Efficient Implementation of A Verification-Friendly Programming Language

Min-Hsien (Sam) Weng

Supervised by
Associate Professor Bernhard Pfahringer
Associate Professor Mark Utting

Reliable Software

- Software in modern life is anywhere and anytime. So are bugs!!!
- Two approaches to improve software quality
    - Testing
        - After production testing, the program still has 5-10 bugs per 1000 line-of-code. [Watts S. Humphrey]
        - Software complexity increases the numbers of bugs.
    - Software Verification / Static Program Analysis

Whiley

- Whiley is a **new** programming language with extended static checking to
  - Detect errors (12/**0?, a[100], null dereference**) at compile-time
  - Produce a program with as few errors as possible
- Whiley has the advantages of **hybrid** imperative and functional programming language:
  - Value Semantics
  - Side-effect Free Function / Referential Transparency
- We choose Whiley as the front-end of this project.

Problems about Whiley

- **Arbitrary-sized** Whiley integers/data structures avoid integer overflows but result in poor performance.
  - Bound analysis finds the lower and upper bounds for each program variable
  - Bound analysis determines “where” and “what” fixed-sized integers and data structures are used.
- **Extra value copying** problem arises from the use of immutable values but increases memory overhead (lowers the efficiency)
  - Reference counting can reduce the copying at runtime.
  - Pointer-to-alias analysis or unique types of Clean can reduce value copying statically.

Research Questions

Can a verification-friendly Programming Language be implemented efficiently?

a. Can abstract interpretation be used to infer static bounds (**integer ranges**, data structure sizes and pointer analysis to avoid copying data) for Whiley programs?

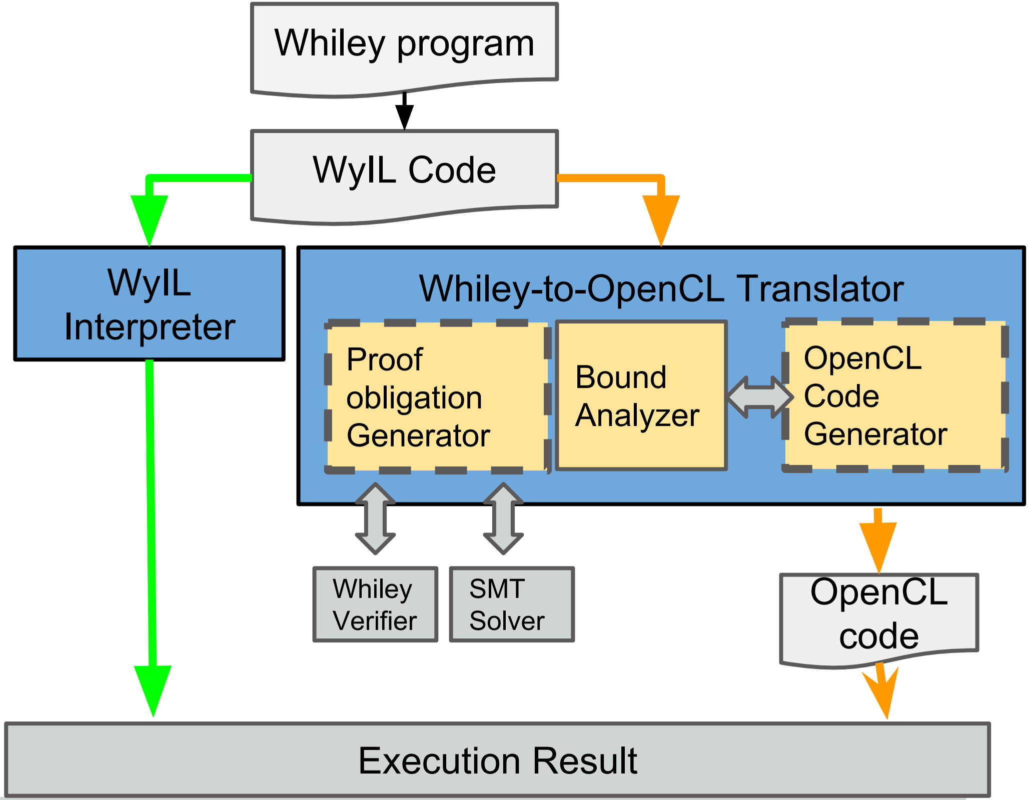
b. Can we automatically identify which parts of programs can be parallelized?

Whiley-to-OpenCL Backend

- WyIL Interpreter
- Whiley-to-OpenCL Translator
  - Proof Obligation Generator
  - Bound Analyzer
  - OpenCL Code Generator

Note blue solid boxes are being developed and yellow dashed boxes will soon be implemented in this project.

.



Whiley-to-OpenCL Translator

- OpenCL/C code generator converts WyIL code into efficient OpenCL code
    a. Use bound analyzer to find fixed-size integer types/data structures and to reduce the number of data copying.
    b. If the bound analyzer fails,
        i. proof obligation generator produces the proof obligations (validated by Whiley checker and SMT solver (Z3)), or
        ii. gives the warning/error messages to programmers for assistants, e.g. stronger assertion and invariants.
- The goal of translator is to implement a large subset of Whiley in C/OpenCL, with parallelism where possible/useful.

Step #1
Compiling the Whiley Program into WyIL Code

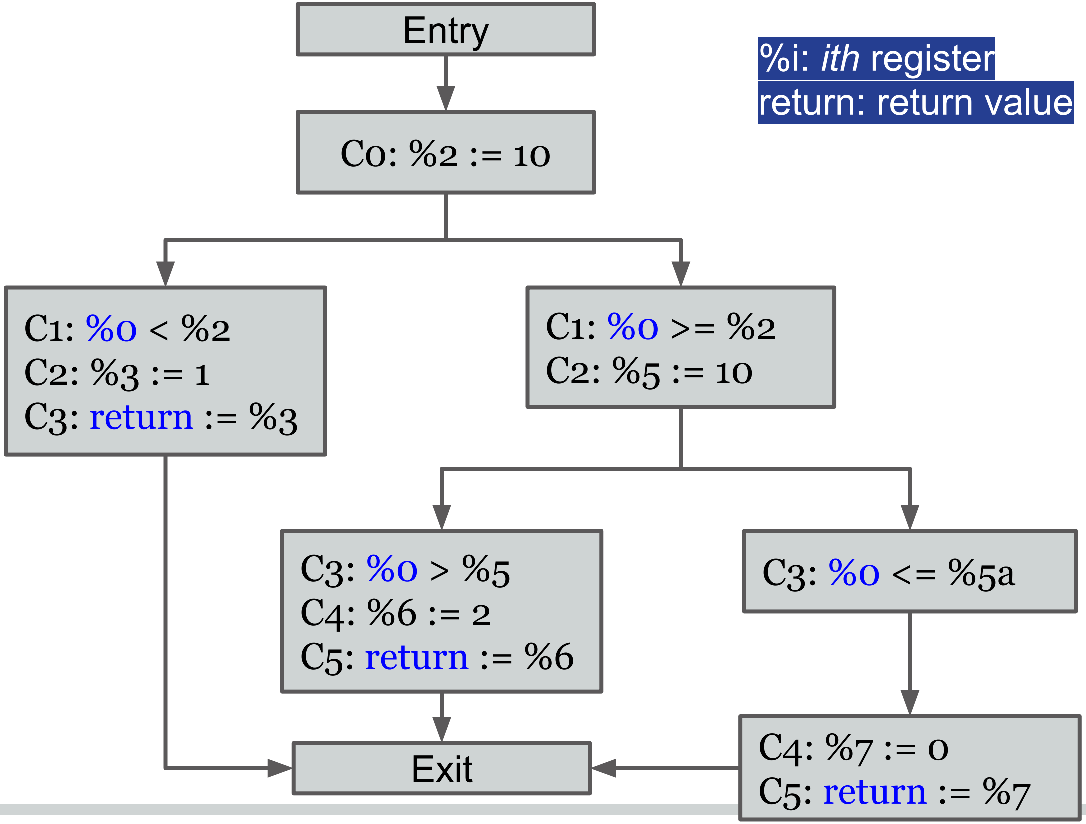
```
function f(int x) => int:
    if x < 10:
        return 1
    else:
        if x > 10:
            return 2
    return 0
```

```
f.0 [    const %2 = 10 : int]
f.1 [    ifge %0, %2 goto blklab0 : int]
f.2 [    const %3 = 1 : int]
f.3 [    return %3 : int]
f.4 [.blklab0]
f.5 [    const %5 = 10 : int]
f.6 [    ifle %0, %5 goto blklab2 : int]
f.7 [    const %6 = 2 : int]
f.8 [    return %6 : int]
f.9 [.blklab2]
f.10 [.blklab1]
f.11 [    const %7 = 0 : int]
f.12 [    return %7 : int]
```

%i : *ith* register

Step #2
Extracting Constraints and Building the Control Flow Graph

```
f.0 [    const %2 = 10 : int]
f.1 [    ifge %0, %2 goto blklab0 : int]
f.2 [    const %3 = 1 : int]
f.3 [    return %3 : int]
f.4 [.blklab0]
f.5 [    const %5 = 10 : int]
f.6 [    ifle %0, %5 goto blklab2 : int]
f.7 [    const %6 = 2 : int]
f.8 [    return %6 : int]
f.9 [.blklab2]
f.10 [.blklab1]
f.11 [    const %7 = 0 : int]
f.12 [    return %7 : int]
```



%i: *ith* register
return: return value

Step #3
Inferring the bounds

B0: { D(%0)=[-inf..inf] }

B1: B0 ∪ {D(%2)=[10..10] }

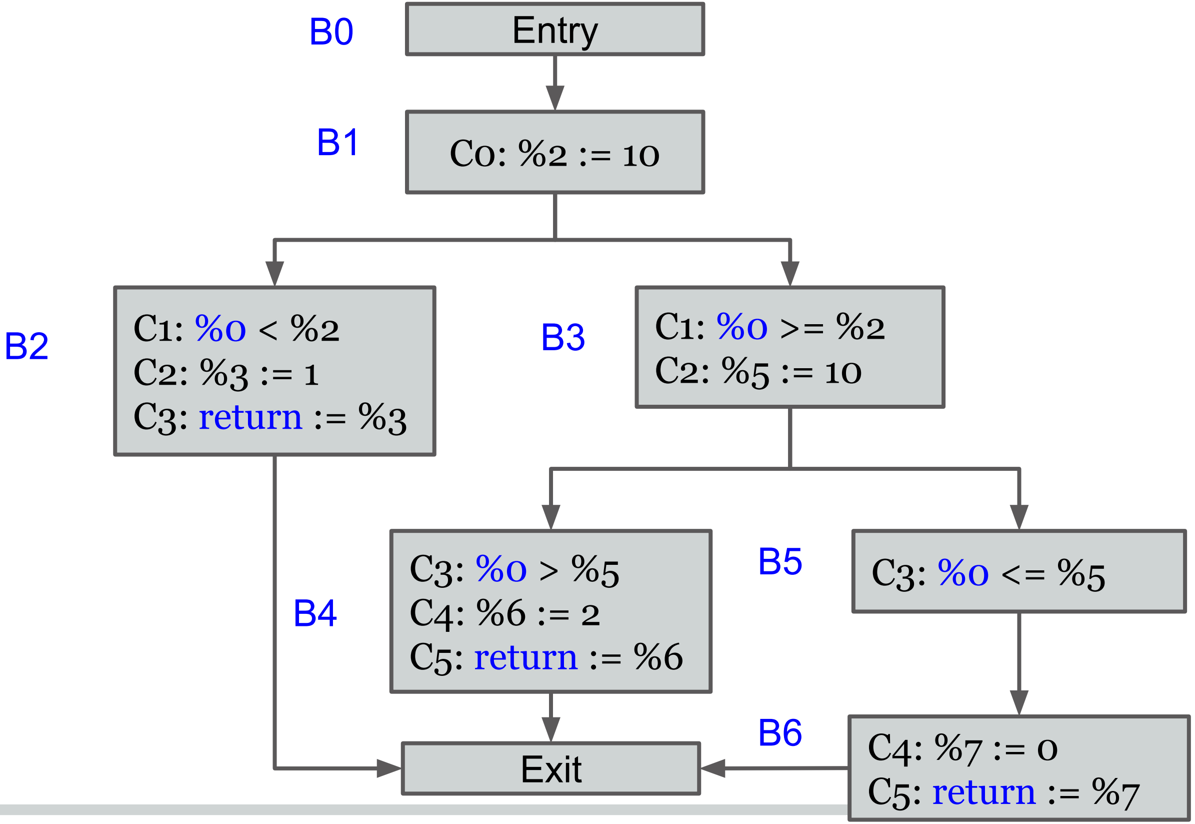
B2: B1 ∪ { D(%0)=[-inf..9]
D(%3)=[1..1]
D(return)=[1..1]}

B3: B1 ∪ { D(%0)=[10..inf]
D(%5)=[10..10]}

B4: B3 ∪ {D(%0)=[11..inf]
D(%6)=[2..2]
D(return)=[2..2] }

B5: B3 ∪ {D(%0)=[-inf..10]}

B6: B5 ∪ { D(%7)=[0..0]
D(return)=[0..0]}



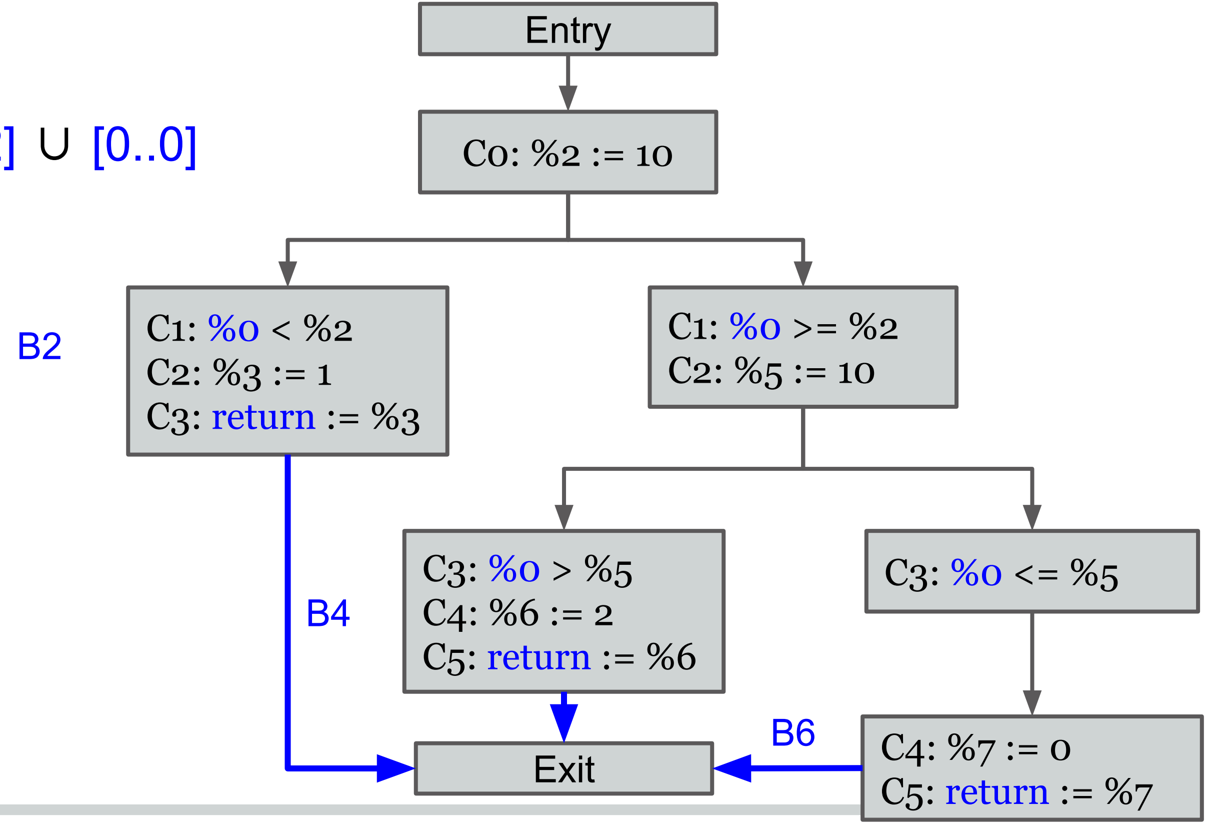
Step #4
Inferring Bounds for Function Result

B7 : B2 ∪ B4 ∪ B6=

{ D(return) = [1..1] ∪ [2..2] ∪ [0..0]

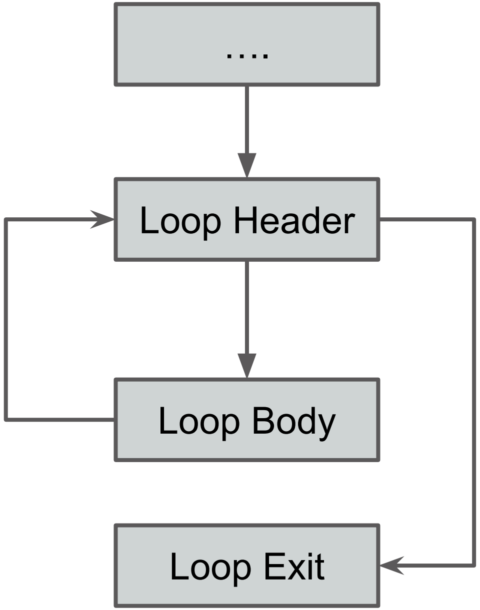
= [0..2]

}



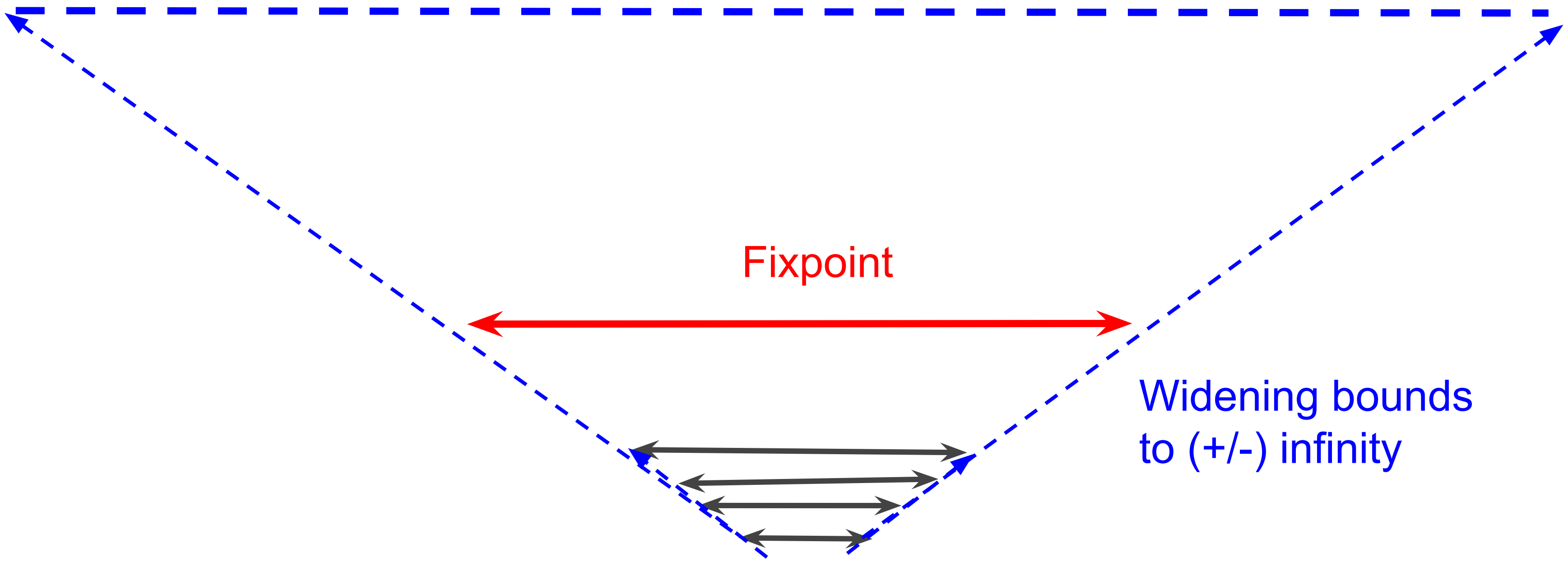
Loops and Fixpoints

- CFG example of a while loop
- Must iterate the bound inference to find a fixpoint.
- Using the widening operator to converge to the fixpoint quickly.



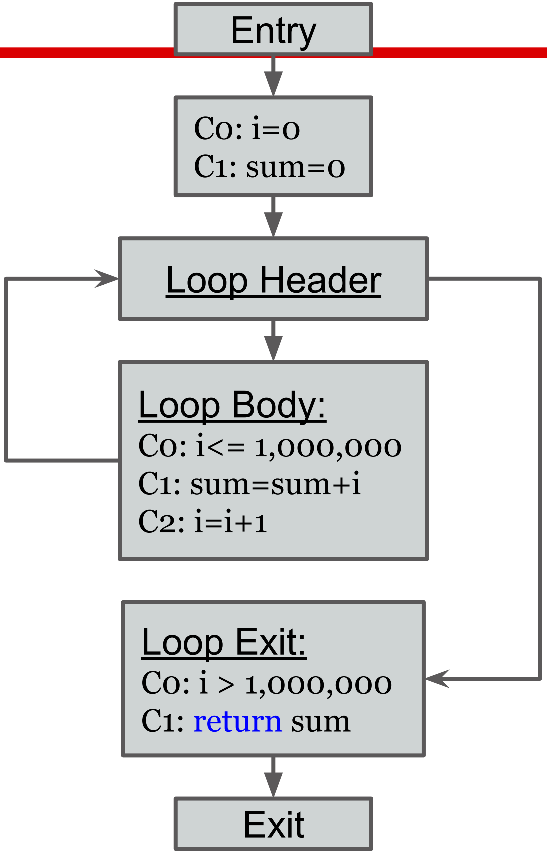
Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. ***Compilers: Principles, Techniques, and Tools,*** chapter 8, pages 529–531. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

Widening Operator



Loop

```
function f() => int
    int i = 0
    int sum = 0
    while i <= 1,000,000:
        sum = sum + i
        i=i+1
    return sum
```



B : {D(i) = [0.. 1,000,001]
D(sum) = [0.. ∞] }

sum ~= 500 billion

Multi-level Widening Operator

- Infinity is too imprecise.
- **Actual ranges**, defined in the compiler, could be used in widening operator.
- Multi-level widening operator widens the lower and upper bounds against a number of thresholds (actual ranges of data types).

Threshold Values

Threshold	Description	Values
INF_MIN	Negative Infinity	-∞
_I64_MIN	Min of ***long long*** Integer	-9,223,372,036,854,775,808
INT_MIN	Min of ***int*** Integer	-2,147,483,648
SHRT_MIN	Min of ***short*** Integer	-32,768
SHRT_MAX	Max of ***short*** Integer	32,767
INT_MAX	Max of ***int*** Integer	2,147,483,647
_I64_MAX	Max of ***long long*** Integer	9,223,372,036,854,775,807
INF_MAX	Positive Infinity	∞

Future work

- Generate efficient C code using the Bound analysis
  - Investigate Su + Wagner's *bound analysis without widening operator*, or Campos *et al.*'s LLVM range analysis algorithm.
- Infer bounds for data structure sizes
  - Use the fixed-size arrays in C code
- Reduce the copying of data structures

This project will improve performance and scalability of Whiley programs while maintaining program correctness.

Thank You!!!