Integer Range Analysis for Whiley on Embedded Systems

David J. Pearce

School of Engineering and Computer Science
Victoria University of Wellington

http://whiley.org

Verifying Compilers

- Tony Hoare proposed the development of a verifying compiler as **grand challenge** for Computer Science (2003)

  > *"A verifying compiler uses automated mathematical and logical reasoning methods to check the correctness of the programs that it compiles"*

- Some **impressive** attempts at this:
  - SPARK/Ada (1983)
  - ESC/Modula3 (1998)
  - ESC/Java (2002)
  - Why / Krakatoa (2002)
  - Spec# (2004)
  - Dafny (2011)

- But, still a **long way** from realising Hoare's dream ...

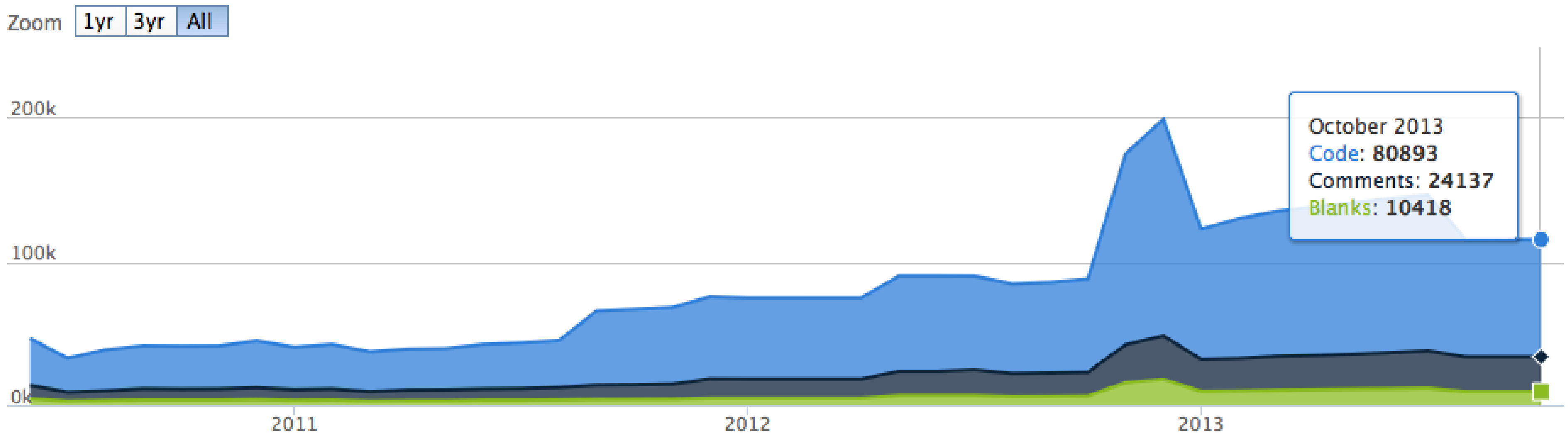
Whiley

Overview: What is Whiley?

```
function max(int x, int y) => (int z)
// result must be one of the arguments
ensures x == z || y == z
// result must be greater-or-equal than arguments
ensures x <= z && y <= z:
    ...
```

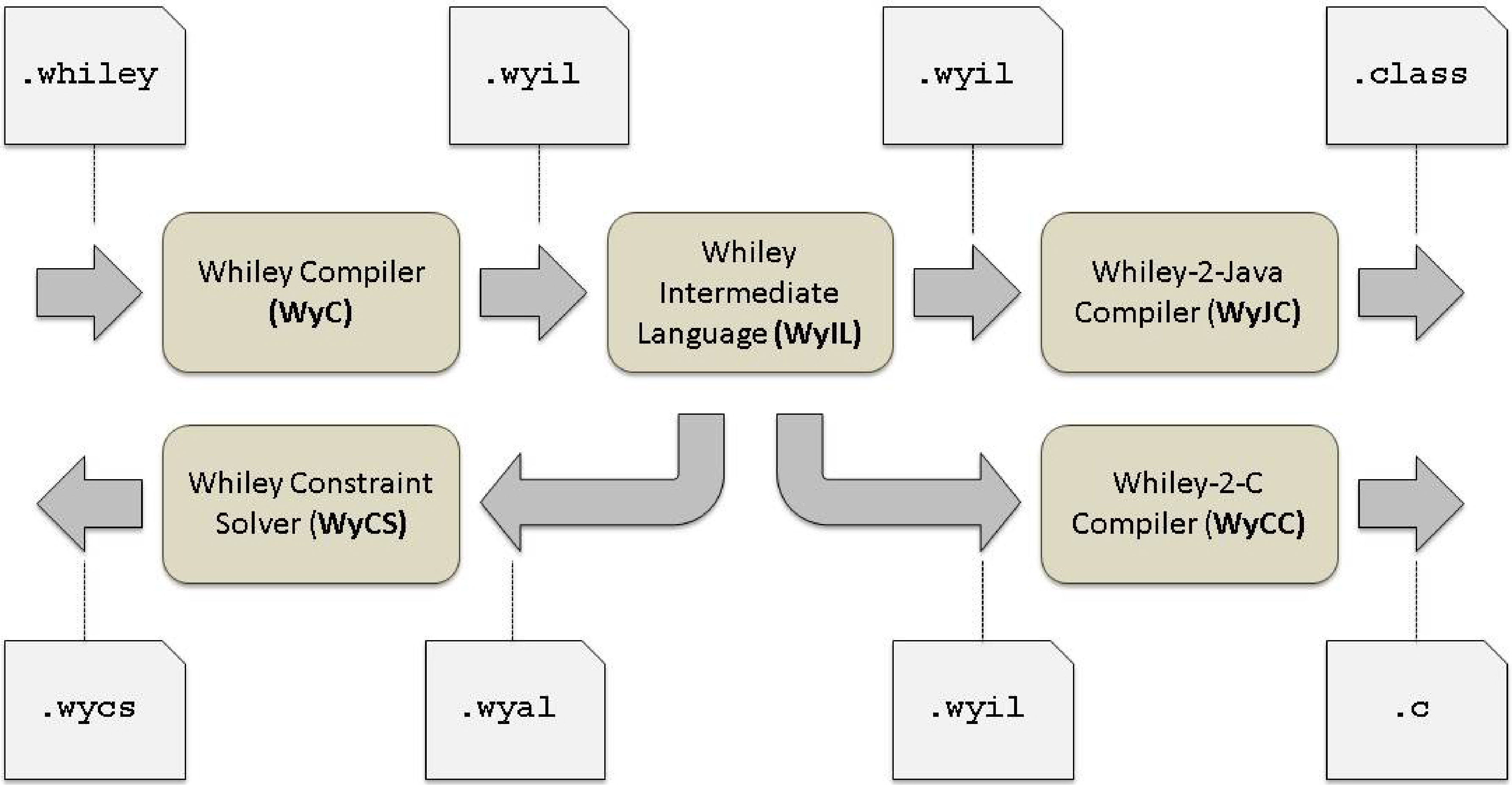
- A language designed specifically to simplify **verifying software**
- Several trade offs e.g. **performance for verifiability**
  - *Unbounded Arithmetic, value semantics, etc*
- **Goal**: to statically verify functions meet their specifications

History of Whiley



- 2009 — **Initial** version of Whiley released (GPL Licence)
- 2010 — **GitHub** repository and **`http://whiley.org`** go live
- 2010 — **Version 0.3.0** released (BSD Licence)
- 2013 — **Latest version** 0.3.20 (approx 81KLOC)
- 2014 — **Version** 0.4.0 released?

Architecture of the Whiley Compiler



Verification

Verification: Example 1

- Consider following example:

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

- Above code is valid and **will verify**

- Verifying compiler **reasons precisely** about information flow

Verification: Example 2

```
function contains([int] xs, int x) -> (bool r)
// if return is true, then some i where xs[i] == x
ensures r ==> some { i in 0 .. |xs| | xs[i] == x }
// if return is false, then no i where xs[i] == x
ensures !r ==> no { i in 0 .. |xs| | xs[i] == x }:
    //
    int i = 0
    //
    while i < |xs| where i >= 0:
        if xs[i] == x:
            return true
        i = i + 1
    //
    return false
```

Embedded Systems

Embedded Systems: Example

- Consider the following example:

```
function append(int item, [int] items) -> [int]:
    return [item] ++ items
```

- What are the **problems** for embedded systems?
  - Arithmetic in Whiley is *unbounded*
  - Lists are *resizable* and *passed-by-value*

Embedded Systems: Example Revised

- Consider a revised version of our example:

```
type u8 is (int x) where 0 <= x && x <= 255

function append(u8 item, [u8] items) -> ([int] r)
requires |items| < 65535
ensures |r| == |items| + 1:
    //
    return [item] ++ items
```

- This is more suitable for an embedded system
- Memory usage can be *constrained*
- In principle, no need for **dynamic memory allocation** either

Embedded Systems: Integer Range Analysis

- Invariants in Whiley can be **arbitrarily complex**:

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

// 8bit integers with a "hole"
type ih8 is (i8 x) where x < -1 || 1 > 0

type p8 is { bool f, nat y }
where (f ==> y < 32) || (!f ==> y < 128)
```

- **Integer range analysis** determines lower and *upper* bound for each variable and upper bounds on **list length**

Embedded Systems: Integer Ranges

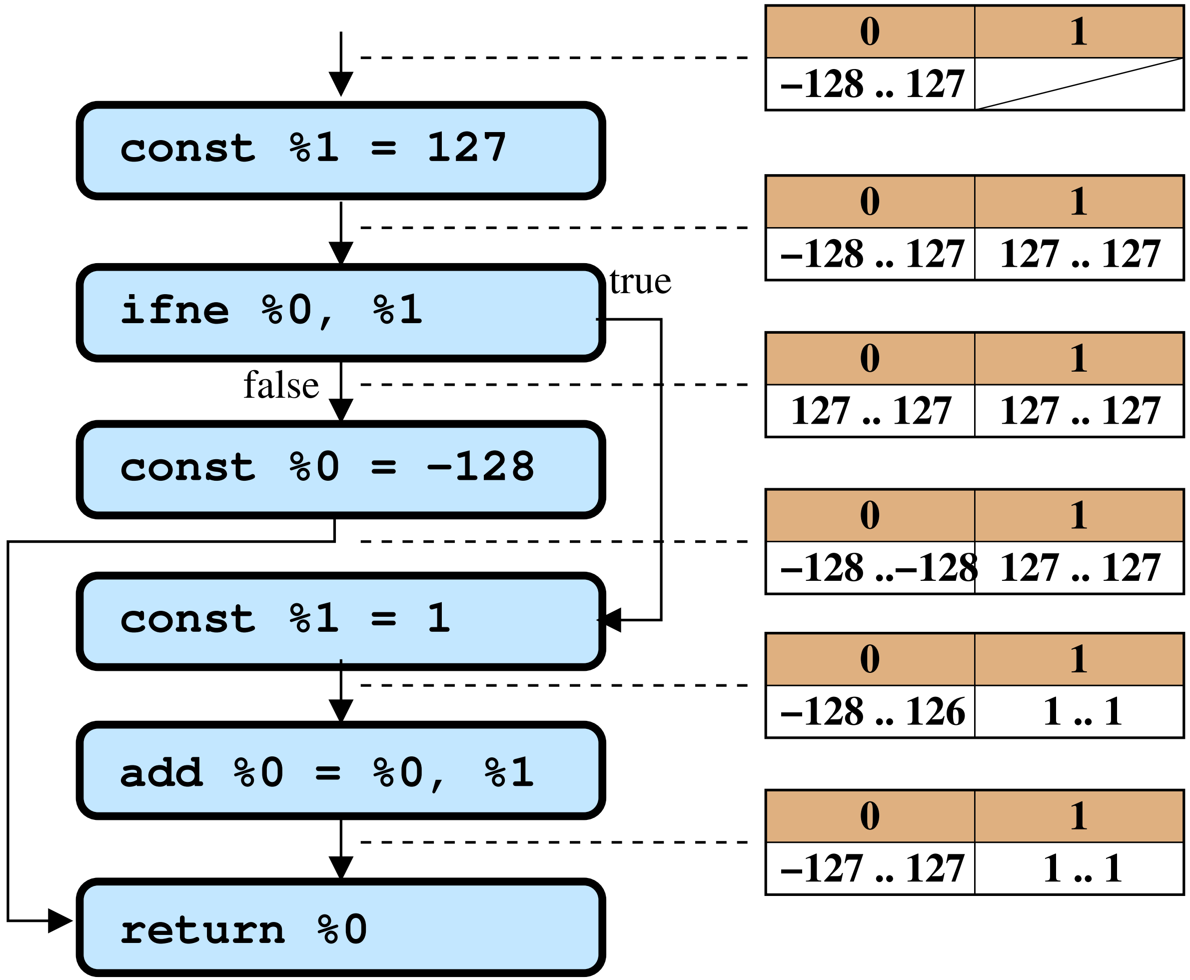
Integer Range

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

- **Arithmetic Operations**:
  - $\text{int}[-\infty, 2] + \text{int}[0, 2] \longrightarrow \text{int}[-\infty, 4]$
  - $\text{int}[-2, 3] * \text{int}[-1, 2] \longrightarrow \text{int}[-4, 6]$
  - $\text{int}[1, 5]/\text{int}[0, 2] \longrightarrow \text{int}[1, 3]$

- **Binary Comparisons**:
  - $\text{int}[0, 2] == \text{int}[-1, 3] \longrightarrow \text{int}[0, 2], \text{int}[0, 2]$ (true branch)
  - $\text{int}[0, 255] <= \text{int}[8, 31] \longrightarrow \text{int}[0, 31], \text{int}[8, 31]$ (true branch)

Embedded Systems: Example



- **Forwards propagation** algorithm in style of dataflow analysis

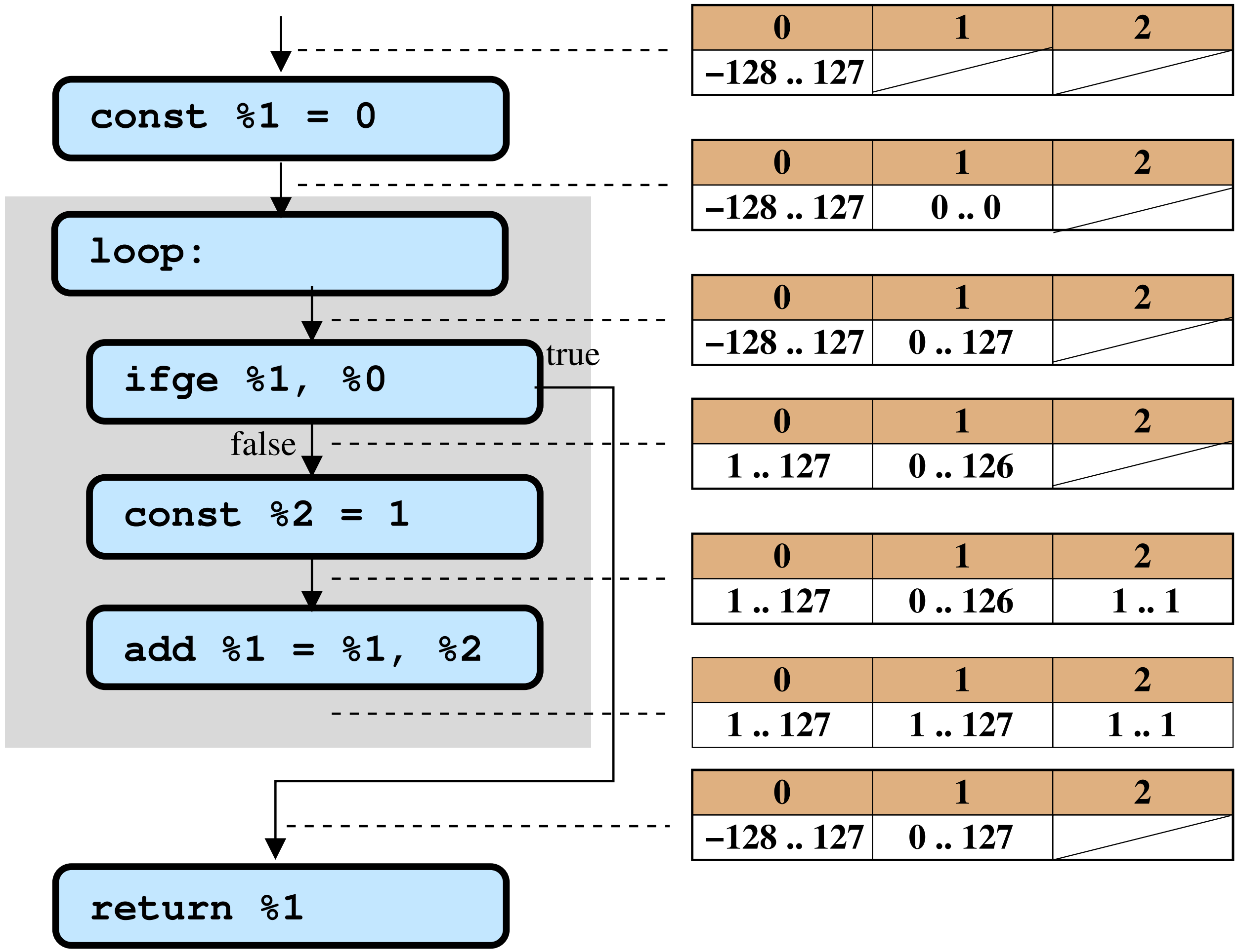
Embedded Systems: Loops

- Consider the following example:

```
function f(u8 n) -> (u8 r):
    int i = 0
    while i < n where i >= 0:
        i = i + 1
    //
    return i
```

- Traditional dataflow analysis handles loops with **fix-point iteration**
- Fix-point iteration **may not terminate** for integer range analysis
- This is **not necessary** here because of type and loop invariants

Embedded Systems: Loop Example



- **Forwards propagation** algorithm in style of dataflow analysis

Embedded Systems: Register Allocation Problem

```
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
```

- Variable `y` can be allocated to 8bit register ... **but should it**?

http://whiley.org

References

- **Reflections on Verifying Software with Whiley**. David J. Pearce and Lindsay Groves. In Proc of FTSCS, (to appear), 2013.
- **Whiley: a Platform for Research in Software Verification**. David J. Pearce and Lindsay Groves. In Proc. SLE, pages 238–248, 2013.
- **A Calculus for Constraint-Based Flow Typing**. David J. Pearce. In Proc. FTfJP, Article 7, 2013.
- **Sound and Complete Flow Typing with Unions, Intersections and Negations**, David J. Pearce. In Proc. VMCAI, pages 335–354, 2013.
- **Implementing a Language with Flow-Sensitive and Structural Typing on the JVM**. David J. Pearce and James Noble. In Proc. BYTECODE, 2011.

Verification: Binary Tree Example

```
type Tree is null | Node

type Node is {
  int data,
  Tree rhs,
  Tree lhs
} where (lhs != null ==> lhs.data < data) &&
        (rhs != null ==> rhs.data > data)
```