Master's Thesis

Lifetime Analysis for Whiley

Sebastian Schweizer

schweizer@cs.uni-kl.de

University of Kaiserslautern
Department of Computer Science
Master's Course of Studies in Computer Science

30.05.2016

Examiner: Prof. Dr. Arnd Poetzsch-Heffter
Software Technology Group

Lifetime Analysis for Whiley

Abstract

Safety critical environments require high programming standards. *Verification* is a way to prove absence of certain faults and to make sure that a program meets a given specification. Unfortunately, most modern programming languages do not actively support verification. External tools are necessary and there is no good integration with the development process.

Whiley is a programming language that aims to popularize verification. Functions can be annotated with pre- and postconditions and the compiler ships with a verifier that ensures that the implementation satisfies the specification. Verification is automatic, the programmer only needs to provide loop invariants. Verified *Whiley* programs are guaranteed to be free of runtime failures like index-out-of-bounds access in arrays and nullpointer dereferences.

Rust is a new programming language that aims to be fast and safe, and it is classified as a systems programming language. *Rust* introduces a revolutionary concept of *ownership* and *lifetimes* that allows for automatic and safe memory management without using garbage collection. A lifetime roughly states how long a dynamically allocated portion of memory can be used. Each allocation belongs to a unique owner, and as soon as the owner's lifetime ends the memory will be freed, without the need for garbage collection. Memory safety is guaranteed by using static checks at compile time.

This thesis extends the *Whiley* programming language to introduce a concept of lifetimes similar to *Rust*. But both programming languages have a different focus. We therefore need to adapt the concept such that it fits *Whiley*'s environment. Our extension involves changes to the language syntax and several parts of the compiler and intermediate code formats. A main challenge is the treatment of lifetimes for subtyping with recursive types and in method invocations.

Whiley currently compiles to bytecode for the *Java Virtual Machine (JVM)*, using its garbage collector to deallocate memory. There is an experimental compiler for *Whiley* that generates *C* code, but it is not yet able to deallocate dynamically allocated memory. We show how the compiler can use lifetimes for memory management without garbage collection, though the actual implementation for this part is left as future work. This allows one to greatly improve the *Whiley* to *C* compiler, which is necessary to run *Whiley* programs on embedded devices that do not have enough resources to execute the *JVM*.

Zusammenfassung

In sicherheitskritischen Umgebungen bedarf es hohen Programmierstandards. *Verifikation* stellt sicher, dass bestimmte Laufzeitfehler nicht auftreten können und dass ein Programm seine Spezifikation erfüllt. Leider bieten die meisten modernen Programmiersprachen keine integrierte Unterstützung für Verifikation. Mit externen Werkzeugen können Programme zwar verifiziert werden, aber sie bieten meist keine gute Integration in den Entwicklungsprozess.

Whiley ist eine Programmiersprache die versucht, Verifikation beliebter zu machen. Funktionen können mit Vor- und Nachbedingungen versehen werden. Der Übersetzer verifiziert das Programm und stellt dabei sicher, dass die Implementierung diese Spezifikationen erfüllt. Verifikation ist automatisiert, der Programmierer muss lediglich Invarianten für Schleifen angeben. Verifizierte *Whiley* Programme enthalten zudem keine Laufzeitfehler wie Zugriffe außerhalb der Indexgrenzen von Arrays oder Dereferenzierung von Nullpointern.

Rust ist eine neue Programmiersprache die sich zum Ziel gesetzt hat, schnell und sicher zu sein. Sie ist als Systemprogrammiersprache klassifiziert. *Rust* führt ein revolutionäres Konzept von *Ownership* und *Lifetimes* ein. Damit ist es möglich, Speicher automatisiert zu verwalten und Speichersicherheit zu garantieren, ohne auf Garbage Collection angewiesen zu sein. Eine Lifetime beschreibt grob gesagt wie lange ein dynamisch zugeteilter Speicherbereich benutzt werden kann. Jede Zuteilung gehört immer einem Besitzer (*Owner*) und sobald dessen Lifetime endet kann der zugeteilte Speicher freigegeben werden. Dazu ist keine Garbage Collection notwendig. Speichersicherheit wird durch statische Checks während der Übersetzung sichergestellt.

In dieser Masterarbeit erweitern wir *Whiley* um ein Konzept für Lifetimes ähnlich wie in *Rust*. Die beiden Programmiersprachen haben jedoch einen unterschiedlichen Fokus, sodass wir einige Anpassungen am Konzept vornehmen müssen, damit es zur Umgebung von *Whiley* passt. Für unsere Erweiterung sind Änderungen an der Sprachsyntax und einigen Teilen des Übersetzers sowie der Zwischensprache von *Whiley* nötig. Der Umgang mit Lifetimes in Verbindung mit Subtyping von rekursiven Typen und bei Methodenaufrufen ist eine Herausforderung.

Derzeit wird *Whiley* in Bytecode für die *Java Virtual Machine (JVM)* übersetzt, dessen Garbe Collector zur Freigabe von dynamisch zugewiesenem Speicher benutzt wird. Es gibt einen experimentellen Übersetzer für *Whiley*, welcher *C* code generiert, aber dieser kann dynamisch zugewiesenen Speicher nicht wieder freigeben. Wir zeigen, wie der Übersetzter Lifetimes zur Speicherverwaltung ohne Garbage Collection verwenden kann. Dies bereitet einen Weg für eine bedeutende Verbesserung des *Whiley* zu *C* Übersetzers. Dadurch wird es möglich, *Whiley* Programme auf eingebetteten Geräten auszuführen, die nicht genug Ressourcen für die *JVM* haben.

Acknowledgments

I want to thank Dr David Pearce of the School of Engineering and Computer Science at Victoria University of Wellington. He provided me an excellent topic to work on and steered me in the right direction when I got stuck. I very much appreciate our regular discussions on my work and current development of *Whiley*.

I would also like to thank my examiner Prof. Dr. Arnd Poetzsch-Heffter of the Software Technology Group at University of Kaiserslautern for getting me known to David and for the help while planning my journey.

I am grateful to all the nice people that I have met in New Zealand. You made my stay there so far away from my homeland especially enjoyable. And I would like to thank my parents and friends at home. Your encouragement bridged such a big distance.

Finally, I want to thank the German Academic Exchange Service for fundings as part of their PROMOS program, and IBM who supports the German national "Deutschland-stipendium" scholarship where I got fundings from in my earlier studies.

Statutory Declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

Kaiserslautern, 30.05.2016

Sebastian Schweizer

Eidesstattliche Erklärung

Ich versichere hiermit, dass ich die vorliegende Masterarbeit mit dem Thema *Lifetime Analysis for Whiley* selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe. Die Stellen, die anderen Werken dem Wortlaut oder dem Sinn nach entnommen wurden, habe ich durch die Angabe der Quelle, auch der benutzten Sekundärliteratur, als Entlehnung kenntlich gemacht.

Kaiserslautern, den 30.05.2016

Sebastian Schweizer

Contents

1. Introduction 1
- 1.1. Whiley . . . 2
- 1.2. Contributions . . . 2
- 1.3. Organization . . . 3

2. Background 4
- 2.1. Rust and Lifetimes . . . 4
  - 2.1.1. Stack, Heap and Ownership . . . 4
  - 2.1.2. Move Semantics . . . 5
  - 2.1.3. Boxes and Reference-Counted Pointers . . . 6
  - 2.1.4. Borrowing . . . 7
  - 2.1.5. Lifetimes . . . 8
  - 2.1.6. Variance and Subtyping . . . 10
  - 2.1.7. Summary: Different Ways to Store and Share Values . . . 10
- 2.2. Whiley . . . 11
  - 2.2.1. Design . . . 11
  - 2.2.2. Functions, Methods and Specification . . . 12
  - 2.2.3. Verification . . . 13
  - 2.2.4. Structural Typing and Records . . . 15
  - 2.2.5. Union Types . . . 16
  - 2.2.6. Type Declarations and Recursive Types . . . 16
  - 2.2.7. Flow Typing . . . 18
  - 2.2.8. Heap-Allocated Memory and References . . . 19
- 2.3. Related Work . . . 20
  - 2.3.1. Ownership . . . 20
  - 2.3.2. Region-Based Memory Management . . . 20
  - 2.3.3. Alias Analysis . . . 20
  - 2.3.4. Separation Logic . . . 21

3. Design 22
- 3.1. Motivation . . . 22
- 3.2. Syntax . . . 24
- 3.3. Lifetimes . . . 27
  - 3.3.1. Subtyping . . . 29
  - 3.3.2. Lifetime Parameters . . . 29
  - 3.3.3. Context Lifetimes . . . 30

3.4. Discussion . . . 30
3.4.1. Backwards-Compatibility . . . 30
3.4.2. Mutability and Move Semantics . . . 31
3.4.3. References to Local Variables . . . 32
3.4.4. Pointer Types . . . 33
3.4.5. Lambda Expressions . . . 33

4. Implementation **36**
4.1. Lexing and Parsing . . . 36
4.1.1. Abstract Syntax Tree . . . 36
4.1.2. Lexer . . . 37
4.1.3. Parser . . . 37
4.2. Type Checking . . . 39
4.2.1. Lifetime Relation . . . 40
4.2.2. Reference Types . . . 40
4.2.3. Method Types . . . 41
4.3. Lifetime Substitution and Method Lookup . . . 42
4.3.1. Substitution in Return Types . . . 43
4.3.2. Substitution for Subtyping . . . 45
4.3.3. Method Lookup . . . 46
4.4. Intermediate Language . . . 47

5. Evaluation **48**
5.1. Test Cases . . . 48
5.2. Memory Management . . . 49

6. Conclusion **51**
6.1. Contribution . . . 51
6.2. Future Work . . . 52
6.3. Conclusions . . . 53

A. Bibliography **55**

1. Introduction

Programmers today have the choice among dozens of different programming languages. They can be classified in paradigms. These are for example functional languages (Haskell, Erlang, Racket), procedural languages (C, Fortran, Pascal) or object-oriented languages (Java, C++, Perl). Programming languages can be compiled (C, C++, Pascal) or interpreted (Perl, Python, Ruby). Another difference is the level of abstraction regarding memory management: some languages like C delegate all memory management to the programmer, others use automated techniques to free unused memory. The first approach, *manual memory* management, comes with some drawbacks: modern programs are inherently complex and especially when in comes to multi-threaded software, memory management can be quite cumbersome. While forgetting to free memory in time only leads to memory leaks, a premature deallocation generates dangling pointers. Accessing such a pointer is referred to as *use after free*. It results in undefined behavior, as the memory pointed to may already have been reallocated to store something else.

One approach to free programmers from the burden of manual memory management is *garbage collection*. The runtime environment continuously analyzes which memory is no longer reachable by the running program and frees it accordingly. One way to implement it is reference counting. This is known to work well with acyclic structures [4], but more complex analysis is needed if structures are cyclic. Other algorithms are for example mark-and-sweep and copying collection [31]. Garbage collection was made popular by the *Java* programming language [30].

All techniques for garbage collection impose some form of runtime overhead. Depending on the algorithm in use it might also be impossible to guarantee specific reaction times, as garbage collection can nondeterministically interrupt program execution [2]. While today's computers get more and more powerful such that garbage collection might not be a problem, technology comes up with small embedded devices like drones or even small computers as part of pacemakers. Here we need a small and energy efficient runtime environment, but cannot allow for any avoidable programming bugs as they might cause serious incidents with those kinds of devices. Another field where garbage collection is usually not applicable is operating systems development.

One solution to this trade-off between performance and safety is to statically analyze the program and let the compiler infer all necessary deallocation points. *Rust* is a new programming language that employs a notion of ownership and lifetimes. The programmer is enforced to obey some constraints and the compiler ensures that program execution will be safe, without having an expensive runtime environment. In fact, *Rust* programs show a similar performance to *C* programs [3].

1.1. Whiley

The way how *Java* guarantees memory safety is to add some runtime checks. When accessing an array there will be a check that the index is within the array's bounds. Another common check is to ensure that a dereferenced pointer is not `null`. A `RuntimeException` gets thrown if one of these checks fails.

An alternative to this approach is *verification*. Given a program, you should be able to statically prove that each array access or pointer dereference will be safe for every possible program execution. This technique ensures that these runtime faults cannot occur at all and renders those checks redundant. Another application is *specification*. The programmer annotates each function with a mathematical description of what is expected to happen. Using verification, we then can check that this is actually the case for the given implementation.

Most programming languages do not provide assistance for these tasks. If at all possible, programmers need to use special extensions or external tools to specify and verify their software. As a result, casually written programs are often neither verified nor formally specified. *Whiley* is a programming language that aims to make specifying and verifying software a common practice [18]. Included directly in the language core, the compiler ships with an integrated verifier that checks all functions and methods against their specification. Furthermore, it ensures absence of runtime exceptions as described above. This provides a perfect base for safety critical systems. However, *Whiley* currently compiles to byte code for the *Java Virtual Machine* (JVM), which comes with a rather large runtime footprint unsuitable for some kinds of smaller embedded devices. There have been efforts to compile *Whiley* to embedded devices, but memory management was a big challenge, because *Whiley* currently relies on garbage collection done by the JVM [28, 20].

1.2. Contributions

In this thesis, we extend the *Whiley* programming language to introduce a concept of lifetimes. Similar to *Rust*, each lifetime describes a region in the source code where a specific reference is considered to be alive. During that timespan, the memory pointed to is guaranteed to still be available. Afterwards it can safely be freed by the compiler.

We extend the language syntax to support lifetime annotations. Several internals of the *Whiley* compiler are affected by our changes, e.g. the type system needs to statically check constraints imposed by lifetimes.

We implement our changes and add several test cases. The *Whiley* maintainer accepted our code and released a new version including lifetimes[1].

Our extension can be used for improved memory management. We describe an approach how to implement automatic and safe deallocation of heap-allocated memory without garbage collection. This allows to write *Whiley* compilers targeting embedded devices that are not capable of running a full JVM.

[1] http://whiley.org/2016/05/28/whiley-v0-3-40-released/

1.3. Organization

Chapter 2 gives a short introduction to the *Whiley* and *Rust* programming languages. We do not cover every detail of these languages, but we establish a basic knowledge of the key features and concepts which are necessary to understand our contribution. Afterwards we examine other related work.

In chapter 3, we give a high level view of our lifetime extension. We first motivate our changes and then present the new syntax and meaning of lifetimes. Finally, we discuss possible alternatives and why they have been discarded.

Chapter 4 covers the actual implementation of our language extension in the *Whiley* compiler. We provide an overview of the necessary code changes and elaborate on some difficult parts.

We give an evaluation of our work in chapter 5. We analyze the performance impact for checking lifetimes using the test programs shipping with *Whiley*. Furthermore, we sketch a possible implementation for memory management using lifetimes to show how the *Whiley* to *C* compiler and possibly other backends can benefit from our work.

Chapter 6 concludes this thesis and shows options for future work.

2. Background

Rust is a multi-paradigm programming language. It claims to be "a systems programming language that runs blazingly fast, prevents segmentation faults, and guarantees thread safety" [24]. To achieve this goal, *Rust* avoids garbage collection and other runtime overhead. Instead, static analysis as well as a notion of *ownership* and *lifetimes* is used to ensure memory safety.

Whiley also is a mutli-paradigm compiled programming language. A major feature is *verification*: programmers can provide specifications for all functions and the compiler checks that they are met by the actual implementation. *Whiley* uses a structural type system and features *flow typing*.

For our contribution in this thesis we adapt *Rust*'s notion of lifetimes to introduce a similar approach to the *Whiley* language. As basic knowledge of both languages and their differences is essential to understand our work, this chapter provides a brief introduction to *Rust* (section 2.1) and *Whiley* (section 2.2). In 2.3, we present some work related to static analysis techniques.

2.1. Rust and Lifetimes

We provide here a short introduction of some concepts of *Rust* that are relevant for this thesis. A more detailed introduction of the programming language can be found in the official *Rust* book [25] and the *Rustonomicon* [26].

Rust aims to provide a fast and yet safe programming language. Our focus is to examine its memory model, to see how it can provide automatic memory management without garbage collection. The three main terms related to this are *ownership*, *borrowing* and *lifetimes*. Reed [22] established a formal model for the key features that provide memory safety and proved soundness of that model.

2.1.1. Stack, Heap and Ownership

Like many programming languages, *Rust* allows to allocate memory on the stack or on the heap. The stack is an automatically managed portion of memory, where each function execution allocates a so-called *stack frame* that contains all local variables. The stack-frame will be freed automatically once the function returns and therefore anything allocated there cannot outlive the function itself. On the other hand, the heap is a portion of memory that is managed more dynamically. Memory there can live longer than a single function execution.

In *Rust*, we can use *variable bindings* to bind a value to a name. The following program simply binds an integer `42` to the variable `x`.

```
1 fn main() {
2   let x = 42;
3 }
```

The compiler will insert an integer with value `42` into the current stack frame and use its address whenever `x` is used.[2] Variable `x` is said to *own* that portion of memory. When the owner goes out of scope, the memory will be freed. The scope of a variable is just the enclosing pair of curly braces `{ }`. In this example there is nothing special to do, as the stack frame holding our integer will be freed automatically once function `main` returns.

Consider now a different program. Since all addresses within a stack frame have to be calculated at compile time, we cannot store constructs of unknown size in it. Instead, that data has to be stored on the heap. One example for constructs of variable size in *Rust* are vectors.

```
1 fn main() {
2   let fib = vec![1, 1, 2, 3, 5, 8];
3 }
```

This example creates a vector containing the first six Fibonacci numbers. Internally, the vector is stored as a triple denoting its current length, its capacity and a pointer to the actual data on the heap. In line 3, variable `fib` goes out of scope. The vector owned by it will then be deallocated, which includes freeing its data on the heap. This deallocation step happens directly and deterministically in line 3. There is no need for reference counting or a garbage collector that eventually finds some unreachable vector data on the heap.

Note that a vector in *Rust* differs from an `ArrayList` in *Java*: to access an element in a vector, we have to take the address of the data pointer, add a calculated offset and access the resulting address. These pointers are called *fat pointers*, because they store additional information (length and capacity) together with the address. For an `ArrayList`, we must first dereference the list object. Then we read the reference to the contained array, add the appropriate offset and access that address. There are two dereference operations necessary while only one is needed for vectors in *Rust*. Dereferences are expensive, because the machine's caching system does not benefit from fetching larger blocks [13].

2.1.2. Move Semantics

One important property of *Rust*'s safety system is that there must only be one owner for each memory location at any time. When we simply assign one variable to another, then ownership is transferred because *Rust* uses so-called *move semantics*. Consider the following modified program:

[2] For the sake of simplicity we do not consider unspilled variables stored in registers and ignore optimizations like constant propagation.

```
1 fn main() {
2   let mut fib = vec![1, 1, 2, 3, 5, 8];
3   let mut fib2 = fib;
4   fib2.push(13);
5   println!("fib[0] is {}", fib[0]); // illegal
6 }
```

First of all we notice the newly inserted **mut** keyword. It states that the bound value is mutable, i.e. we can later modify the assigned vector. In line 3, we assign our vector from variable `fib` to a new variable `fib2`. In line 4, we append one more number to that vector. The following will happen internally: we start with our initial vector, where length, capacity and a pointer is stored on the stack and the actual data on the heap. In line 3, we *move* `fib` to `fib2`. Moving means that all data at `fib`'s memory location will be copied to `fib2`. So we copy the triple consisting of length, capacity and pointer, but not the actual vector elements. The operation therefore runs in constant time, independent of the vector's length.

Now we push a new entry to that vector. That updates the vector's length and adds the new element. If the capacity does not allow to add an element, a new and bigger portion of memory will be allocated on the heap such that the data can be transfered there. Afterwards, the pointer is updated and the old heap data freed.

The last line now tries to access the vector through our old name `fib`. This is illegal, as the value has moved to the new name `fib2`. If we allow accessing it via its old variable, then we might access a dangling pointer. The `push` call in the previous line might have deallocated the old data array after allocating a bigger one. The semantics of *Rust* therefore forbids using moved variable bindings.

For some types this protection provided by move semantics is not necessary. For example, a primitive type like a number or boolean value can just be copied without invalidating the old binding. *Rust* uses a marker trait called `Copy` to identify those types. After doing an assignment with such a type, there are two independent values with separate owners. All primitive types are `Copy`. `Copy` can be derived for records if all their fields are `Copy`. More complex types like vectors are not `Copy`.

Some types that are not marked as `Copy` can still be *cloned* explicitly, but this process involves creating a complete deep copy, which is expensive for big data structures.

2.1.3. Boxes and Reference-Counted Pointers

Even bigger structures can be allocated on the stack, if their size is statically known. To avoid moving around big structures, we can explicitly allocate them on the heap. *Rust* uses the type `Box` for a heap-allocated portion of memory. A `Box` is just a pointer to the heap. It is similar to a vector with length one, but as the length cannot change there is no need to store length or capacity. Each `Box` has a single owner. Once the owner goes out of scope, the `Box`'s content on the heap will be deallocated. For moving a `Box`, only the pointer itself has to be copied into the new location. This happens analogue to the behavior already described for vectors. Boxes are allocated with `let b = Box::new(5)` and

can be accessed as `*b`.

In some cases it might be very difficult to find a flow through your program such that a `Box` always has only one owner. For these cases, *Rust* provides the type `Rc` which are *reference-counted* pointers. This type imposes a runtime overhead, as it manages a counter that tracks how many pointers to it exist. The `Rc` owns its contained value and deallocates it when the last pointer to it goes out of scope.

`Rc` cannot be shared over different threads in a multi-threaded scenario. There is a special type `Arc` (*atomic* `Rc`) for that use case. It has proper synchronization and therefore imposes more overhead than the simpler `Rc`.

The programmer has to be aware that cyclic reference counted structures will not be freed automatically and therefore lead to memory leaks [15]. These cycles have to be broken manually.

2.1.4. Borrowing

Instead of moving the value and thereby transferring ownership to another variable, the owner can also *lend* a value to another variable that *borrows* it for a specific amount of time. Borrowing allows to give another variable or a called function access to a value without transferring ownership.

```
1 fn main() {
2   let fib_original = vec![1, 1, 2, 3, 5, 8];
3   let fib_borrowed = &fib_original;
4   println!("The 4th fibonacci number is {}", fib_original[4]);
5   println!("The 5th fibonacci number is {}", fib_borrowed[5]);
6 }
```

The notation `let fib_borrowed = &fib_original` states that `fib_borrowed` borrows the vector from `fib_original`. We can access it through both names, as `fib_borrowed` is just a reference to the original value. This immediately should raise the question how to prevent the situation described above, where one of both variables contains a dangling pointer after an update.

To see how that problem is solved, we again need to consider mutability. As seen before, variable bindings must be declared with the `mut` keyword to allow the value to be modified. There are mutable and immutable values. The same holds for borrowed references: A value can be borrowed mutable or immutable. Consider the following program:

```
1 fn main() {
2   let mut fib_original = vec![1, 1, 2, 3, 5, 8];
3   let fib_borrowed = &fib_original;
4   println!("The 4th fibonacci number is {}", fib_original[4]);
5   fib_original.push(13); // illegal
6   println!("The 5th fibonacci number is {}", fib_borrowed[5]);
7 }
```

This program contains four borrowings: the immutable borrowing in line 3 is directly visible. But there are two more immutable borrowings in line 4 and 6 to read the value that is printed. And finally there is a mutable borrowing in line 5. The type `Vec<T>` contains a function `fn push(&mut self, value: T)`, which is the one called in line 5. `&mut self` hereby means that the object receiving the function call is borrowed mutable until the function returns. Another way to borrow the vector mutable would be to replace line 3 by `let mut fib_borrowed = &mut fib_original;`.

One fundamental principle of *Rust*'s borrow system is that there can be only a single mutable borrowing or multiple immutable borrowings at any time. In line 3 of the program above `fib_borrowed` borrows `fib_original` immutable. That borrowing remains as long as `fib_borrowed` stays in scope, i.e. until the closing brace in line 7. Line 4 is allowed, as it just initiates a second immutable borrowing lasting for that single statement, and multiple immutable borrowings are allowed. If we had changed line 3 to a mutable borrowing, line 4 would already have been illegal as it would have combined a mutable borrowing with an immutable one.

Line 5 attempts to borrow `fib_original` mutable while it is still borrowed immutable by `fib_borrowed`. This is not allowed and the program gets rejected by the compiler's borrow checker.

These constraints might look restrictive, but they provide a safe memory model without additional runtime checks. Data races are avoided by design, as they would need two pointers to the same variables where at least one of them is mutable. Consider the following example that iterates over a vector:

```
1 fn main() {
2   let mut fib = vec![1, 1, 2, 3, 5, 8];
3   for x in &fib {
4     // ...
5   }
6   fib.push(13);
7 }
```

Hidden in syntactic sugar within line 3, we implicitly call the `iter` function on `fib` which borrows the vector immutable until the end of our loop. Therefore, we cannot modify the vector within the loop. However, line 6 is fine as the immutable borrowing ends in line 5 and we can borrow the vector mutable afterwards. Modifying what you are iterating over is problematic also in other languages. If you try to modify a Collection in *Java* while iterating over it, the iterator will throw a `ConcurrentModificationException` at runtime. *Rust* does not allow that situation and the program does not compile in the first place.

2.1.5. Lifetimes

Memory whose owner goes out of scope will be freed. This is either unavoidable, as the memory is allocated on the stack and the enclosing stack frame will be deallocated once the function returns, or it is by design to manage heap-allocated memory without garbage

collection. In either case we must ensure that there are no other accessible references to the freed portion of memory. This is why *Rust* enforces one simple rule for borrowed references: a borrowed reference must not outlive the actual value. Consider the following program:

```
1 fn main() {
2   let mut x = &1;
3   {
4     let y = 2;
5     x = &y; // illegal
6   }
7   println!("x points to value {}", *x);
8 }
```

Line 2 allocates memory for an integer with value `1`. A pointer to this memory is stored into another portion of memory, which is bound to variable `x`. The curly braces start a new scope that spreads from line 3 to 6. Inside this scope, we allocate another integer that is bound to `y`. Afterwards we try to store a reference to it into `x`. This is not allowed! The memory bound to `y` will be freed once `y` goes out of scope. This is at line 6. As `x` is still in scope, we might still access it. In fact, we attempt to read the integer pointed to by `x` in line 7. As `y` has been deallocated, we access a dangling pointer.

The *Rust* compiler rejects the program above due to *lifetimes*. Each value and each borrowing is assigned with a lifetime. It describes a part of the program where that value can be used, i.e. roughly the scope of its declaration. The lifetime for a local variable starts at its declaration and ends at the end of its enclosing scope. In our example, lifetime of `x` goes from line 2 to line 8 and lifetime of `y` from line 4 to line 6. In line 5, `x` tries to borrow the value of `y`, but it's own lifetime is longer than the lifetime of `y`. We say that the lifetime of `y` does not *outlive* the lifetime of `x`. This is not allowed. There is no runtime overhead for checking lifetimes, as they are just a restriction checked by the compiler to ensure memory safety.

To handle lifetimes across functions, *Rust* offers lifetime parameters. Consider the following program:

```
1 fn inc<'a>(x : &'a mut u32) -> &'a mut u32 {
2     *x = *x + 1;
3     return x;
4 }
5
6 fn main() {
7   let mut x = 1;
8   let p = inc(&mut x);
9 }
```

Function `inc` is declared to take one parameter `x` that is a mutable reference to an unsigned 32 bit integer. The return value will also be a reference to such an integer. And the annotated lifetime `'a` states that both references will have the same lifetime. While the above program is fine, the following modification will not compile:

```
1 fn main() {
2   let p : &u32; // uninitialized declaration
3   {
4     let mut x = 1;
5     p = inc(&mut x); // illegal
6   }
7   println!("p points to {}", *p);
8 }
```

Because `inc` returns a reference of the same lifetime as it gets passed as as an argument, the lifetime of the argument `x` in line 5 must be at least the lifetime of `p`, where the result gets assigned to. But `x` does not outlive `p` because it is declared in a smaller scope.

If the function has only one reference type parameter, then you can leave out lifetime parameters, as *Rust* automatically declares a lifetime parameter for that reference type and assigns it to all reference types in the return value. This feature is called *lifetime elision*[3].

2.1.6. Variance and Subtyping

In *Rust*, lifetimes are part of the type system and must therefore be considered when talking about subtyping. To understand the subtyping constraints defined by a programming language, it is helpful to ask the following question: *if I expect type T, what other types can I be given without breaking safety?* For the following, we consider two types *Square* and *Rectangle*, where *Square* is a subtype of *Rectangle*.

If you expect to get an immutable reference to a rectangle, you can safely use a mutable one instead; you just do not make use of its ability to mutate the value pointed to. There is also no problem if the provided reference is actually a reference to a square, because its special characteristic does not affect read-only access. Lastly, if the given reference has a longer lifetime than expected, then you can still safely access it.

If the expected reference is a mutable reference to a rectangle, then the provided one must obviously be mutable. But now we cannot accept a reference to a square: you might want to store a non-quadric rectangle to it. This would break safety for other readers that still expect the reference to point to a square. To summarize and formalize, the following rules hold for subtyping in *Rust* (cf. [26]):

- `&'a mut T` is a subtype of `&'a T`
- `&'a T` is covariant over `'a` and `T`
- `&'a mut T` is covariant over `'a` but invariant over `T`

2.1.7. Summary: Different Ways to Store and Share Values

We have seen different ways to store values in a *Rust* program. Most importantly, every value needs a single owner at all times. A value can be directly owned by a local variable.

[3] Lifetime elision actually can handle more cases than described here. They are not needed for a basic understanding and therefore left out for simplicity.

These *variable bindings* are *immutable* unless declared as *mutable* using the **mut** keyword. A normal assignment always *moves* the value, which makes the former name invalid for future access.

To share a value, it can be *borrowed* by another reference, using the `&T` and `&mut T` types. A borrowing is immutable unless the **mut** keyword is used. The borrow checker imposes constraints to prevent data races. Furthermore, each borrowing has a *lifetime* that must be within the lifetime of the borrowed value.

To store a value on the heap, a `Box` can be used. It owns its contained value, but needs a single owner for itself. If you are unable to provide a single owner for a value, then it can be wrapped in one of the types `Rc` or `Arc`, but that imposes runtime overhead.

2.2. Whiley

This section provides an introduction to the *Whiley* programming language. We cover the language's main goals and all parts that are relevant for our extension to *Whiley*.

A more complete specification of *Whiley* is available in the official *Whiley Language Specification* [18]. It introduces *Whiley* as follows:

> "Whiley is a hybrid imperative and functional programming language designed to produce programs with as few errors as possible. Whiley allows explicit specifications to be given for functions, methods and data structures, and employs a verifying compiler to check whether programs meet their specifications." [18, p. 7]

2.2.1. Design

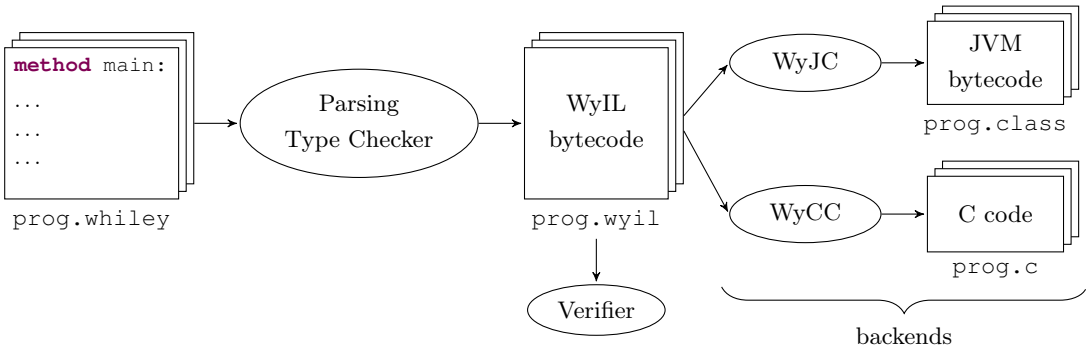


Figure 1: Design of the *Whiley* compiler

Whiley consists of different components as shown in Figure 1. Programs are written in the *Whiley* language and stored as `.whiley` files. Compilation happens in two steps: The program is first translated to the *Whiley Intermediate Language (WyIL)*. The WyIL program is then passed to a *backend* which generates code for a destination language. Currently, the only usable backend is the *Whiley to Java Compiler (WyJC)*, which

generates byte code for the *Java Virtual Machine (JVM)*. There is a backend to generate *C* code, but it does not free any memory. Therefore it is not yet usable for large or long-running programs, or for small embedded systems that do not have a large amount of memory.

Furthermore, the program can be *verified*. This is an optional but recommended step during compilation. Verification is presented in section 2.2.3.

2.2.2. Functions, Methods and Specification

Whiley distinguishes between *functions* and *methods*. The former one are *pure*: a function will always return the same result if the same arguments are given. Furthermore, functions cannot have any side-effects. Methods do not obey these restrictions. The can dereference pointers and issue I/O operations. It is not allowed to call a method from a function.

Whiley uses a call-by-value semantics. The function parameters can therefore be assumed to be a deep copy of the arguments given by the caller. Furthermore, all values are copied on assignment to other variables. Even when a function called with an array `a` assigns `a[0] = 2`, it does not affect the caller's array. This is similar to the types in *Rust* that are marked with `Copy`. There is no aliasing for values in *Whiley*, unless references are used. References will be introduced in section 2.2.8.

Specification of functions and methods in *Whiley* follows a simple precondition-postcondition schema. Both functions and methods can be specified. If a precondition is given, then it is not allowed to call that function or method unless the precondition is satisfied for the given arguments. After executing the function or method, the postcondition must be satisfied. Both is statically checked by the compiler, provided that it is used with the `-verify` flag.

Consider a function that takes an array of numbers and yields the position of the maximum entry. If the maximum appears more than once in our array, then the first position shall be given. But what should happen if the array is empty? In *Whiley*, you typically specify that the function must not be called with an empty array. A possible specification is as follows:

```
1 function max(int[] input) -> (int result)
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   // ...
```

Listing 1: Specification for our max function

Line 1 declares the parameters and the return type. Line 2 states the precondition: length of the given array must be greater than `0`. In line 3 to 5 we give several post-conditions. All **`ensures`** clauses are semantically connected as conjunctions. We can reference the returned value by the name declared together with the return type, `result` in our case. The first postcondition specifies that the returned result is a valid index in the given array. The second condition ensures that the returned position actually holds a

maximal value, i.e. all array values are less or equal than the maximal value. The last condition ensures that we return the first maximal position: all smaller positions hold values that are smaller (but not equal).

2.2.3. Verification

A simple implementation for the program specified in listing 1 is as follows:

```
1 int result = 0
2 int i = 0
3 while (i < |input|):
4   if input[i] > input[result]:
5     result = i
6   i = i + 1
7 return result
```

The idea is as follows: we start with `0` as position holding a maximal value. Then we iterate through the array. If we find a position with a greater value, then we remember that new position as holding the maximal value.

But if we enable verification, then the program will not compile. *Whiley*'s verifier can handle sequential program flow quite well. It applies *Hoare Logic* and *strongest postcondition transformation*. This does not work for loops. We have to give a *loop invariant* to help the theorem prover that internally does the verification. For values that are not assigned inside the loop we do not need to give any additional invariants. But our loop contains assignments to `i` and `result`, so we need to come up with an invariant that somehow contains these values.

Listing 2 presents a working implementation that passes verification. Loop invariants

```
1  function max(int[] input) -> (int result)
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
```

Listing 2: Specified and verified max function

are given using the **where** keyword. Multiple invariants are semantically connected as conjunctions. Loop invariants must be satisfied before entering the loop and after executing the loop body.

Our first invariant states that the loop variable `i` is in bounds of the `input` array, or equal to the arrays length. The latter one is for the case after the last loop iteration. One job of the verifier is to ensure that each array access is in bounds. This invariant combined with the loop condition ensures that `input[i]` is always in bounds.

The second invariant states that `result` is always a valid index and it is smaller or equal to the current loop variable `i`. On entry, both variables are **0**, which satisfies that condition because the array is not empty. If `result` is updated, then it is set to the current index which is known to be in bounds. The invariant is therefore restored after each loop iteration.

The last two invariants are very similar to the function's postcondition. The only difference is that we consider only values up to position `i`. Therefore these conditions trivially hold on loop entry. It is easy to see that the loop restores both conditions with every iteration.

After exiting from the loop, it is known that `!(i < |input|)` holds. Furthermore, `i <= |input|` from our invariants still holds. The combination of both yields `i == |input|`. Combined with invariant from line 11 we can satisfy the postcondition in line 4. The remaining postconditions are directly contained in our loop invariants.

Assert and Assume

Besides loop invariants, *Whiley* offers two statements that interact with verification: **assert** and **assume**. Both statements will check at runtime that the given condition is satisfied. At verification time, i.e. on compile time if the `-verify` flag is given, they influence the verifier. If **assert** is used, then the verifier must find a proof for that condition. The condition will then automatically hold for all program executions.

On the other hand, **assume** inserts the given condition as an additional assumption at that position. It can be used to omit verifying at some places, e.g. when the theorem prover cannot find a suitable proof. A wrong use of **assume** can compile unsound programs. The program still aborts at runtime because of the invalid assumption, but it passes the verifier at compile time although it might contain an out-of-bounds array access:

```
1 int[] a = [1, 1, 2, 3, 5, 8]
2 assert |a| == 6
3
4 int i = 7
5 assume i >= 0 && i < |a|
6 int x = a[i] // out of bounds check effectively disabled
```

2.2.4. Structural Typing and Records

Another main feature of *Whiley* is *structural typing*. To understand this class of type systems, we look at the differences between nominal and structural typing. In nominal typing, an entity's type is defined by the use of a specific name. Even if two type declarations have exactly the same definition, they declare two different types. In structural type systems, an explicit type declaration is not needed. The type of an entity is derived from its structure. Two types with equivalent structures are considered to be the same.

One example for a use of structural types in *Whiley* are *records*. A record is a compound that contains *fields*. Each field has a name and a type. There is no specific order among record fields. Consider the following program:

```
1 {int x, int y} rec1 = {y: 7, x: 8}
2 {int y, int x} rec2 = {x: 8, y: 7}
3 assert rec1 == rec2
4
5 assert rec1.x == 8
6 rec1.x = 9
7 assert rec1 != rec2
```

Focus on the first half. We assign two local variables with records. Although the order of fields does not match, these assignments are valid because records are unordered. Furthermore, the assertion in line 3 is correct, i.e. both records have the same type and value. The second half of the program illustrate how we can read and write to record fields.

Record types can be *open*. An open record allows to have additional fields that are not specified by the type itself. Consider the following code snippet:

```
1 {int x, int y, ...} rec1 = {y: 7, x: 8, z: 9, hello: "world"}
2 {int y, ...} rec2 = rec1
```

We declare a local variable `rec1` as open record and assign a value. The assigned record actually has more fields than the type, but it is allowed because the type is an open record. In line 2, we assign our record to another local variable `rec2`. Interestingly, the type of `rec2` is not at all the same as `rec1`. *Whiley* uses the type's structure for *subtyping* and considers `{int x, int y, ...}` to be a subtype of `{int y, ...}`. This is referred to as *width subtyping* in the literature.

To understand subtyping in *Whiley*, consider for each type the set of all values it describes. A type `T1` is subtype of `T2` if the set of all values with type `T1` is a subset of the values of `T2`. The type of `rec1` in the program above describes the set of all records that contain at least the fields `int x` and `int y`. The type of `rec2` describes the set of all records that contain at least the field `int y`. It imposes less restrictions to its values and therefore is a superset of the set described by `rec1`'s type.

Two types are the same if they are subtypes of each other. This is equivalent to stating that their sets of values are the same.

2.2.5. Union Types

Whiley's type system allows us to define types as unions of other types. A union contains all values that are present in any of its components. One possible usage is to use *Whiley*'s `null` type in a union. The type `null` has exactly one value, namely `null`. It is meant to express the absence of something. Used in a union, it can make the type somehow optional. Consider a function that takes an array and an element. It should find a position in that array where the given element occurs. The function can be specified and implemented as follows:

```
1  function indexOf(int[] a, int x)->(int|null r)
2  ensures r is null ==> no { j in 0..|a| | a[j] == x }
3  ensures r is int  ==> a[r] == x:
4    int i = 0
5    while (i < |a|)
6    where i >= 0 && i <= |a|
7    where no { j in 0..i | a[j] == x }:
8      if (a[i] == x):
9        return i
10     i = i + 1
11   return null
```

The return type is a union of `int` and `null`. The specification ensures that `null` is only returned when the given element is not contained in the array.

2.2.6. Type Declarations and Recursive Types

Structural types can be quite long and writing the type every time it is needed might clutter your program. *Type declarations* allow to define a name which can be used as alias for its declared type. Both the name and its structural type refer to the same type:

```
1  type Point is {int x, int y}
2
3  method main():
4    Point rec1 = {x: 8, y: 7}
5    {int y, int x} rec2 = rec1
```

Type declarations can carry *type invariants*. They constrain the declared type to only contain values that satisfy the given boolean expressions. For example, the following snippet specifies natural numbers and even numbers:

```
1  type nat  is (int x) where x >= 0
2  type even is (int x) where x % 2 == 0
```

Type declarations allow to specify *recursive types*. A recursive type directly or indirectly refers to itself within its definition. The following program defines and uses a recursive type `LinkedList`.

```
type LinkedList is null | {int head, LinkedList tail}

method main():
  LinkedList list =
      {head: 1, tail: {head: 2, tail: {head: 3, tail: null}}}
```

Although recursive types are not uncommon in other programming languages, they impose some challenges when it comes to structural typing. Consider the following type definitions:

```
type A is null | B
type B is {tail: C, head: int}
type C is {int head, A tail} | null
```

After having a close look on these types we can see that `A` and `C` are structurally the same as `LinkedList`. *Whiley* therefore considers `A`, `C` and `LinkedList` to be the same type. Furthermore, all three are a subtype of type `B`. But there is no obvious algorithm that detects these relations. A trivial approach is to continuously replace type names with their declarations and compare the resulting structure. But this will never yield the same result because unfolding does not terminate for these recursive structures. To solve this challenge, *Whiley* uses *type automata*.

Type Automata

We give here a concise overview of how types are internally represented in *Whiley*. A more extensive introduction can be found in [17, 19].

A type is represented using a *finite state automaton*. Each automaton state describes a type. The type represented by the automaton itself is the one of its start state, *Whiley* uses the term *root state*. Each state is annotated with a *kind* that denotes what class of type that state represents. Possible kinds are for example the primitive types, union, reference, record, function and method. Some states have outgoing transitions to other states, called *children*. A reference state has exactly one child, namely the state describing the referenced type. States for primitive types do not have any children. A union state has a child for every type that is part of the union. States can carry supplementary data. Functions and methods for example are annotated with the number of parameters. Records carry the names of their fields and a flag whether the record is open. Figure 2 illustrates the type automaton for the type `LinkedList` from the example above.

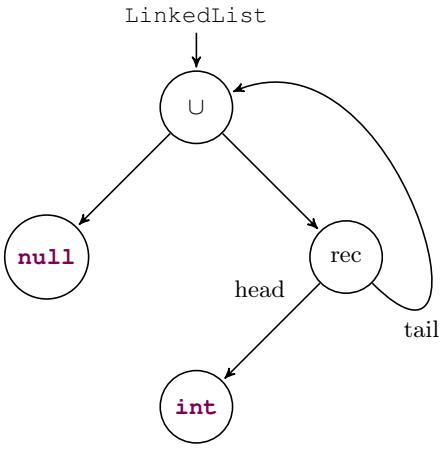


Figure 2: Automaton for `LinkedList`

Values in *Whiley* always have the form of a finite tree, i.e. they cannot be cyclic. The recursive type `LinkedList` does not limit the depth of its values, but *Whiley* does not

provide a way to produce a cyclic value or values of infinite depth. The type automata are designed such that they accept such a tree value if and only if it is contained in the automaton's type.

Whiley implements a single operation on type automata: given two automata A_1 and A_2 it is possible to check whether the intersection $A_1 \cap A_2$ is not empty, i.e. if there is a value that is accepted by both A_1 and A_2. It is further possible to optionally invert one or both automata. We use the notation $\overline{A}$ for the inversion of automaton A. Subtyping can be reduced to this intersection problem: T_1 is a subtype of T_2 if and only if $A_1 \cap \overline{A_2} = \emptyset$, i.e. the intersection is empty, where A_1 and A_2 are the type automata representing T_1 and T_2. In other words: the values in T_1 are a subset of the values in T_2.

The implementation of that intersection check is aware of recursive types. It recursively descends into the children of both given automata until a decision can be made. For recursive types, the automata will be cyclic. To break infinite recursion, the algorithm tracks which states have already been compared and assumes that there is an intersection for a pair of states if recursion reaches the same comparison again.

2.2.7. Flow Typing

The type of a variable in *Whiley* changes along the control flow that the program takes. This technique is called *flow typing*. Consider the following example:

```
1  function f(int|bool|null x)->(int r):
2    if x is null:
3      return 0
4
5    if x is bool:
6      if x:
7        r = 1
8      else:
9        r = 0
10
11   else:
12     r = x
13
14   return r
```

Function `f` accepts a parameter that can have any of the types **`int`**, **`bool`** or **`null`**. The type of `x` is the union of these types, as declared in the function signature. In line 2 we do a so-called *type test*. Each type test changes the current type of the tested variable. Inside the true branch `x` will have type **`null`**. *Whiley* sees that the program flow can take two paths: if the condition is true, then we will end up in line 3 and immediately return from our function. The other path implies that `x` is not **`null`**. *Whiley* calculates a type **`int|bool|null - null`** which simplifies to **`int|bool`**. This is the type of `x` from line 4 on.

The next type test ensures that `x` is typed as **`bool`** from line 6 to 9. Therefore we can use it as boolean expression in line 6. Finally, the else branch starting in line 11 knows

that x is not **bool**. The type is **int**|**bool** − **bool** which simplifies to **int**. Therefore we are allowed to assign x to the local variable r which is typed as **int**.

2.2.8. Heap-Allocated Memory and References

Whiley allows to allocate memory on the heap. The **new** operator takes an expression as argument. It allocates the amount of memory needed to store values of the given expression's type. The expression is then evaluated and the result stored in the allocated space. A reference to it is returned as result from the **new** operator. It cannot be used in functions, only methods can allocate memory.

There is no explicit deallocation operator in *Whiley*. Heap-allocated memory has to be either garbage collected at runtime or deallocated with operations inserted by the compiler. The *Whiley to Java compiler* relies on garbage collection provided by the JVM. The experimental *C* backend does not yet do deallocation of heap-allocated memory.

The following program gives some examples for memory allocations:

```
1  type Point is {int x, int y}
2
3  method m():
4    &int r1 = new 1
5    &bool r2 = new 3 < 4
6    Point p = {x: 5, y: 6}
7    &Point r3 = new p
```

To access the memory pointed to by a reference, the *dereference* operator ⋆ is used. We can read from and write to references. References provide a way to produce aliases. The following program shows some examples:

```
1  method main(whiley.lang.System.Console console):
2    &int r1 = new 1
3    &int r2 = r1
4    console.out.println(*r1) // 1
5    console.out.println(*r2) // 1
6    *r2 = 2
7    console.out.println(*r1) // 2
8    console.out.println(*r2) // 2
9    r2 = new 3
10   console.out.println(*r1) // 2
11   console.out.println(*r2) // 3
12
13   &(&int) rr = new r1
14   console.out.println(**rr) // 2
```

r2 is initially declared as alias for r1. The assignment in line 6 changes the value on the heap, such that both aliases will read the updated value. In line 9 we set the reference to something else. This does not affect the value stored on the heap, it only removes the alias we created initially. r2 now points to a separate portion of memory, holding the value **3**.

Line 13 and 14 illustrate that it is possible to create references to references. In order to read the actual value, two dereference operators are necessary.

2.3. Related Work

This section provides a rough overview of work related to ownership systems, automatic memory management without garbage collection, and verification.

2.3.1. Ownership

Work on ownership models dates back to 1998. Clarke et al. [9] presented a type system featuring ownership types to control aliasing.

Boyapati et al. [5] extend the Real-Time Specification for Java (RTSJ) and introduce a static type system using ownership types. It guarantees that the runtime checks ensuring memory safety in RTSJ will not fail, thus allowing to remove these checks.

In order to allow for parametric ownership types, Potanin et al. [21] developed an ownership system for Java 5, combining both generic types and generic ownership.

2.3.2. Region-Based Memory Management

Grossman et al. developed a programming language called *Cyclone* [11]. It is an extension of *C* that provides safe memory management. The unsafe `free` operator is removed. The memory is divided into several *regions*. Each region corresponds to a block in the program code. When the control flow leaves the block, then its region is freed automatically. Intraprocedural static analysis on region annotations in pointer types ensure memory safety.

Instead of relying on the programmer to annotate the program with regions, [7] proposes an interprocedural region analysis algorithm that reads a *Java* program and generates an equivalent program with region-based memory managed support.

These region-based techniques come with the disadvantage that only a whole region can be freed, deallocation of a single object inside the region is not permitted. Region-based memory management therefore utilizes more memory than garbage collection [7, 10]. Dhurjati et al. present an interprocedural analysis for *C* programs that manages memory in *pools* [10]. Explicit `free` commands are allowed if they do not violate the safety constraints. Otherwise the command will be removed, but the memory is still freed when the whole pool gets deallocated.

2.3.3. Alias Analysis

Aliasing is a key problem in memory management and verification. We say that a reference is an alias if there are multiple references to the same memory location. The referenced memory can be mutated through any of these aliases, which affects also the other references. Furthermore, deallocating the memory while there are still aliases to it leads to dangling pointers.

AliasJava [1] is an annotation system for Java. It extends the type system to explicitly annotate to what extend references can be aliased. These annotations enable programmers to better reason about aliasing. Furthermore, the authors present an algorithm that can automatically infer suitable annotations.

Brandauer et al. [6] recently proposed *Disjointness Domains* as a system for fine-grained alias control. All references belong to these domains that impose restrictions on aliasing within a single domain and among several domains.

2.3.4. Separation Logic

Separation logic [23] is an extension of *Hoare logic* [12]. It adds a way to reason about heap memory. We can make statements about disjointness of heap regions which basically is the absence of aliasing between both regions. Furthermore, we can reason about the actual values stored on the heap.

Tuch et al. [29] establish a formal model for a subset of the *C* programming language. They use the Isabelle/HOL theorem prover to reason about programs with pointers. A challenge is to model the real behavior of finite integers with overflow semantics.

3. Design

The purpose of this thesis is to introduce lifetimes to the *Whiley* programming language. In the following, we present the objective of introducing lifetimes and give a broad overview of our extensions regarding syntactical changes, a definition of lifetimes in *Whiley*, and a discussion about our design decisions. The actual implementation is covered in the next chapter.

Section 3.1 motivates the introduction of lifetimes, giving a major use case regarding memory management. In 3.2 we present our syntactical language extensions, providing formal definitions and example programs. We define lifetimes and their semantics for *Whiley* in 3.3. Section 3.4 provides a discussion about alternative approaches and the rationale behind our design decisions. We also elaborate the differences to the *Rust* programming language. Afterwards, chapter 4 provides a detailed view of the actual implementation for the concepts introduced here.

3.1. Motivation

The main objective of our lifetime extension for *Whiley* is to create a basis for improved memory management. Consider the following perfectly fine *Whiley* code:

```
1  method inc(&int x) -> &int:
2    return new ((*x) + 1)
3
4  method fourtytwo() -> &int:
5    &int i = new 41
6    i = inc(i)
7    return i
8
9  method main():
10   &int p = fourtytwo()
11   int i = *p
12   assert i == 42
```

Listing 3: *Whiley* example without lifetimes

We have three methods: `inc` is supposed to take a reference to a number and returns a reference to a number whose value is the input increased by one. Method `fourtytwo` should return a reference to a memory location holding the number `42` and `main` just checks that this is actually the case.

It might not be that useful to use references in this small example program. But a bigger real world example could actually benefit from passing references instead of

values: passing values can be expensive, if a lot of data has to be copied for each method call. Furthermore, it might me necessary for the program logic to have call-by-reference semantics.

Before we can discuss compiling this specific program we have to outline typical memory management done in compilers: memory can be allocated either on the stack or on the heap. Stack-allocated memory will be freed automatically as its containing stack frame gets deallocated when the method returns [27]. Data there cannot outlive the current method invocation. On the other hand, heap-allocated memory has to be explicitly freed by other means, e.g. garbage collection at runtime or manual memory management done by the programmer. To avoid this kind of overhead, compiler writers prefer to allocate memory on the stack whenever possible. This usually holds for local variables. Fixed-size objects that do not escape the method they are allocated in can also be placed on the stack [8].

Looking again at our example program from listing 3, we can see that the allocation from line 5 does not leave the scope of method `fourtytwo`, as the reference returned in line 7 is the one allocated by `inc` in line 2. It might therefore be a good idea to allocate it on the stack. However, consider the following modified implementation of `inc`:

```
1 method inc(&int x) -> &int:
2   *x = (*x) + 1
3   return *x
```

Listing 4: Modified implementation of `inc`

From a method-local point of view within `fourtytwo` we cannot know what method `inc` will be doing with the passed reference. While it is totally safe to do the initial allocation on the stack for the program in listing 3, it is not possible for the modified version in listing 4.

In a real-world example both methods could be part of different program packages. Only expensive inter-procedural static analysis can find out whether stack-allocation is possible. Utilizing these kind of analysis comes with the disadvantage that you cannot compile single units any more: Changing the implementation of `inc` from listing 3 to listing 4 does not change its signature, but turns compiled code for method `fourtytwo` unsafe if it uses stack-allocation.

Our system of lifetimes in *Whiley* solves that problem by encoding more behavior into method signatures. The first example can be annotated with lifetimes as follows:

```
1 method <a> inc(&a:int x) -> &*:int:
2   return new ((*x) + 1)
3
4 method fourtytwo() -> &*:int:
5   &this:int i = this:new 41
6   &*:int i2 = inc(i)
7   return i2
```

The signature of method `inc` states that it accepts one parameter which must be a reference to an integer, where the reference can have any lifetime `a`. Lifetime `a` is a

lifetime parameter for method `inc`. Furthermore, the return type is declared to be a reference with the *default lifetime* `*`.

Method `fourtytwo` can now do its allocation using the special lifetime **`this`** which is valid for the whole method body. Calling `inc` with type `&this:int` instantiates lifetime parameter `a` with **`this`**. The return type is `&*:int`. We store it to a separate variable that can then be returned from method `fourtytwo`.

When we try to annotate the modified version from listing 4 with lifetimes, we end up with **`method`** `<a> inc(&a:int x) -> &a:int` as method signature. We cannot declare the return type as `&*:int` as the return expression's type is `&a:int` and our system does not allow that one to be a subtype of `&*:int`. If we try to use a method `inc` with the modified signature, it no longer returns type `&*:int`. Instead, the lifetime parameter `a` gets substituted with the actual lifetime given as method argument, which in our case is **`this`**. Again, method `fourtytwo` cannot return a value of that type, as the declared return type is `&*:int`.

It is important to see that our extension allows to analyze both methods independently. The decision whether it is safe to allocate memory in `fourtytwo` on the stack instead of using the heap can be made only considering the own method body as well as all signatures of called methods. Their actual implementation does not matter.

3.2. Syntax

This section states syntax extensions we make to the *Whiley* language. Only updates to the grammar are given, the full specification is available in [18]. An introduction to the *Whiley* language is provided in section 2.2. The syntax is given in a form derived from the popular *Backus–Naur Form* [14]. Optional elements are enclosed in square brackets `[ ]`. Groups are enclosed in parentheses `( )`. Repetitions are denoted by a superscript star. Nonterminals are presented as plain words and terminals (tokens) are contained in a box.

The core extension is to annotate reference types with a concept of lifetimes:

```
ReferenceType ::= [&] Type
                | [&] Lifetime [:] Type

     Lifetime ::= [*] | [this] | Ident
```

The first option for `ReferenceType` is from the current *Whiley* specification. Type `&int` represents a reference to **`int`** and is equivalent to `&*:int`, where `*` explicitly specifies the *default lifetime*. It is meant to be used when the programmer cannot or does not want to specify any lifetime. The compiler then has to manage memory for that reference by some form of static analysis or garbage collection. Form `&this:int` uses the special lifetime name **`this`** that denotes the lifetime of the current method body. We introduce **`this`** as new keyword to the *Whiley* language. Finally, any declared lifetime identifier `a` can be used in the form `&a:int`.

Allocation of memory in *Whiley* is done using the **new** operator. We extend the syntax such that we can optionally define a lifetime for the allocated portion of memory:

```
NewExpr ::= new Expr
          | Lifetime : new Expr
```

While **new 42** and ⋆**:new 42** allocate an integer with default lifetime, the new options are **this:new 42** and a**:new 42**, which allocate memory for the annotated lifetime.

To declare new lifetime names we introduce *named blocks*. They can occur in method bodies and may be nested with other blocks:

```
1  method m():
2    int x = 1
3    myblock: // declares a new lifetime 'myblock'
4      &myblock:int y = new x
5      int i = 0
6      while (i < 5):
7        i = i + 1
8        mynestedblock:
9          &mynestedblock:int z = new i
10         // ...
11   // here neither 'myblock' nor 'mynestedblock' are valid
```

Formally, we define an additional statement `NamedBlock`:

```
NamedBlock^ℓ ::= Ident : Block^γ    (where ℓ < γ)
```

A `Block` is a list of statements. The annotated condition $\ell < \gamma$ states that all statements whose indentation is greater than the `Ident`'s indentation are considered as part of the `Block`, i.e. line 4 to 10 in the example program above belong to the `Block` of `myblock`. Keyword `this` is not allowed as identifier for lifetime names in named blocks.

Furthermore, we extend the syntax of method declarations to allow for specifying lifetime parameters. Function declarations do not need to be extended, as functions are pure and must not make use of references. Inspired by the syntax for *Java* generic type parameters, we declare all lifetime names used in method parameters and return type into angle brackets before the method name:

```
1  method <a, b> m1(&a:int x, &b:int y) -> &a:int:
2  method <a> m2(&{&a:int val} container) -> &a:int:
```

The relevant update to the grammar is as follows:

```
        MethodDecl ::= method [ < LifetimeParameters > ]
                       Ident ( Parameters ) ...

LifetimeParameters ::= [ Ident ( , Ident )* ]
```

The omitted part contains elements not affected by our lifetime extension, namely return type, **requires** and **ensures** declarations, and the method body. As seen before in named blocks, the special lifetime **this** is considered to be a keyword and must not appear in `LifetimeParameters`.

Our next syntax extension affects method invocations. It is needed to pass lifetime names into the `LifetimeParameters` that we just defined. In line 2 of the following program we call a method `min` with two lifetime parameters. Its actual implementation is not relevant for this extension, but given as an example:

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

The updated grammar for method invocation is:

```
      InvokeExpr ::= Name [ < LifetimeArgsList > ]
                         ( ArgsList )

LifetimeArgsList ::= [ Lifetime ( , Lifetime )* ]
```

Readers familiar with the *Java* syntax for invoking methods with generic type parameters might be concerned about the defined syntax. A detailed discussion is given in section 4.1.

As *Whiley* combines functional and imperative features, we can use methods as values. To do so, there is a syntactical representation for method types. Furthermore, methods can be declared as an anonymous method, using so-called *Lambda expressions*. We have to update both concepts to allow for specifying lifetime parameters.

The following program utilizes our updated syntax. It declares a type alias `mymethod` for methods that take two references of possibly different lifetimes and yields a reference with default lifetime. The lambda expression in line 4 represents such a method. We call it in line 5, using the already introduced syntax for specifying lifetime arguments.

```
1 type mymethod is method<a, b>(&a:int, &b:int)->(&*:int)
2
3 method main():
4   mymethod m = &<a, b>(&a:int x, &b:int y -> new (*x) + (*y))
5   &int p = m<this, *>(this:new 1, new 2)
6   assert (*p) == 3
```

The updated grammar also introduces a concept named *context lifetimes*. These are lifetimes from the enclosing method that can be used inside the lambda expression. The example above does not use that concept. A detailed explanation will be given in section 3.4.5. The updated grammar is as follows:

```
      MethodType ::= method [ [ ContextLifetimes ] ]
                     [ < LifetimeParameters > ]
                     ParameterTypes [ -> ParameterTypes ]

ContextLifetimes ::= [ ContextLifetime ( , ContextLifetime )* ]

 ContextLifetime ::= this | Ident

  ParameterTypes ::= ( [ Type ( , Type )* ] )

      LambdaExpr ::= & [ [ ContextLifetimes ] ]
                     [ < LifetimeParameters > ] (
                     [ Type Ident ( , Type Ident )* ]
                     -> Expr )
```

3.3. Lifetimes

A *lifetime* in our extension to *Whiley* describes a region in the source code. Possible regions are method bodies and named blocks. Each lifetime has a name. The lifetime for method bodies is named **`this`**. The lifetime of named blocks have the block's name. Furthermore, the special lifetime `*` persists for the entire program.

Each reference type is annotated with a lifetime. It states when the memory pointed to by that reference should be alive. More precisely, the following invariant must be maintained by the compiler:

Invariant 1. *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 1. *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*

We explain the invariant using the example program from listing 5. Line 2 declares a local variable `p` without initializing it with a value. This is where the condition *initialized reference* of invariant 1 comes into play. Although the local variable has already been declared as reference, it does not yet hold a value, i.e. does not point to anything. It is *uninitialized*. The *Whiley* compiler already ensures *definite assignment*, i.e. we cannot read a variable `v` if there might be a control flow path to that read operation without prior assignment to `v` [18, pages 71, 82].

```
1 method main():
2   &this:int p
3   int x = 1
4   p = this:new x
5   myblock:
6     &myblock:int q = p
7     &myblock:int r = myblock:new 2
8     p = r // ilegal
9   x = *p
```

Listing 5: Example program to illustrate invariant 1

We initialize p in line 4. The declared type of p is `&this:int`. Its lifetime is **this**, which denotes the body of method main. Invariant 1 therefore demands that the memory pointed to by p must not be freed before the control flow leaves method main. The **new** operator is annotated with a lifetime. It tells the compiler that the allocated memory must be alive until the control flow leaves method main such that it matches the type's lifetime.

In line 6, we assign p to another reference q. The type of q is `&myblock:int`. To satisfy invariant 1, the memory pointed to by the assigned reference must not be freed before control flow leaves myblock. On the other hand, invariant 1 ensures that the memory pointed to by p will not be freed before the control flow leaves method main. This holds regardless of what has been assigned to p, because its declared type is `&this:int`. Memory that will not be freed before the control flow leaves main will in particular not be freed before the control flow leaves myblock, because the first implies the latter. We can therefore allow this assignment.

There is an assignment in line 8 where we attempt to assign r to p. Using the same argumentation, we know that the memory pointed to by r will not be freed before the control flow leaves myblock. But p must point to memory that cannot be freed before the control flow leaves method main. The implication does not hold in this case. We cannot allow this assignment because it would break invariant 1 and the safety of other commands might rely on it. In fact, we dereference p in line 9. It points to the memory assigned in line 7. The **new** operator was annotated with lifetime myblock, telling the compiler that this specific allocation can be freed when control flow leaves myblock. Allowing the assignment in line 8 could therefore lead to dereferencing freed memory, which is unsafe.

We can distinguish the allowed from the forbidden assignment by looking at the involved lifetimes. In the first case, the lifetime of the assigned value is longer than demanded by the declared type of the variable that we assigned to. We say that lifetime **this** *outlives* lifetime myblock.

Definition 2. *A lifetime* a *outlives lifetime* b *if the region described by* b *is fully contained in the region described by* a*.*

Note that definition 2 is reflexive, i.e. a lifetime always outlives itself. Furthermore, the

lifetime $\star$ outlives all lifetimes.

3.3.1. Subtyping

Whether or not assigning a value to a variable is allowed depends on their types. *Whiley* allows to assign a value `x` to a variable `v` if and only if the type of `x` is a *subtype* of `v`'s declared type [18, p. 40]. We therefore need to extend subtyping for references to compare their lifetimes.

Definition 3. *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*

The first condition is part of our extension, the second condition is *Whiley*'s existing rule for subtyping of references. We cannot allow `A` to only be a subtype of `B` here because the following snippet breaks memory safety:[4]

```
1 method m(&any x):
2   *x = 5 // allowed because x points to 'any' type
3
4 method main():
5   &bool p = new true
6   m(p) // allowed if &bool is a subtype of &any
7   bool b = *p // saves '5' into a bool variable
```

Types in *Whiley* describe sets of values. Type `T1` shall be a subtype of `T2` if the set of values in `T1` is a subset of values in `T2` [18, p. 37].

Our extension complies with the intended semantics, though the set analogy is complicated for references. Assume `A` and `B` describe the same type and `a` outlives `b`. Let `p` be a reference of type `&a:A`. Using invariant 1, we know that `p` points to memory that will be alive at least until the control flow leaves `a`. Since `a` outlives `b`, the memory will in particular be alive until the control flow leaves `b`. Reference `p` is therefore also in type `&b:B`.

3.3.2. Lifetime Parameters

Methods can accept references as their parameters. A method can also return references. As with every reference type, parameter and return types need a lifetime. Furthermore, it might be necessary to relate lifetimes of parameter and return types. *Lifetime parameters* provide a way to declare lifetime names that describe lifetimes originating from the method's caller. The caller will pass actual *lifetime arguments* for these parameters.

We do a *lifetime substitution* for a method call: Lifetime parameters are mapped to the provided lifetime arguments. Every occurring lifetime in the method's parameter and

[4] *Whiley* actually used to allow it, but the bug was found and resolved while working on this thesis. See https://github.com/Whiley/WhileyCompiler/issues/583

return types is substituted by its appropriate lifetime argument. Consider the following example:

```
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 = new 1
9    &this:int y = m<this>(x)
```

The program above declares a method `m` that accepts and returns type `&a:int`. Lifetime `a` is a lifetime parameter. The method therefore returns a reference of the same lifetime than given as parameter. When calling `m`, we give `this` as lifetime argument. Lifetime `a` in the parameter and return type is therefore substituted with `this`, such that the method invocation returns `&this:int` where `this` refers to `main`'s method body.

3.3.3. Context Lifetimes

Anonymous methods, so-called *lambda methods*, can access local variables from the enclosing method. This also holds for references. If such a reference is dereferenced inside the lambda method, then we have to ensure that the referenced memory location is still alive when the lambda method is executed.

To solve this problem, we extend lambda methods and method types with a concept called *context lifetimes*. These are lifetimes from the enclosing method that can be dereferenced inside a lambda method. A detailed discussion on the problem and our solution is given in section 3.4.5.

3.4. Discussion

This section gives a discussion on our design decisions provided in the previous sections. We present possible alternatives and the reasons why they have been discarded.

Our extension to *Whiley* has been developed as part of this Master's Thesis. The extend of work for this thesis is limited to six months, done by a single student. All implementation and the scientific elaboration had to be done with these limited resources. This imposes some constraints on the extent of changes that can be made.

3.4.1. Backwards-Compatibility

One goal for our work was to maintain backwards compatibility. *Whiley* is still a developing language, such that backwards-incompatible changes are basically possible, but being able to use all existing test cases to see whether our changes break intended behavior and semantics of *Whiley* was a big benefit.

However, some changes are not fully backwards-compatible: The introduction of `this` as a keyword breaks every program that uses it as identifier. Indeed, some test cases and also some components of *Whiley*'s standard library had to be adopted for that change. But a simple replacement of affected identifiers resolves the problem.

We decided against a requirement to explicitly annotate every reference with a lifetime. This allows to still compile programs that do not make use of our extension. The implicit default lifetime for references without explicit annotation therefore had to be a lifetime that does not impose additional constraints to existing programs. References can be passed between methods, so the default lifetime should outlive all methods. The legacy notation `&int` can therefore be regarded as syntactic sugar for `&*:int`.

One possibility for future optimization is to infer shorter lifetimes if no explicit lifetimes are given. As an example, a method whose parameter and return types do not contain any reference types cannot leak references to the caller. All allocations and references in that method can therefore be assumed to be annotated with lifetime `this` instead of `*`. But this is not possible for the general case where methods take and return references. Inferring a shorter lifetime for these programs would need expensive inter-procedural static analysis, which is out of scope for this thesis.

3.4.2. Mutability and Move Semantics

In *Whiley*, functions have call-by-value semantics. We can describe call-by-value semantics with two properties:

1. the caller will not observe any modifications of passed values done within the callee
2. the callee will not observe any concurrent modifications of passed values done by other parts of the program

The second point is trivial as there is not yet a concept for concurrency in *Whiley*. The compiler is responsible to ensure the first point.

Rust distinguishes between mutable and immutable values and references. A primary use for it is to classify borrowed references: there can either be multiple immutable borrowings or only one mutable borrowing at any time. Call-by-reference semantics with immutable references is basically the same as call-by-value semantics: the callee cannot make any changes to the referenced value. This ensures the first point described above. Furthermore, the value is borrowed immutable for the time that the callee is executed. The borrow checker in *Rust* ensures that the value cannot be mutated during that time, so also the second point holds.

To provide a function with read-only access to a value, we can pass it an immutable reference in *Rust*. This is equivalent to calling the function with the value itself in *Whiley*. When we want to provide the callee also with write access, we have different options in *Rust*: we can pass the value itself. Ownership will be transferred to the callee and the caller's former variable binding cannot be used anymore. Alternatively, we can pass a mutable reference to that value. Then the caller remains owner of the value. In *Whiley*, the only option is to pass a reference. To modify the referenced value, the callee uses the

dereference assignment. The callee therefore needs to be a method, because dereference operations are not allowed in pure functions.

Whiley has a focus on verification. But verification with references is difficult, as there might be numerous aliases and an assignment can affect multiple variables. The current verifier in *Whiley* cannot handle references.

```
method main():
  &int x = new 1
  assume (*x) == 1
  *x = 2
  assume (*x) == 2
```

Neither of the two verification conditions generated by the **assume** statements passes verification. It therefore is a good practice to use values instead of references when possible.

Whiley does not have a concept of immutable values or references. We decided against introducing them for two reasons. First of all, it is not possible to add the invariant of having only one mutable or several immutable references without breaking backwards compatibility. Existing programs can create multiple aliases and mutate the value via all of these references. That invariant is particularly useful in a concurrent setting, as it prevents data races. But *Whiley* does not yet have concurrency.

Another point to consider are function or method parameters: in *Rust*, a function with two references as parameters where at least one of them is mutable can always assume that they do not point to the same memory. This is due to the fact that the caller cannot borrow the value mutable and immutable at the same time. If a method in *Whiley* relies on the fact that parameters are not aliases of each other, we can simply add an appropriate precondition using **requires**.

Call-by-value as in *Whiley* and call-by-reference semantics can differ in their performance: for a call-by-reference semantics we only need to pass an address to the callee. For call-by-value, the implementation in general needs to generate a deep-copy of the passed value in order to allow the callee to modify an independent copy. But if there is no such modification the compiler can optimize the code by using a call-by-reference semantics.

3.4.3. References to Local Variables

Rust allows to create references to local variables. We cannot allow that in *Whiley*, as it would easily break type safety. This is due to *flow typing* as described in section 2.2.7. Consider the following program:

```
int|null x = 5
&int y = new 42;
if x is int:
  y = &x // invalid syntax!
x = null
int z = *y
```

Assume line 4 would store a reference to `x` into `y`, similar to *Rust*'s borrowed references. Due to flow typing, the static type of `x` in line 4 is **`int`**. Therefore the assignment in line 4 is fine. In line 5, `x` has again type **`int|null`**. Therefore we should be able to assign **`null`** to it. In line 6, we dereference `y`. It expects type `&`**`int`** and therefore should yield an integer. But the actual value will be **`null`**!

Flow typing relies on the fact that there are no aliases for local variables. Allowing to create references to local variables would introduce these aliases. As there is no easy solution to keep flow typing in a backwards-compatible way while handling aliases, our extension does not allow creating references to local variables.

3.4.4. Pointer Types

As discussed in section 3.4.2, references are a challenge for verification. While values in *Whiley* cannot have aliases we always have to consider side-effects when calling a method with references or doing dereference assignments. *Rust* offers several different type of values and pointers: First of all, there is the owned value itself. It can be mutable or immutable. Furthermore, there are borrowed references, `Box`, `Rc` and `Arc`. Each of them comes with different advantages and benefits. The choice of the right type suitable for the specific use-case is important. Changing your decision later on involves refactoring all types, e.g. in function signatures or explicitly typed variable bindings. In its current state, *Whiley* should remain as simple as possible. This is also the reason why the former support for floating point numbers has been removed during recent development[5]. *Rust* offers these different types to be as performant as possible: the types provide different trade-offs between imposed restrictions and runtime overhead. An `Arc<Box<T>>` can always be used instead of `&T`, but the additional overhead might not be necessary.

As described above, references to local variables are not a feasible option for our extension. Instead, we need to have all references memory to be allocated using the **`new`** operator. The compiler needs to decide whether the allocated memory can be placed on the stack and therefore use a similar performance as local variables. A key input to that decision are lifetimes. The *Whiley* reference type `&T` is similar to *Rust*'s type `Rc<Box<T>>`. Our extension leaves it up to the compiler to optimize that type to an equivalent of `&T`. The *Whiley* language itself stays simple, as the decision where to allocate the memory is delegated to the compiler. The programmer can influence that process by annotating lifetimes, telling the compiler how long an allocation can be accessed. Without lifetime annotations, the backwards-compatible default will be assumed.

3.4.5. Lambda Expressions

Lambda expressions allow to declare anonymous functions and methods. They differ from normal functions and methods in such a way that they are declared in a context. We can access variables that are declared outside from inside the lambda expression:

[5] http://whiley.org/2016/01/29/whiley-v0-3-38-released/

```
1 method main():
2   int x = 5
3   function(int)->(int) f = &(int y -> x + y)
4   int z = f(3)
5   assert z == 8
```

Even though variable `x` is declared in method `main`, we can still access it in the anonymous function declared in line 3. We can assume that these anonymous functions store a copy of all accessed variables that are declared in the outer context. For actual values this works well. When it comes to references, then we have to ensure that the memory pointed to is still alive. This is trivial when using garbage collection, but consider the following program using our extension:

```
1 method generateLambda() -> method(int)->(int):
2   &this:int x = this:new 5
3   method(int)->(int) f = &(int y -> (*x) + y)
4   return f
5
6 method main():
7   method(int)->(int) f = generateLambda()
8   int z = f(3)
```

Similar to our first example, we have an anonymous method declared in line 3 that accesses the variable `x` declared in line 2. It is important to realize that `x` does not hold an integer. It holds a reference, i.e. the address of a memory location holding that integer. The lambda expression therefore stores a copy of that address. The address is dereferenced inside the lambda expression.

Our lifetime extension suggests that `x` should be allocated on the stack, because its annotated lifetime is **`this`**. That implies that the memory holding our value **`5`** is freed when `generateLambda` returns. But the actual dereference operation to that memory happens afterwards, due to the call of method `f` in line 8. We dereference freed memory!

There are two general approaches to prevent that problem: first of all we can forbid to use the dereference operator inside lambda expressions, or at least forbid dereferencing of variables that are part of the lambda's context. The second approach is to prevent calling lambda expressions after the end of a lifetime that is dereferenced inside. We selected the second choice, because it is less restrictive.

The lifetimes of references from the lambda's context that are dereferenced inside the lambda expression are called *context lifetimes*. In the example above, **`this`** is a context lifetime for the lambda expression declared in line 3 because `x` is dereferenced inside the lambda expression but declared outside and its lifetime is **`this`**. The programmer has to specify a set of context lifetimes that he wants to dereference inside the lambda expression: `&[this](int y -> (*x) + y)`. Furthermore, the context lifetimes are part of the method's type: `method[this](int)->(int)`. These context lifetimes are considered to be using occurrences, i.e. the specified lifetime names need to be declared before and still be in scope. This is what ensures effectiveness of our approach. In the program above, we can add the context lifetime to both the lambda expression itself and

its type in line 3. But we cannot add it to the return type of `generateLambda`, because lifetime **`this`** is only valid in the method body and not in return types.

To call a lambda method we need to access its value. The value can be stored in a local variable, in an array or in a record. In all cases, its declaration will be in scope. The declaration states the type, containing the method's context lifetimes. This transitively ensures that the context lifetimes are still in scope when calling a lambda function. therefore also all context lifetimes will be in scope.

When type-checking a dereference operation inside a lambda expression, the compiler now has to extract the lifetime of the given reference. It has to be part of the context lifetimes. Alternatively, the context lifetimes may also contain a lifetime that outlives the reference's lifetime.

Context lifetimes have to be considered for subtyping. First of all, there is no order among context lifetimes. They can be treated as sets, i.e. all of the three types **`method`**`[a,b]()->()`, **`method`**`[a,b]()->()`, and **`method`**`[a,b,a]()->()` are equivalent. If a method is allowed to access some context lifetimes but actually does make use of it, then the execution is still save. Subtypes therefore can have a subset of the supertype's context lifetimes. Consider the case that the method type declares `b` as context lifetime. That imposes a restriction that the method can only be executed while `b` is still in scope. If there is another lifetime `a` that outlives `b`, then it is also safe to dereference references with lifetime `a`. We therefore define the following restriction for subtyping among references: every context lifetime in the subtype must outlive a context lifetime in the supertype.

4. Implementation

This chapter provides a detailed view on the different parts of the implementation. A broad overview of our design and a discussion among alternative solutions has been given in chapter 3.

Whiley is an open source project. The compiler is written in *Java* and its source code can be found at `https://github.com/Whiley/WhileyCompiler`. The lifetime extension covered by this thesis has already been merged into the `develop` branch. A copy of that `git` repository is also stored on the digital medium attached to this thesis.

The implementation of our lifetime extension comprises a total of 2388 added and 464 removed lines of *Java* source code, excluding new test cases which add another 549 lines. Furthermore, the author of this thesis contributed some independent changes to *Whiley*, mainly in order to fix bugs discovered while developing the lifetime extension. These changes cover additional 665 added and 96 removed[6] lines.

Section 4.1 presents how we change the parser to accept the new syntax. In section 4.2, we extend the type checker to respect lifetimes. Method calls and lifetime substitution are presented separately in section 4.3. Section 4.4 shows the changes to the intermediate language.

4.1. Lexing and Parsing

The *Whiley* parser can be found in package `wyc.io` and consists of two parts: the *lexer*, implemented in class `WhileyFileLexer`, transforms the source file into a *token stream*. Class `WhileyFileParser` is the actual parser which generates an *abstract syntax tree* (AST) from these tokens.

We had to adjust these components to support the syntax changes presented in section 3.2.

4.1.1. Abstract Syntax Tree

The *Java* classes for the AST nodes can be found in package `wyc.lang`. The following additions have been made:

- **Declared Method**: a list of lifetime parameters was added to the AST node representing a method.

[6] This number excludes the 5725 removed lines from `https://github.com/Whiley/WhileyCompiler/pull/582`. That change removed an explicit list of JUnit test cases and added a parameterized test that runs for all available test programs.

- **Lambda Expression**: a list of lifetime parameters and a set of context lifetimes was added to the AST node representing a lambda expression.
- **Method Invocation Expression**: a list of lifetime arguments was added to the AST node representing a method invocation expression and the return type now reflects lifetime substitution as presented in section 4.3.1.
- **Reference Type**: a lifetime name was added to the AST node representing a reference type.
- **Method Type**: a list of lifetime parameters and a set of context lifetimes was added to the AST node representing a method type.
- **Named Block**: a new AST node for *named blocks* was added. It contains the block's name and the list of statements within that block.

4.1.2. Lexer

The only change to the lexer is to add `this` as a new keyword. That change is backwards-incompatible, because it affects all programs that use `this` as an identifier. Some of the existing test cases were affected and had to be updated to use another name instead of `this`. This is due to the historic actor model in *Whiley* that has been removed a long time ago. It is unlikely that many programs are affected by our change.

4.1.3. Parser

Changes to `WhileyFileParser` are more complex. The parser is structured in several methods that are responsible to *consume* tokens for a specific AST node, e.g. an expression, a method declaration or a type. To chose the right method, a *lookahead* of one or more next tokens is used.

Enclosing Scope

While parsing a function or method, some contextual information is tracked in an object of a class called `EnclosingScope`. The main purpose of it is to store declared variable names. We extend that class to also store the declared lifetime names.

Whiley already expects variable names to be unique in a scope, i.e. it is not allowed to declare a local variable where the name is already used for another variable that is still in scope. We extend this constraint for lifetimes. Particularly, local variables and lifetimes must have different names. We use this fact later for method invocations.

We add helper methods in `EnclosingScope` to declare new variables or lifetimes. They check whether the name is already in use and throw an appropriate exception if that is the case.

Methods

The main entry point for parsing *Whiley* functions and methods is the parser method `parseFunctionOrMethodDeclaration`. We extend it to also parse lifetime parameters for method declarations. They are added to the `EnclosingScope` such that the lifetime names can be used in a method's parameter and return types and within the method body. Lifetime **`this`**, denoting the body of a method, is added just before parsing the method body, thus ensuring that it cannot be used in parameter or return types.

Headless Statements

Some statements are not identifiable with a dedicated keyword. They are called *headless statements* and handled in method `parseHeadlessStatement`. *Whiley* used to have the following headless statements: assignments, invocations, and variable declarations. The latter one starts with a type. But there are token sequences that can be parsed as a type or expression, for example an identifier can refer to a type alias or a local variable. The sequence `&foo` can denote the type *reference to foo* or an address expression, where the value is the method with name `foo`.

To disambiguate these cases, the compiler attempts to parse a type and checks whether it cannot be parsed as expression. In case of such a *definite type*, the headless statement is a variable declaration. The check is done in method `mustParseAsType`.

We extend that check to be aware of lifetimes in reference types. The sequences `&*:foo`, `&this:foo` and `&a:foo` cannot be parsed as expression. The legacy form `&foo` is syntactic sugar for `&*:foo`. But the check needs to know whether the default lifetime `*` was explicitly given or omitted. We therefore also extend the AST node for reference types to store such a flag.

Named Blocks

The newly introduced *named blocks* are headless statements. We therefore extend `parseHeadlessStatement`. A named block starts with an identifier followed by a colon and a line break.

```
1 myblock:
2   // statements inside the block
```

When parsing a named block, an independent clone of `EnclosingScope` is created. We add the identifier as a lifetime name and parse the statements within the named block. All following statements with greater indent are considered to be part of the named block. The logic for parsing that kind of indented blocks already existed, e.g. to parse the body of **`while`** loops. We reused that logic for named blocks.

Method Invocations

Method invocation expressions can now have optional lifetime arguments. Our syntax for calling a method with explicit lifetime arguments is `m<a, b>(x, y)`. But the part

`m<a` could also be an expression, where both `m` and `a` are numeric local variables. To disambiguate, we enforce that lifetime names and local variables are disjoint. If `a` is a lifetime, then it has to be an invocation. The new disambiguation logic has been added in `parseAccessExpression` and `parseTermExpression`.

Reference Type and Method Type

Reference types are parsed in `parseReferenceType` and method types in `parseFunctionOrMethodType`. We extend both methods to reflect our syntax extension. Reference types can be annotated with a lifetime, e.g. `&this:int`. Method types now include context lifetimes and lifetime parameters, e.g. `method[this]<a,b>(&a:int,&b:int)->(&this:int)`.

The parser checks that the lifetime name in reference types and the context lifetimes for method types are already declared. Declared lifetimes are stored in the `EnclosingScope` object. We therefore need it as parameter to both methods and all methods that call them either directly or transitively. Most of the parser methods already have that parameter, but some methods and their invocations had to be adjusted to add it.

Allocation Expression

Memory allocation is done using the **`new`** expression. Our extension allows to specify a lifetime for the allocated portion of memory: `this:new 1`. Allocation expressions are parsed by method `parseNewExpression`. We extend it to parse the optional lifetime.

Allocation expressions can now start with different tokens: either **`new`** if the lifetime is omitted, or any lifetime (identifier, **`this`**, or `*`). There is a lookahead logic in `parseTermExpression` that decides when to call `parseNewExpression`. We extend that logic to also consider a sequence of three tokens: lifetime, colon, **`new`**.

Lambda Expressions

Lambda expressions are parsed in method `parseLambdaExpression`. We extend that method to parse optional context lifetimes and lifetime parameters.

4.2. Type Checking

The next phase after parsing a *Whiley* program is type checking. The type checker ensures that all functions, methods, statements and expressions are well-typed. A key part of it is *subtyping*.

All types in *Whiley* are represented as automata, as described in section 2.2.6. The implementation for subtyping can be found in class `wyil.util.type.SubtypeOperator`. The `SubtypeOperator` object stores the type automata of the two types that should be tested. Method `isSubtype` translates the subtype query into an intersection test: `T1` is a subtype of `T2` if and only if the intersection of `T1` with the complement of `T2` is non-empty.

An intersection can either return a result immediately or depend on one or more recursive intersection queries. The intersection of **int** and **bool** is empty. Types `&a` and `&b` intersect if and only if types `a` and `b` intersect. Recursive queries use the same type automata, but a different automaton state as entry point. The `isIntersection` method therefore takes the automaton state index for both types as parameter. Furthermore, it takes for both types a flag denoting whether we consider the complement instead of the type itself. In the following, we write `!T` to denote the complement of type `T`.

For recursive types, recursive intersection tests can incur an infinite recursion. To mitigate that problem, the intersection test is split into two methods, `isIntersection` and `isIntersectionInner`. The first one contains logic to break recursion. When recursion leads to the same test again, then the intersection is assumed to be non-empty. Method `isIntersectionInner` contains the actual intersection logic that needs to be extended in order to introduce lifetimes.

4.2.1. Lifetime Relation

The new subtyping rules presented in section 3.3.1 use the *outlives* relation among lifetimes. We introduce a new class `wyil.util.type.LifetimeRelation` that stores this relation. Entries are added while recursing the AST in `wyc.builder.FlowTypeChecker`. An instance of `LifetimeRelation` is passed to `SubtypeOperator` in order to check the *outlives* constraints in subtyping rules.

The *outlives* relation is a *partial order*. Lifetimes declared in named blocks are ordered such that the lifetimes of outer blocks outlive the lifetimes of inner blocks. We store the currently declared named blocks in a `Stack`. The type checker treats lifetime **this** like a named block: method bodies are assumed to be enclosed in a block with name **this** such that it is always the outermost block. Furthermore, we store a `Set` of declared lifetime parameters. They outlive all named blocks, because the lifetimes are declared by the current method's caller. But there is no order within lifetime parameters.

4.2.2. Reference Types

We need to extend the intersection logic for reference types to respect the annotated lifetimes. One or both types can be inverted for the test, such that we have to consider four cases:

1. `&a:A` ∩ `!(&b:B)`
   This intersection is empty if and only if `&a:A` is a subtype of `&b:B`. As described in section 3.3.1, this is the case if
   - lifetime `a` outlives lifetime `b` and
   - `A` and `B` describe the same type, i.e. the intersections `A` ∩ `!B` and `B` ∩ `!A` are empty.

   The first condition can be checked using the newly introduced `LifetimeRelation`. Remember that our definition of *outlives* is reflexive. The second one incurs two recursive intersection queries.

2. `!(&a:A) ∩ &b:B`
   This case is symmetric to the first one.

3. `&a:A ∩ &b:B`
   We have to check whether there is a value in both `&a:A` and `&b:B`. We claim that this is equivalent to intersecting `&A` and `&B`, i.e. using the lifeitme $\star$ instead of `a` and `b`. If there is a value in both `&a:A` and `&b:B`, then we can take that value and change its lifetime to $\star$, which outlives all lifetimes. Due to the subtyping rule for references, the value will then be in both `&A` and `&B`. For the other direction, assume that there is a value in both `&A` and `&B`. Then it is also in both `&a:A` and `&b:B` because the lifetime $\star$ outlives `a` and `b`.

4. `!(&a:A) ∩ !(&b:B)` The complement of a reference type contains values of types that are not references, e.g. **`true`** and **`0`**. These values are in the intersection, so it is non-empty regardless of `a`, `b`, `A` and `B`.

4.2.3. Method Types

Our extension adds lifetime parameters and context lifetimes to method types. These concepts have been introduced in sections 3.3.2 and 3.3.3. They have to be considered for intersection tests.

Lifetime parameters are an ordered list of lifetime names that may appear in the methods parameter and return types. The actual name of these lifetime parameters does not matter for the method's type, they are only placeholders. Consider the following method types:

1. **`method`**`<a,b>(&a:`**`int`**`)->(&b:`**`int`**`)`

2. **`method`**`<b,a>(&b:`**`int`**`)->(&a:`**`int`**`)`

3. **`method`**`<a,b>(&b:`**`int`**`)->(&a:`**`int`**`)`

Our intuition suggests that the first two types are the same, because we only need to rename the lifetime parameters. But the third type is different: it cannot be transformed into the first two types only by renaming lifetime parameters.

In section 4.3.2 we present an algorithm to substitute the actual lifetime names with abstract placeholders. The first lifetime parameter will be named **`$1`**, the second one **`$2`** and so forth. The types above will therefore be transformed to:

1. **`method`**`<`**`$1`**`,`**`$2`**`>(&`**`$1`**`:`**`int`**`)->(&`**`$2`**`:`**`int`**`)`

2. **`method`**`<`**`$1`**`,`**`$2`**`>(&`**`$1`**`:`**`int`**`)->(&`**`$2`**`:`**`int`**`)`

3. **`method`**`<`**`$1`**`,`**`$2`**`>(&`**`$2`**`:`**`int`**`)->(&`**`$1`**`:`**`int`**`)`

Using this substitution and ignoring the symmetric and trivial case, we have to adopt the following intersections:

1. `method1` ∩ `!method2`
   This intersection is empty if and only if `method1` is a subtype of `method2`. As described in section 3.4.5, every context lifetime in the subtype must outlive a context lifetime in the supertype. Furthermore, the number of lifetime parameters must be the same in both method types.

   The remaining condition is that parameter types are contra-variant and return types co-variant. This is not changed by our lifetime extension, except for the substitution of lifetime parameters.

2. `method1` ∩ `method2`
   We have to check whether there is a method that is in both method types. For such a method to exist, the number of lifetime parameters must match. But we do not need to consider context lifetimes here: an intersecting method can be chosen with an empty set of context lifetimes. Then it is in method types with arbitrary context lifetimes, provided that the remaining conditions are met.

   The remaining condition is that return types intersect. This is not changed by our extension, but we need to apply the lifetime substitution.

4.3. Lifetime Substitution and Method Lookup

So far we have discussed two different scenarios for lifetime substitution:

1. Section 3.3.2 describes substitution of lifetime parameters to calculate the actual return type of method invocations. Lifetime parameters have to be substituted with the appropriate lifetime arguments.

2. In section 4.2.3, we discussed subtyping and intersection of method types. In order to compare two method types we need to ensure that their lifetime parameters have the same names. We therefore substitute them with ordered placeholders **`$1`**, **`$2`**, and so forth.

Even though both use cases involve some form of substitution, they have different requirements: For the first case, we need to calculate the resulting type because it is the type that will be assigned to an invocation expression. This is not necessary for the second case. Here we only need the boolean result, whether or not the input types intersect. But the intersection algorithm has to be aware of recursive types and must avoid an infinite recursion.

We therefore discuss appropriate substitution algorithms separately. Section 4.3.1 presents a solution for the first case, i.e. the substitution in return types. Section 4.3.2 discusses lifetime substitution for subtyping.

Whiley allows for method overloading, i.e. we can have multiple methods with the same name but different parameter types. Furthermore, our extension allows method invocation without specifying lifetime arguments. The compiler then needs to select a suitable method and infer appropriate lifetime arguments. The necessary changes to the method lookup algorithm are presented in section 4.3.3.

4.3.1. Substitution in Return Types

Motivation

Assume we have a method `m` of type `method<a>(&a:int)->(&a:int)`. To calculate the type of the invocation `m<this>(this:new 1)`, we need to *substitute* lifetime parameters in the return type of `m`, i.e. instantiate lifetime parameter `a` with the actual lifetime argument `this`. The substituted type in that example is `&this:int`.

The input to our algorithm is a type and a substitution. A substitution is a mapping from lifetime parameters to their appropriate lifetime arguments. The output of our algorithm is the substituted type.

An intuitive solution would be to take the type automaton and simply replace all lifetime names according to the provided substitution. But this does not work because we could *capture* lifetimes that are bound to lifetime parameters of nested method types. Consider the following program:

```
1 method <a, b> m1(&a:int x, &b:int y) -> int:
2   return *x + *y
3
4 method <a> m2() -> method<a,b>(&a:int,&b:int)->(int):
5   return &m1
6
7 method main():
8   b:
9     any m = m2<b>()
```

What should be the type of the invocation expression in line 9? We call the substitution algorithm with the return type of `m2`, i.e. `method<a,b>(&a:int,&b:int)->(int)`, and the substitution `a` $\mapsto$ `b`. The substituted type should be equivalent to

`method<c,d>(&c:int,&d:int)->(int)`.

In particular, we must not substitute it to

`method<a,b>(&b:int,&b:int)->(int)` or
`method<b,b>(&b:int,&b:int)->(int)`.

Whiley supports recursive types. We previously discussed `LinkedList` as an example for recursive record types, but also method types can be recursive. The following program uses a recursive method type:

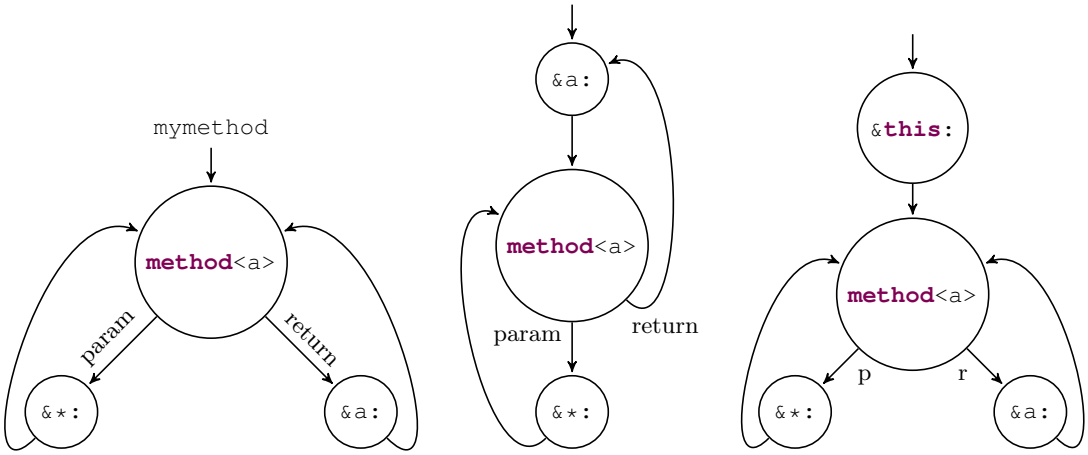
```
1 type mymethod is method()->(mymethod)
2 method m()->mymethod:
3   return &m
```

Recursive types are challenging for lifetime substitution. The structure of the substituted type might be different from the original one and the type automaton might get bigger during substitution. Consider the recursive method type declared by

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

Its type automaton is provided in Figure 3a.

Our substitution algorithm will be called on the return type. We therefore first need to



(a) recursive method type **(b)** return type **(c)** substituted return type

Figure 3: Substitution in a type automaton for a recursive method type.

extract the type automaton for the return type. It is shown in Figure 3b. This extraction is easy, as we only need to change the root state.

Assume there is a method `m` of this type `mymethod` and we invoke it as `m<this>(new &m)`. To calculate the type of this invocation expression we need to apply the substitution `a` $\mapsto$ `this` to the extracted return type from Figure 3b. The result is given in Figure 3c.

The substituted type is bigger than the original one, i.e. the automaton has more states. It is therefore important to prevent infinite recursion in the substitution algorithm.

Solution

Our substitution algorithm is aware of these challenges. The implementation can be found in class `wyil.util.type.LifetimeSubstitution`. It takes a type and a substitution as input and returns the substituted type.

The key approach is as follows: the algorithm uses the extracted type automaton from the original type and builds a new type automaton for the substituted type. We recursively copy all automata states, beginning with the root state. We apply the substitution while copying the states to the new automaton.

The algorithm recurses down the automaton structure in a depth-first manner. When we reach a method type, then we remember its lifetime parameters in an `ignored` set. They will not be substituted while copying the method's parameter and return types. This prevents the capturing problem described above.

Finally, we need a solution to mitigate infinite recursion for recursive types. We therefore store a mapping of all previously copied states to their new states. If we need to copy the same state again, then we take the one that has already been copied. But the mapping needs to be aware of lifetime parameters that are excluded from substitution:

we can only re-use an already copied state if the previous and current `ignored` sets are compatible.

To see whether two `ignored` sets are compatible, we additionally store the set of substituted lifetimes for each copied state. It contains only the lifetimes from the substitution that have actually been replaced while copying that state and its children. We can re-use a copied state if the current `ignored` set is a superset of the previous one and it does not contain any lifetimes that have been substituted while when copying the state. In particular, two `ignored` sets will be compatible if they are the same.

We can have several copies of the same original state that use different `ignored` sets. The resulting automaton therefore might have more states than the original one. But the size is limited: all `ignored` sets are subsets of lifetimes contained in the substitution. For an original automaton with n states and a substitution that contains m lifetimes, the resulting automaton has a maximum of $n * 2^m$ states.

4.3.2. Substitution for Subtyping

The second use-case for substitution is subtyping and intersection of method types. The key problem is that we need to compare two type automata that might be cyclic, i.e. for recursive types. The logic that avoids infinite recursion relies on the fact that the automata do not change, we only use a different state as root state. We therefore cannot use the algorithm from section 4.3.1.

Our solution is added to class `wyil.util.type.SubtypeOperator`. The intersection algorithm recurses down both automata structures. We maintain for each automaton a lifetime substitution. It is applied when comparing lifetimes.

The substitution maps lifetime parameters to **`$1`**, **`$2`**, etc. By choosing the `$`-prefixed names we avoid collision with other lifetime names, because identifiers cannot start with `$`. Furthermore, we can avoid capturing of nested lifetime parameters easily: when recursion visits a nested method type, then the current substitution is updated. We add entries for the nested lifetime parameters, overwriting any existing substitution for that name. This way we ensure that lifetime names in structures with nested method types are always bound to the innermost lifetime parameter with the same name, i.e. the outer lifetime parameters cannot capture any lifetime names from nested method types.

We have to distinguish lifetime parameters from different nesting levels. The two types

```
method<a>() -> method[a]<b>(&a:int)->(&b:int) and
method<a>() -> method[a]<b>(&b:int)->(&a:int)
```

are different. But both `a` and `b` refer to a first lifetime parameter of a method and therefore would be replaced with **`$1`**. We therefore encode a nesting level into the lifetime names. The lifetime parameters from the innermost method type will be substituted with **`$1`**, **`$2`**, etc. For each outer level we prepend a `$`, i.e. **`$$1`** and **`$$2`** for one nesting level, **`$$$1`** and **`$$$2`** for two nesting levels and so forth. For the two types above, `&a:int` will be substituted with `&`**`$$1`**`:int` and `&b:int` with `&`**`$1`**`:int`. This allows to distinguish both types.

We start with an empty substitution for both automata. When we hit a method type then we first prepend a `$` to each entry that is already part of the current substitutions.

Then we add the new entries, overwriting possibly existing ones. Instead of modifying the automata we apply the substitution when reading a lifetime name. This allows us to keep the existing logic that avoids infinite recursion.

4.3.3. Method Lookup

Method lookup is the process of resolving overloading, i.e. selecting a suitable method when there are multiple methods with the same name. *Whiley* collects all methods that have compatible parameter types, i.e. the given argument is a subtype of the expected parameter type. From these *candidates*, the method with the most specific parameter types is used. If there is not a unique most specific candidate, then the method call is said to be *ambiguous*, which is a compile-time error. This is similar to *Java*'s behavior for calling overloaded methods.

The method lookup algorithm needs to apply a lifetime substitution in parameter types. Furthermore, the lifetime arguments might not be explicitly specified. Consider the following program:

```
1 method <a> id(&a:int x) -> &a:int:
2   return x
3
4 method main():
5   &this:int x = id(this:new 1)
```

To see whether the method from line 1 is a candidate for the invocation in line 5, we need to compare the parameter types with the types of the actual arguments. The first method parameter is of type `&a:int`, but the provided parameter has type `&this:int`. In order to compare them, we need to replace the lifetime parameter `a` with a lifetime argument. But there are no lifetime arguments provided in the program. Assume we guess **`this`** as suitable lifetime argument for `a`. Then we use the substitution $\texttt{a} \mapsto \textbf{\texttt{this}}$ and the substituted parameter type `&this:int` matches the provided argument type.

Our extension considers that substitution when comparing parameter and argument types. We use the substitution algorithm from section 4.3.1. But the challenging task is to actually guess suitable lifetime arguments if they are not provided by the programmer.

Our solution is as follows: we consider all lifetimes occurring in the provided argument types and the default lifetime $\star$ as possible values for lifetime arguments. Then we enumerate all possible combinations for a list of lifetime arguments with these values.

Let p be the number of lifetime parameters and a be the number of lifetimes extracted from the given arguments, including $\star$. We have a different options for each of the p parameters. The total number of combinations therefore is a^p.

Method lookup is implemented in the `resolveAsFunctionOrMethod` method of class `wyc.builde.FlowTypeChecker`. It first collects all methods with the right name and number of parameters. The actual selection logic is in `selectCandidateFunctionOrMethod`. If lifetime arguments have been specified explicitly, then we only consider methods with a matching lifetime parameter count. We check each method with the substitution defined by the provided lifetime arguments.

If the lifetime arguments are omitted, then we first extract the lifetimes that shall be used as lifetime arguments. Then we try out all combinations for every method. A *valid candidate* is a pair of method and lifetime arguments.

A single method can yield multiple valid candidates. Consider the following method:

```
1 method <a, b> fst(&a:int x, &b:int y) -> &a:int:
2   return x
```

Assume we call `fst` as `fst(this:new 1, *:new 2)`. Our algorithm extracts the lifetimes `this` and `*` from the arguments. There are four combinations for lifetime arguments: `<this, this>`, `<this, *>`, `<*, this>`, and `<*, *>`. The latter two do not yield a valid candidate because the first argument is of type `&this:int` which is not a subtype of the substituted first parameter type `&*:int`. The former two will successfully type-check, we therefore get two valid candidates. The substituted parameter types for the first valid candidate are `(&this:int, &this:int)`, the types for the second candidate are `(&this:int, &*:int)`. We have to chose the one with the most specific parameter types. We see that the types in the second candidate are subtypes of the first candidate. The second candidate is therefore more specific than the first one.

Our algorithm iterates over all valid candidates. If there is another candidate that is more specific, then the less specific one is discarded. If there is more than one choice left then the method invocation is ambiguous.

4.4. Intermediate Language

The *Whiley Intermediate Language (WyIL)* is a byte-code format. The *Whiley* compiler first translates the source file to the intermediate language. This step involves parsing, type-checking and method lookup. The intermediate code is then translated to a destination language. This step is done by a backend, e.g. the *Whiley to Java Compiler*.

We needed to adjust some parts of the intermediate language for our lifetime extension. The main part is to extend types. Class `wyil.lang.Type` contains logic to print types in a human-readable format, to save types in a byte-code format and to interpret that byte-code format to construct the type again. The affected types are reference types and method types.

The type automata nodes are annotated with supplementary data, e.g. the field names in records types. There is an `Object` typed field `data` for that purpose. Reference types do not yet use that field. We therefore can store the lifetime name in it.

Method types are more complicated. They use the `data` field to store the number of parameters. We define a dedicated `Data` class for method types, containing the number of parameters, the context lifetimes, and the lifetime parameters. *Whiley* uses an algorithm to *normalize* type automata. It relays on suitable `equals` and `hashCode` methods and uses an order among types. To define that order, we need to implement `Comparable<Data>` in our `Data` class.

5. Evaluation

In this chapter we evaluate our contribution. We defined a concept of lifetimes for the *Whiley* programming language and implemented our changes. This chapter provides a short performance evaluation and a case study on how to use our extension for improved memory management.

5.1. Test Cases

Whiley ships with a collection of test cases in the form of programs. There are valid and invalid programs. The test cases check that valid programs compile successfully and their execution passes all **`assert`** and **`assume`** runtime checks. Compilation of invalid programs has to fail and yield the expected error message.

To see the performance impact of our lifetime extension on legacy programs, we compare execution times of existing tests with and without the lifetime extension. The test execution time is the compilation time for the test program and for the valid programs additionally the program's execution time. To minimize the effect of statistical distribution, we run the tests 25 times each and take the average value for comparison. We execute the tests on an Intel Core i3 2 GHz dual core processor.

There are 483 existing test cases for valid programs. The total test execution time for these tests increases from 51.95 to 57.49 seconds when using our extension. This is an increase of 10.66%. Compilation time of the 261 existing invalid programs increases from 9.09 to 9.81 seconds, i.e. by 7.92%.

test case	old	new	diff [ms]	diff [%]
FunctionRef_Valid_11	0.119	0.188	69	57.98
FunctionRef_Valid_12	0.097	0.212	115	118.56

Table 1: Tests with significantly changed execution time

We filtered for the tests with an increase of at least $60ms$ absolutely and at the same time 25% relatively. this yields only two tests, they are listed in Table 1. Both test programs call functions with other functions as parameter, e.g. `f1(&f2)`. The compiler then has to check whether the type of function `f2` is a subtype of the parameter type of `f1`. Subtyping on function and method types got more complicated because we have to consider context lifetimes and lifetime parameters. Even for programs that do not use lifetimes these checks impose some overhead because of added case distinctions. The general increase in execution time can be explained with an increased size for the abstract syntax tree and type representations.

Our implementation ships with 17 new valid and 14 new invalid programs. The total execution time for these tests is 2.12 seconds for the valid and 0.12 seconds for the invalid

programs. The test cases also cover the method lookup algorithm with omitted lifetime arguments as discussed in section 4.3.3. None of these tests takes longer than 0.2 seconds.

5.2. Memory Management

One goal of introducing lifetimes in *Whiley* is to provide a basis for improving memory management. In particular, we want to allow for deallocating dynamically allocated memory without garbage collection.

We establish here a short study on how our lifetime extension can be used for memory management. This is especially interesting for the *Whiley* compiler backend that generates *C* code because it currently cannot deallocate dynamically allocated memory. Some parts of the presented work can also be implemented in the *Whiley to Java Compiler*, but the crucial step of deallocating memory is not possible in the JVM because it relies on garbage collection.

The tactic presented in the following avoids garbage collection. Memory that is allocated with a lifetime other than $\star$ will be deallocated automatically. For memory allocated with $\star$, we rely on reference counting and accept that cyclic structures cannot be freed without manually breaking the cycles. This also holds for *Rust* [15].

A program's memory is divided into the *stack* and the *heap*. The stack contains local variables and management information to organize method calls. The heap contains dynamically managed data. Data on the stack is addressed as a statically known offset to the current stack pointer. This imposes the constraint that the size of these objects must be known at compile-time. This holds for primitive types like booleans or 32-bit integers. Also arrays of these types with a known length have a known size, but we cannot calculate the size of an array with unknown length.

Consider the following program:

```
1 type u8 is int x where 0 <= x && x < 256
2 method main():
3   &this:u8 x = this:new 1
4   m(x) // method<a>(&a:u8)
5   // ...
```

We define a type `u8` for 8 bit unsigned integers, because *Whiley*'s type **`int`** is not bounded in size. We allocate an integer and pass its pointer `x` to a method `m`. That method can then modify the referenced value and `main` can see the changes afterwards. We can allocate the integer directly on the stack. Figure 4 sketches a possible stack layout for method `main`. `x` actually contains the memory address of the cell pointed to by the arrow. Method `m` expects a reference as parameter, i.e. a memory address. It does not need to know whether the address is part of the stack (as in the figure) or part of the heap. There is no need to explicitly free the memory allocated in line 3,

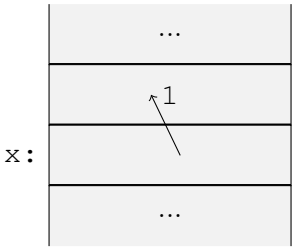


Figure 4: Stack Layout

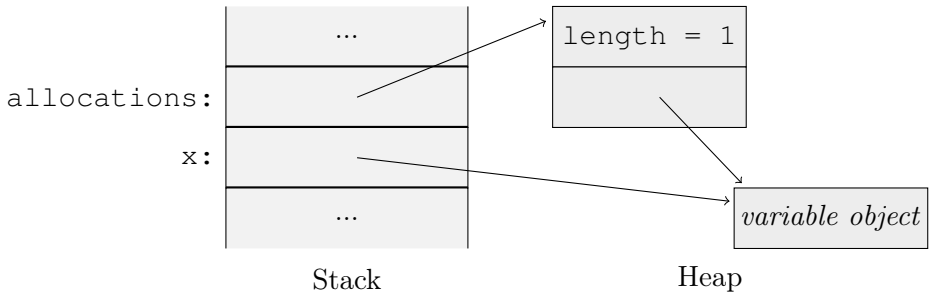
because the whole stack frame will be freed when `main` returns.

If the allocation is inside a loop and the loop's body does not outlive the lifetime of the allocation, then it might not be possible to place these allocations on the stack: each loop iteration allocates a distinct portion of memory and with an unknown number of iteration it might not be possible to statically compute a stack layout with space for every allocation. These allocations and allocations of objects with unknown size must be placed on the heap.

But also the heap-allocated memory can be deallocated without garbage collection: for each lifetime we manage a list of memory allocations. When the lifetime goes out of scope, i.e. at the end of a named block or a method, then the allocations in that list will be freed. Each allocation with **`new`** generates an entry in that list. Consider the following sketch of a program:

```
1 type unknownSize is ...
2 method main():
3   &this:unknownSize x = this:new ...
```

The memory layout can be as follows:



The address of the allocation done in line 3 will be stored in the dynamic `allocations` list. Assigning another reference to `x` later in the program does not affect the reference in that list. When method `main` ends then we free all memory from the `allocations` list and the list itself. The list is placed on the heap such that we can dynamically add elements, this might be necessary for loops.

When calling a method with lifetime parameters, then the compiler will also pass these `allocation` lists for all argument lifetimes. That way, allocations in the callee that use a lifetime parameter can be linked in the appropriate `allocations` list.

Only lifetimes that are used with **`new`** or passed as lifetime argument to another method need an `allocations` list. We can omit the list for other lifetimes. Lifetime arguments, either explicitly given or automatically inferred, are available to the called method as lifetime parameters. The called method might allocate memory for those lifetimes and therefore needs access to their `allocations` lists.

Furthermore, if the number of allocations for a lifetime is statically known, then we can place the `allocations` list for that lifetime as an array directly onto the stack.

6. Conclusion

This chapter briefly revisits the contributions from this thesis. Furthermore, we provide some options for future extensions of *Whiley*. We outline how these extensions are affected by and possibly can benefit from our work on lifetimes. Afterwards, we draw our final conclusions.

6.1. Contribution

- We extended the *Whiley* language and introduced the concept of lifetimes. Our extension fits to *Whiley*'s environment. We had to consider structural typing and flow typing, both features are not present in *Rust*. We kept the *Whiley* language as simple as possible and mostly backwards-compatibility, with keyword **`this`** being the only breaking change. We gave a discussion on our design decisions and found a solution that can be implemented as part of this thesis.

- We presented all changes to the *Whiley* syntax. We gave formal specifications for each change such that they can be easily incorporated into the *Whiley Language Specification*. The given examples help especially those readers who are unfamiliar with *Whiley* to understand each change.

- We implemented the lifetime extension for the *Whiley* compiler which is written in *Java*. Our implementation covers all aspects that are presented in this thesis. We can now compile and run *Whiley* programs with lifetimes and the compiler statically checks all defined constraints. Main challenges were the substitution algorithm for subtyping, the algorithm for method lookup, and the handling of lambda expressions. Our implementation has been merged and released as part of *Whiley* v0.3.40[7].

- We documented the implementation in chapter 4. Example programs, figures and further explanation aim to improve understandability of the algorithms in our implementation.

- We outlined a possible future extension for the *Whiley* compiler backends that utilizes lifetime information for automatic and safe memory management. The presented technique can be used to greatly improve the *Whiley to C compiler* (though actually doing this remains as future work).

[7] http://whiley.org/2016/05/28/whiley-v0-3-40-released/

- Our implementation ships with several additional test cases, covering valid and invalid programs. They demonstrate expressiveness of our lifetime extension and are a good indication of correctness of our implementation.

6.2. Future Work

Embedded Devices and Memory Management

One main reason for porting the idea of lifetimes from *Rust* to *Whiley* was to provide a basis for automatic and safe memory management without garbage collection. Previous work on compiling *Whiley* programs to embedded devices could not deallocate dynamically allocated memory [28]. In section 5.2, we presented a technique how a *Whiley* compiler backend can use lifetime information for this purpose.

Parametric Type Declarations

There are two main reasons for type declarations in *Whiley*: they shorten programs because we can use the type's name instead of its possibly long definition, and more importantly, they allow to define recursive types. There is currently no way to declare types with parametric lifetimes. For example, it might be desirable to declare a type for linked lists where all components are connected using references of the same lifetime:

```
1 type LinkedList<a> is null | &a:{int: head, LinkedList tail}
2 method <a> add(LinkedList<a> l, int x):
3   // should modify l and append x
4 method main:
5   LinkedList<this> l = this:new {head: 2, null}
6   add(l, 4)
```

The syntax `LinkedList<a>` is not yet supported. A future extension could add support for parametric lifetimes together with parametric types, i.e. allowing to define one `LinkedList` that can be instantiated for **`int`** and **`bool`** types as `head`.

Inference

Some programming languages allow to declare variables without specifying the type explicitly, the type will then be inferred from the remaining program. *Rust* is one of these languages. Type inference can speed up programming, because there is less code that has to be written. On the other hand, explicitly specified types can improve understandability of programs.

Our implementation is able to infer suitable lifetime arguments for method invocations as part of the method lookup algorithm. But we currently require the programmer to specify lifetimes on reference types and when using the **`new`** operator. If no lifetime is given, then the default lifetime $\star$ is assumed. It might be desirable for some places to omit an explicit lifetime annotation and automatically infer a better solution.

To benefit from the memory management strategy outlined in section 5.2, we should avoid `*` and use smaller lifetimes when possible. First of all, the lifetime in a **`new`** expression can be chosen as the smallest lifetime in scope such that the expression still meets the expected type. An assignment like `&this:int x = new 1` should allocate an integer of lifetime **`this`** and not `*`. But inferring the expected type might not be possible if the expression is an argument for a method call: with overloading, the method to chose and therefore the expected parameter types depend on the provided argument types. Furthermore, lifetime parameters without lifetime arguments might prevent such a type inference for the method arguments.

For reference types without specified lifetime we currently assume lifetime `*`. In some cases it is possible to change these lifetimes to **`this`**, in particular when the method's parameter and return types do not contain any references.

Another possible optimization affects method signatures: a method declared as `method m(&int x, &int y) -> &int` accepts two references of type `&*:int` and returns such a reference. Depending on the method body, it might be possible to change the signature to `method <a> m(&a:int x, &a:int y) -> &a:int`. This change does not affect the caller: the compiler is able to find a suitable lifetime argument for `a`. It can be `*`, then the change does not have any effect. But the compiler might also be able to type-check the method call with smaller lifetimes for `a`, e.g. the caller's method body denoted by **`this`**.

Concurrency

As of the time of writing, *Whiley* does not yet provide concurrency. The challenge is to support concurrency in a safe way such that data races and runtime faults are avoided, and that it is still possible to verify programs easily.

Verifying programs with shared memory is non-trivial. The safe core of the *Rust* language uses communication channels instead of shared memory. *Rust* additionally provides some unsafe operations. They are used in libraries that expose a safe interface for accessing shared memory only via mutual exclusion locks.

We can reason about shared memory using *Concurrent Separation Logic* as proposed by O'Hearn [16]. The main challenge here is to find suitable partitions of the heap to establish a proof. It seems to be difficult to find an algorithm that automatically verifies programs without a lot of work done by the programmer.

6.3. Conclusions

The goal of this thesis was to design and implement a concept of lifetimes for *Whiley*. This has been achieved and our added test cases demonstrate that the implementation is actually working.

Furthermore, we hoped that a lifetime system can optimize memory management and in particular remove the dependency on garbage collection. As shown in the evaluation chapter, our lifetime extension successfully satisfies that goal.

A. Bibliography

[1] Aldrich, J., Kostadinov, V., and Chambers, C. Alias annotations for program understanding. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications* (New York, NY, USA, 2002), OOPSLA '02, ACM, pp. 311–330.

[2] Andreae, C., Coady, Y., Gibbs, C., Noble, J., Vitek, J., and Zhao, T. Scoped types and aspects for real-time java memory management. *Real-Time Systems 37*, 1 (2007), 1–44.

[3] Benchmarks Game. Rust programs versus c gcc. benchmarksgame.alioth.debian.org/u64q/rust.html. Retrieved 2016-05-01.

[4] Bevan, D. *PARLE Parallel Architectures and Languages Europe: Volume II: Parallel Languages Eindhoven, The Netherlands, June 15–19, 1987 Proceedings*. Springer Berlin Heidelberg, Berlin, Heidelberg, 1987, ch. Distributed garbage collection using reference counting, pp. 176–187.

[5] Boyapati, C., Salcianu, A., Beebee, Jr., W., and Rinard, M. Ownership types for safe region-based memory management in real-time java. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation* (New York, NY, USA, 2003), PLDI '03, ACM, pp. 324–337.

[6] Brandauer, S., Clarke, D., and Wrigstad, T. Disjointness domains for fine-grained aliasing. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (New York, NY, USA, 2015), OOPSLA 2015, ACM, pp. 898–916.

[7] Cherem, S., and Rugina, R. Region analysis and transformation for java programs. In *Proceedings of the 4th International Symposium on Memory Management* (New York, NY, USA, 2004), ISMM '04, ACM, pp. 85–96.

[8] Choi, J.-D., Gupta, M., Serrano, M., Sreedhar, V. C., and Midkiff, S. Escape analysis for java. In *Proceedings of the 14th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications* (New York, NY, USA, 1999), OOPSLA '99, ACM, pp. 1–19.

[9] Clarke, D. G., Potter, J. M., and Noble, J. Ownership types for flexible alias protection. In *Proceedings of the 13th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications* (New York, NY, USA, 1998), OOPSLA '98, ACM, pp. 48–64.

[10] Dhurjati, D., Kowshik, S., Adve, V., and Lattner, C. Memory safety without runtime checks or garbage collection. In *Proceedings of the 2003 ACM SIGPLAN Conference on Language, Compiler, and Tool for Embedded Systems* (New York, NY, USA, 2003), LCTES '03, ACM, pp. 69–80.

[11] Grossman, D., Morrisett, G., Jim, T., Hicks, M., Wang, Y., and Cheney, J. Region-based memory management in cyclone. *SIGPLAN Not. 37*, 5 (May 2002), 282–293.

[12] Hoare, C. A. R. An axiomatic basis for computer programming. *Commun. ACM 12*, 10 (Oct. 1969), 576–580.

[13] Kahn, O. D., Spillinger, I. Y., and Yoaz, A. Mechanism for prefetching targets of memory de-reference operations in a high-performance processor, Oct. 13 1998. US Patent 5,822,788.

[14] Knuth, D. E. Backus normal form vs. backus naur form. *Commun. ACM 7*, 12 (Dec. 1964), 735–736.

[15] Matsakis, N. D. On reference-counting and leaks. smallcultfollowing.com/babysteps/blog/2015/04/29/on-reference-counting-and-leaks/. Retrieved 2016-05-01.

[16] O'Hearn, P. W. Resources, concurrency, and local reasoning. *Theoretical computer science 375*, 1 (2007), 271–307.

[17] Pearce, D. J. The whiley automata library (wyautl). whiley.org/2011/09/20/the-whiley-automata-library-wyautl/. Retrieved 2016-05-01.

[18] Pearce, D. J. The whiley language specification. whiley.org/download/WhileyLanguageSpec.pdf. Retrieved 2016-05-01.

[19] Pearce, D. J. *An Algorithmic Framework for Recursive Structural Types*. School of Engineering and Computer Science, Victoria University of Wellington, 2011.

[20] Pearce, D. J. Integer range analysis for whiley on embedded systems. In *Object/Component/Service-Oriented Real-Time Distributed Computing Workshops (ISORCW), 2015 IEEE International Symposium on* (2015), IEEE, pp. 26–33.

[21] Potanin, A., Noble, J., Clarke, D., and Biddle, R. Generic ownership for generic java. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications* (New York, NY, USA, 2006), OOPSLA '06, ACM, pp. 311–324.

[22] Reed, E. Patina: A formalization of the rust programming language. *University of Washington, Department of Computer Science and Engineering, Tech. Rep. UW-CSE-15-03-02* (2015).

[23] Reynolds, J. C. Separation logic: A logic for shared mutable data structures. In *Logic in Computer Science, 2002. Proceedings. 17th Annual IEEE Symposium on* (2002), IEEE, pp. 55–74.

[24] Rust Team. Rust homepage. www.rust-lang.org. Retrieved 2016-05-01.

[25] Rust Team. The rust programming language. doc.rust-lang.org/book/. Retrieved 2016-05-01.

[26] Rust Team. The rustonomicon - the dark arts of advanced and unsafe rust programming. doc.rust-lang.org/nomicon/. Retrieved 2016-05-01.

[27] Seidl, H., Wilhelm, R., and Hack, S. *Compiler Design - Analysis and Transformation*. Springer, 2012.

[28] Stevens, M. Demonstrating whiley on an embedded system. Bachelor's thesis, Victoria University of Wellington, 2014.

[29] Tuch, H., Klein, G., and Norrish, M. Types, bytes, and separation logic. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 2007), POPL '07, ACM, pp. 97–108.

[30] Venners, B. *Inside the Java Virtual Machine*. McGraw-Hill, Inc., New York, NY, USA, 1996.

[31] Zorn, B. Comparing mark-and sweep and stop-and-copy garbage collection. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming* (New York, NY, USA, 1990), LFP '90, ACM, pp. 87–98.