Lifetime Analysis for Whiley
Master's Thesis

Sebastian Schweizer

06.07.2016
Agenda

1. What is Whiley?
2. Lifetimes
3. Lifetime Extension for Whiley
4. Conclusion & Outlook
Agenda

1. What is Whiley?
   - Verification
   - Type System
   - Heap Memory

2. Lifetimes

3. Lifetime Extension for Whiley

4. Conclusion & Outlook
What is Whiley?

- Hybrid imperative and functional programming language
- Focus on verification
  - At compile-time
  - Ensures absence of exceptions (e.g. `IndexOutOfBoundsException`)
  - Verify implementation against provided specification
function max(int[] input) -> (int result) // 1st index with max. val
requires |input| > 0
ensures result >= 0 && result < |input|
ensures all { j in 0..|input| | input[j] <= input[result] }
ensures all { j in 0..result | input[j] < input[result] }:
  result = 0
  int i = 0
  while (i < |input|)
    where i >= 0 && i <= |input|
    where result >= 0 && result < |input| && result <= i
    where all { j in 0..i | input[j] <= input[result] }
    where all { j in 0..result | input[j] < input[result] }:
      if input[i] > input[result]:
        result = i
        i = i + 1
  return result
Type System Features

- Union Types: \texttt{int | bool} \ x = \texttt{true}
- Negation Types: \texttt{!bool} \ x = \texttt{42}
- Structural (Sub-)Typing
- Recursive Types
  - \texttt{type LinkedList is null | \{int head, LinkedList tail\}}
  - \texttt{type A is B | null}
  - \texttt{type B is \{int head, A tail\}}
Heap Memory

- Reference Types: &int, &LinkedList, ...
- Allocation: new Expression
  - &int x = new 5
  - Problem: How to deallocate?
Deallocation

Options

- Manual by programmer (free operator)? Unsafe!
- Using a Garbage Collector? Safe, but runtime overhead!
- Static Analysis!

Current Solution

- Whiley to JVM: Garbage Collector
- Whiley to C: no deallocation
Agenda

1. What is Whiley?
2. Lifetimes
3. Lifetime Extension for Whiley
4. Conclusion & Outlook
Lifetimes

- Pioneered by the *Rust* programming language
- Region in the program's source code
  - region must be a scope, i.e. a block
- Reference with lifetime: guarantee that referenced location has not yet been deallocated
- Static property, checked by type system
- Helps to automatically manage dynamically allocated memory, without garbage collection
Agenda

1. What is Whiley?

2. Lifetimes

3. Lifetime Extension for Whiley
   - Goals
   - Design
   - Subtyping
   - Lifetime Parameters
   - Lifetime Substitution

4. Conclusion & Outlook
Goals

- Backwards compatibility (as much as possible)
- Keep the language simple
- Develop a basis for memory management without garbage collection
  - Future improvement of the *Whiley to C Compiler* (needed for embedded systems)
Designing the Extension

- Reference Type annotated with (optional) lifetime: &a:T
- Allocation operator annotated with (optional) lifetime: a:new expr
- Possible lifetimes: this, *, a
- Named Blocks to declare lifetime names
Example

```java
method main():
    &this:int x = this:new 1

    a:
        &a:int y = a:new 2
    y = x
```
Lifetime Invariant

Invariant

An initialized reference of type &a : T points to a portion of memory that will be alive at least until the program’s control flow leaves the region described by lifetime a.

Definition (Liveness)

A memory location is alive if and only if:

- it has been allocated using the new operator and
- it has not yet been freed by the runtime system
Outlives Relation

Definition (Outlives)

A lifetime $a$ outlives lifetime $b$ (denoted $a \succ b$) if the region described by $b$ is fully contained in the region described by $a$.

Example

```java
method m():
  a:
    // ...
  b:
    c:
      // ...
```

- $\text{this} \succ a$
- $\text{this} \succ b \succ c$
Subtyping

Subtyping of Reference Types

A type \( \&a : A \) is subtype of \( \&b : B \) if and only if
- lifetime \( a \) outlives lifetime \( b \) and
- \( A \) and \( B \) describe the same type

Why the same type?

```
method m():
    \&int x = new 5
    \&(int|null) y = x
    *y = null
```
Lifetime Parameters

- Method with reference types as parameters
- What should be the lifetime?
- Use parametric lifetimes
### Lifetime Parameters

```cpp
1 method <a> m(&a:int x) -> &a:int:
2   if ((*x) == 42):
3     return x
4   else:
5     return a:new 42

method main():
1   &this:int x = this:new 1
2   &this:int y = m<this>(x)
```
Consider the method `method <a> m(&a:int x) -> &a:int:`

Call it with parameter of type `&this:int`

What should be the return type?

Substitute lifetime parameter `a` with lifetime argument `this`, get `&this:int`.

Capturing?
type mymethod is method<a>(&*:mymethod) -> (&a:mymethod)
Algorithm

- Recursively copy all states (depth-first)
- Thereby apply the substitution
- When entering a method type, do not substitute the lifetime parameters in subgraph
- Reuse already copied state if it ignores the same lifetime parameters
- $n$ states and $m$ lifetimes $\Rightarrow$ maximal $n \times 2^m$ states in substituted type
Agenda

1. What is Whiley?
2. Lifetimes
3. Lifetime Extension for Whiley
4. Conclusion & Outlook
Conclusion

- Design lifetime extension
  - Lifetime parameters
  - Lifetime substitution
  - Lifetime argument inference
  - Subtyping

- Implementation
  - 2388 added and 464 removed lines
  - additional: new test cases with 549 lines
  - independent bug fixes: 665 added and 96 removed lines
Goals

- Backwards compatibility (as much as possible)
  - Yes, except for new keyword `this`
- Keep the language simple
  - Yes, introduced only necessary concepts
- Develop a basis for memory management without garbage collection
  - Next slide!
Outlook: Memory Management

allocations:

x:

Stack

Heap

length = 1
dynamic object
Thank you for your attention!

Questions?