TECHNISCHE UNIVERSITÄT
KAISERSLAUTERN

Lifetime Analysis for Whiley

Master's Thesis

Sebastian Schweizer

06.07.2016

Agenda

1 What is Whiley?

Agenda

Agenda

Agenda

Agenda

What is Whiley?

- Hybrid imperative and functional programming language
- Focus on verification

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

```
1  function max(int[] input) -> (int result) // 1st index with max. val
2  requires |input| > 0
3  ensures result >= 0 && result < |input|
4  ensures all { j in 0..|input| | input[j] <= input[result] }
5  ensures all { j in 0..result  | input[j] <  input[result] }:
6
7
8
9
10
11
12
13
14
15
16 //
```

```
1  function max(int[] input) -> (int result) // 1st index with max. val
2  requires |input| > 0
3  ensures result >= 0 && result < |input|
4  ensures all { j in 0..|input| | input[j] <= input[result] }
5  ensures all { j in 0..result  | input[j] <  input[result] }:
6      result = 0
7      int i = 0
8      while (i < |input|):
9
10
11
12
13         if input[i] > input[result]:
14             result = i
15         i = i + 1
16     return result
```

```
1  function max(int[] input) -> (int result) // 1st index with max. val
2  requires |input| > 0
3  ensures result >= 0 && result < |input|
4  ensures all { j in 0..|input| | input[j] <= input[result] }
5  ensures all { j in 0..result  | input[j] <  input[result] }:
6      result = 0
7      int i = 0
8      while (i < |input|)
9      where i >= 0 && i <= |input|
10     where result >= 0 && result < |input| && result <= i
11     where all { j in 0..i       | input[j] <= input[result] }
12     where all { j in 0..result  | input[j] <  input[result] }:
13         if input[i] > input[result]:
14             result = i
15         i = i + 1
16     return result
```

Type System Features

- Union Types: `int|bool x = true`

Type System Features

- Union Types: `int|bool x = true`
- Negation Types: `!bool x = 42`

Type System Features

- Union Types: `int|bool x = true`
- Negation Types: `!bool x = 42`
- Structural (Sub-)Typing

Type System Features

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

Type System Features

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

Heap Memory

- Reference Types: `&int`, `&LinkedList`, ...

Heap Memory

- Reference Types: `&int`, `&LinkedList`, ...
- Allocation: `new Expression`

Heap Memory

- Reference Types: `&int`, `&LinkedList`, ...
- Allocation: `new Expression`
  - `&int x = new 5`

Heap Memory

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

Deallocation

Options

Deallocation

Options

- Manual by programmer (`free` operator)?

Deallocation

Options

- Manual by programmer (`free` operator)? Unsafe!

Deallocation

Options

- Manual by programmer (`free` operator)? Unsafe!
- Using a Garbage Collector?

Deallocation

Options

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

Deallocation

Options

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

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

Lifetimes

- Pioneered by the *Rust* programming language

Lifetimes

- Pioneered by the *Rust* programming language
- Region in the program's source code

Lifetimes

- Pioneered by the *Rust* programming language
- Region in the program's source code
  - region must be a scope, i.e. a block

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

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

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

Goals

- Backwards compatibility (as much as possible)

Goals

- Backwards compatibility (as much as possible)
- Keep the language simple

Goals

- Backwards compatibility (as much as possible)
- Keep the language simple
- Develop a basis for memory management without garbage collection

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`

Designing the Extension

- Reference Type annotated with (optional) lifetime: `&a:T`
- Allocation operator annotated with (optional) lifetime: `a:new expr`

Designing the Extension

- Reference Type annotated with (optional) lifetime: `&a:T`
- Allocation operator annotated with (optional) lifetime: `a:new expr`
- Possible lifetimes: `this`, `*`, `a`

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

```
1  method main():
2    &this:int x = this:new 1
3
4    a:
5      &a:int y = a:new 2
6      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 $\texttt{a} \succ \texttt{b}$) if the region described by `b` is fully contained in the region described by `a`.

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

```
1  method m():
2    a:
3      // ...
4    b:
5      c:
6        // ...
```

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

```
1  method m():
2    a:
3      // ...
4    b:
5      c:
6        // ...
```

- **this** $\succ$ a
- **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

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?

```
1  method m():
2    &int x = new 5
3    &(int|null) y = x
4    *y = null
```

Lifetime Parameters

- Method with reference types as parameters
- What should be the lifetime?

Lifetime Parameters

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

Lifetime Parameters

```
1  method <a> m(&a:int x) -> &a:int:
2    if ((*x) == 42):
3      return x
4    else:
5      return a:new 42
6
7  method main():
8    &this:int x = this:new 1
9    &this:int y = m<this>(x)
```

Lifetime Substitution

- 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?

Lifetime Substitution

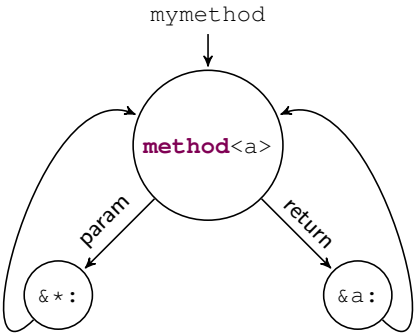
- Consider the method `method <a> m(&a:int x) -> &a:int:`
- Call it with parameter of type `&this:int`
- What should be the return type?
- Substitute lifetime parameter `a` with lifetime argument `this`, get `&this:int`.

Lifetime Substitution

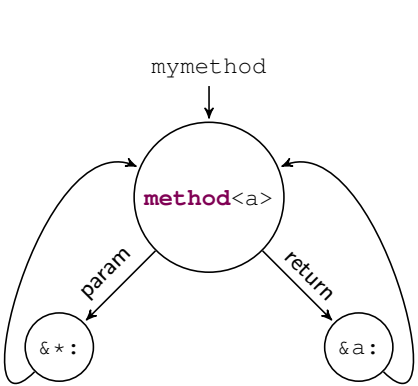
- Consider the method `method <a> m(&a:int x) -> &a:int:`
- Call it with parameter of type `&this:int`
- What should be the return type?
- Substitute lifetime parameter `a` with lifetime argument `this`, get `&this:int`.
- Capturing?

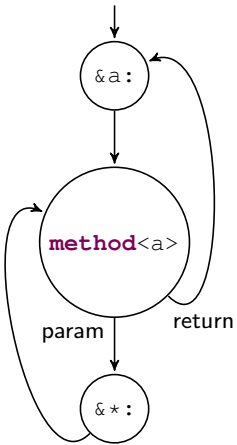
```
type mymethod is method<a>(&*:mymethod)->(&a:mymethod)
```

```
type mymethod is method<a>(&*:mymethod)->(&a:mymethod)
```

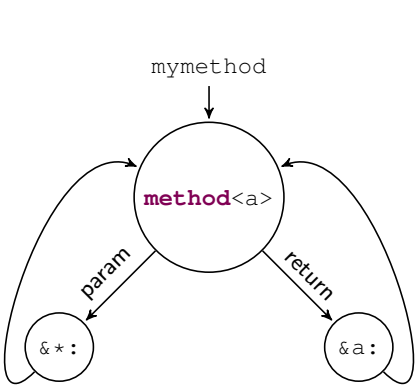


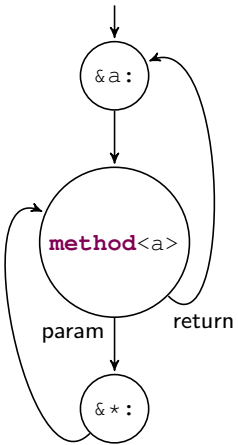
```
type mymethod is method<a>(&*:mymethod)->(&a:mymethod)
```

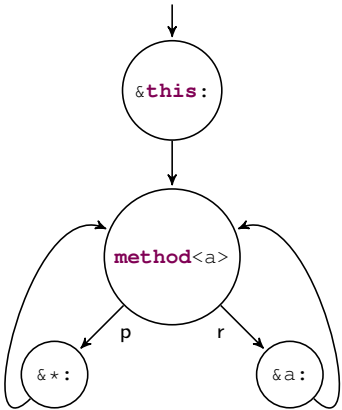




```
type mymethod is method<a>(&*:mymethod)->(&a:mymethod)
```







Algorithm

- Recursively copy all states (depth-first)

Algorithm

- Recursively copy all states (depth-first)
- Thereby apply the substitution

Algorithm

- Recursively copy all states (depth-first)
- Thereby apply the substitution
- When entering a method type, do not substitute the lifetime parameters in subgraph

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

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 $n * 2^m$ states in substituted type

Agenda

Conclusion

- Design lifetime extension

Conclusion

- Design lifetime extension
  - Lifetime parameters

Conclusion

- Design lifetime extension
  - Lifetime parameters
  - Lifetime substitution

Conclusion

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

Conclusion

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

Conclusion

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

Conclusion

- Design lifetime extension
  - Lifetime parameters
  - Lifetime substitution
  - Lifetime argument inference
  - Subtyping
- Implementation
  - 2388 added and 464 removed lines

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

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)

Goals

- Backwards compatibility (as much as possible)
  - ✓ Yes, except for new keyword `this`

Goals

- Backwards compatibility (as much as possible)
  - ✓ Yes, except for new keyword `this`
- Keep the language simple

Goals

- Backwards compatibility (as much as possible)
  - ✓ Yes, except for new keyword `this`
- Keep the language simple
  - ✓ Yes, introduced only necessary concepts

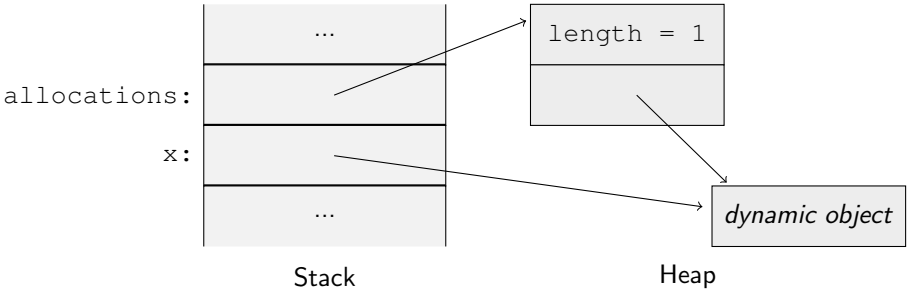
Goals

- 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

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



Thank you for your attention!

Thank you for your attention!

Questions?