the number of iterations of each node in the control-flow graph. In contrast, Patterson employs integer range analysis for static branch prediction [18]. His approach is unusual as it computes weighted probability ranges, rather than exact ranges as we do. Chin and Khoo focus on higher-order functional programs and are concerned with ensuring termination of recursive functions [25]. They adopt a technique based on linear inequalities to bound the sizes of return values which can, for example, be used to establish they decrease in size. Finally, Bultan *et al* employ model checking to symbolically evaluate safety and liveness properties for programs involving unbounded integer variables [26]. Their encoding also exploits linear inequalities to succinctly describe integer sets and employs a fixpoint algorithm whose convergence is guaranteed using approximation techniques (i.e. widening).

VII. Conclusion

In this paper we have presented a range analysis for variables of integer and list type in the Whiley programming language. Variables of integer type are modelled with intervals, whilst those of list type are modelled by their maximum length along with a range summarising their elements. Our formalisation of integer ranges exploits the knowledge that Whiley programs have been verified as free of divide-by-zero errors. Our range analysis consists of a forward propagation algorithm which, unlike many comparable approaches, does not require the use of a fixed-point computation that could iterate indefinitely. Instead, knowledge of declared variable types and loop invariants is exploited to ensure only a single pass through the control-flow graph is required and, hence, termination is guaranteed. We have developed a prototype implementation which uses a relatively simple approach to register allocation based on the computed ranges. In the future, we hope to explore the register allocation problem further.

References

[1] "The Whiley Programming Language, http://whiley.org."

[2] D. J. Pearce and J. Noble, "Implementing a language with flow-sensitive and structural typing on the JVM," *ENTCS*, vol. 279, no. 1, pp. 47–59, 2011.

[3] D. J. Pearce and L. Groves, "Whiley: a platform for research in software verification," in *Proc. SLE*, 2013, pp. 238–248.

[4] ——, "Reflections on verifying software with Whiley," in *Proc. FTSCS*, 2013, pp. 142–159.

[5] B. Jacobs, J. Smans, and F. Piessens, "The verifast program verifier: A tutorial," Tech. Rep., 2014.

[6] C. Flanagan, K. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata, "Extended static checking for Java," in *Proc. PLDI*, 2002, pp. 234–245.

[7] M. Barnett, M. Fähndrich, K. R. M. Leino, P. Müller, W. Schulte, and H. Venter, "Specification and verification: the Spec# experience," *CACM*, vol. 54, no. 6, pp. 81–91, 2011.

[8] F. Nielson, H. R. Nielson, and C. L. Hankin, *Principles of Program Analysis*. Springer-Verlag, 1999.

[9] S. H. Yong and S. Horwitz, "Pointer-range analysis," in *2004*, ser. LNCS, vol. 3148. Springer-Verlag, 2004, pp. 133–148.

[10] R. P. Wilson and M. S. Lam, "Efficient context-sensitive pointer analysis for C programs," in *Proc. PLDI*, 1995, pp. 1–12.

[11] R. Bodik, R. Gupta, and V. Sarkar, "ABCD: eliminating array bounds checks on demand," in *Proc. PLDI*, 2000, pp. 321–333.

[12] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken, "A first step towards automated detection of buffer overrun vulnerabilities," in *NDSS*, 2000, pp. 3–17.

[13] R. Rugina and M. C. Rinard, "Symbolic bounds analysis of pointers, array indices, and accessed memory regions," *ACM TOPLAS*, vol. 27, no. 2, pp. 185–235, 2005.

[14] D. J. Pearce, "Getting started with Whiley," Last updated, 2014.

[15] M. Stephenson, J. Babb, and S. Amarasinghe, "Bitwidth analysis with application to silicon compilation," in *Proc. PLDI*, 2000, pp. 108–120.

[16] T. Nakanishi, K. Joe, C. D. Polychronopoulos, and A. Fukuda, "The modulo interval: A simple and practical representation for program analysis," in *Proc. Pact*. IEEE, 1999, pp. 91–96.

[17] G. Gange, J. A. Navas, P. Schachte, H. Søndergaard, and P. J. Stuckey, "Interval analysis and machine arithmetic: Why signedness ignorance is bliss," *ACM TOPLAS*, vol. 37, no. 1, pp. 1:1–1:35, 2015.

[18] J. R. C. Patterson, "Accurate static branch prediction by value range propagation," in *Proc. PLDI*, 1995, pp. 67–78.

[19] M. Budiu, M. Sakr, K. Walker, and S. C. Goldstein, "Bitvalue inference: Detecting and exploiting narrow bitwidth computations," in *Proc. Euro-Par*. Springer-Verlag, 2000, pp. 969–979.

[20] O. Ergin, D. Balkan, K. Ghose, and D. V. Ponomarev, "Register packing: Exploiting narrow-width operands for reducing register file pressure," in *MICRO*. IEEE, 2004, pp. 304–315.

[21] A. A. Gaffar, O. Mencer, W. Luk, P. Y. K. Cheung, and N. Shirazi, "Floating-point bitwidth analysis via automatic differentiation," in *FPT*. IEEE, 2002, pp. 158–165.

[22] A. Miné, "The octagon abstract domain," *Higher-Order and Symbolic Computation*, vol. 19, no. 1, pp. 31–100, 2006.

[23] P. Cousot and N. Halbwachs, "Discovery of linear restraints among variables of a program," in *Proc. POPL*, 1978, pp. 84–97.

[24] C. A. Healy, M. Sjödin, V. Rustagi, D. B. Whalley, and R. van Engelen, "Supporting timing analysis by automatic bounding of loop iterations," *Real-Time Systems*, vol. 18, no. 2/3, pp. 129–156, 2000.

[25] W.-N. Chin and S.-C. Khoo, "Calculating sized types," in *Proc. PEPM*. ACM Press, 2000, pp. 62–72.

[26] T. Bultan, R. Gerber, and W. Pugh, "Model-checking concurrent systems with unbounded integer variables: symbolic representations, approximations, and experimental results," *ACM TOPLAS*, vol. 21, no. 4, pp. 747–789, 1999.