



NTNU – Trondheim
Norwegian University of
Science and Technology

A Programming Language with Deterministic Multithreading

Tormod Gjeitnes Hellen

Master of Science in Cybernetics and Robotics

Submission date: June 2015

Supervisor: Sverre Hendseth, ITK

Norwegian University of Science and Technology
Department of Engineering Cybernetics

Problem Statement

Multithreading is often problematic in that effects of timing and scheduling may lead to nondeterministic behavior. This again leads to race conditions and limits testability of the system, in a manner that might not be acceptable for critical systems.

The possibility of making a programming language with multithreading semantics more suitable to critical systems should be explored.

The student shall:

1. Provide a short summary of systematic approaches to multithreading
2. Define a language with syntax and semantics that support writing high-reliability, real-time multithreaded programs
3. Make a prototype implementation of this language

Preface

This report describes a master thesis done as a part of the MSc Engineering Cybernetics course at NTNU. The master thesis represents 30 ECTS points. It was written in the spring semester of 2015.

It assumes that readers have prior experience working with programming languages and knows some common nomenclature.

8th of June, 2015

NTNU, Trondheim

Tormod Gjeitnes Hellen

Acknowledgements

I want to thank:

- My parents, Knut Jacob Windelstad Hellen and Liv Signy Gjeitnes Hellen, who love me even though I call them way less than I should
- My flatmates, Caroline Einen and Ingerid Gjeitnes Hellen, who tolerate my mess
- My supervisor, Sverre Hendseth, whose suspender-belt combination taught me the value of redundancy. He was also a valuable source of calm, motivation and advice while working on the thesis
- My closest friends, Adrian Hjelvik and Jon-Håkon Bøe Røli, for making my days a little brighter and a little more indecent
- The faculty at TU Berlin's compiler course, Peter Pepper and Judith Rohloff, for teaching me how to write a compiler and for giving me points even though I failed the exam
- The Norwegian State Educational Loan Fund, which funded my education, this thesis and my other questionable endeavors
- A diverse set of characters who have brought me joy and taught me a lot about life

Abstract

This report presents a programming language with deterministic multithreading and its compiler. The language demonstrates that when making IO and inter-thread communication sequential, most problems with multithreaded programming disappears, while most of the architectural and some of the performance benefits of multithreading are preserved. Much difficulty in modern programming is a result of insufficient abstraction, and while the popular embedded programming languages are unlikely to be replaced anytime soon, effort still has to be made to figure out the next step in the language evolution. In the language presented, there are also other changes meant to aid in the programming of critical systems besides determinism: Threads are written much like functions, dependencies between functions not contained in each other are explicit and arguments are distinguished by name, not sequence. Finally, threads and objects shared between threads are all visible in a single place.

Sammen drag

Denne rapporten omhandler et programmeringsspråk med deterministisk multitråding og dette språkets kompilator. Språket demonstrerer at når en gjør IO og intertråd kommunikasjon sekvensielt, så forsvinner de fleste problemer med multitråding, mens de fleste arkitekturmessige og noen av de ytelsesmessige fordelene forblir. Mange av problemene i moderne programmering er et resultat av manglende abstraksjon, og selv om de populære språkene for mikrokontrollere ikke ser ut til å bli erstattet med det første, må vi fremdeles gjøre en innsats for å finne det neste steget i språkutviklingen. I det presenterte språket er det også andre forandringer ment å lette programmeringen av kritiske systemer: Tråder skrives som funksjoner, avhengigheter mellom funksjoner som ikke er erklært i hverandre er uttalt og argumenter er skilt fra hverandre ved navn, ikke relativ posisjon. Til slutt er tråder og objekter delt mellom tråder synlig på ett sted.

Contents

1	Introduction	7
1.1	Report Structure	7
2	Background	9
2.1	Author’s Prior Knowledge	9
2.2	Concurrency Paradigms	9
2.3	Compilers	12
2.4	Parser Combinators	15
2.4.1	The Scala Standard Parser Combinator Library	16
2.5	A Quick Tour of Scala, the Compiler Implementation Language . . .	17
2.5.1	Execution	17
2.5.2	Hello World	17
2.5.3	Creating and Using Objects	18
2.5.4	Classes and Pattern Matching	19
2.5.5	Inheritance	21
2.5.6	Iteration	21
2.5.7	Container Types	24
2.6	C++11	25
2.7	Regular Expressions	25
2.8	Deterministic Multithreading	26
2.8.1	Deterministic Logical Clock	26
2.8.2	Deterministic Memory Consistency Model	26
2.8.3	Dthreads	27
2.8.4	Consequence	27
2.9	Related Work	27
2.9.1	CoreDet	28
2.9.2	Deterministic Parallel Java	28
3	Specification	29
3.1	Language Design Goals	29
3.2	Runtime Execution Model	29
3.3	Inter-thread communication	32
3.4	Input and Output	32
3.5	Syntax	32

3.5.1	Modern and Familiar	32
3.5.2	Predictable and Helpful	33
3.6	Scope	34
3.7	Pointers	34
3.8	Operators	35
3.9	Immutability	35
3.9.1	Loops	35
3.10	Types	36
3.10.1	Classes	36
3.10.2	Interfaces	37
3.10.3	Modules	37
3.11	Program Declaration	37
3.12	Built-in Functions	37
4	Analysis and Design	39
4.1	Choice of Intermediate Target	39
4.2	Choice of Compiler Implementation Language	39
4.3	Choice of C++ Compiler	40
4.4	Synchronization Mechanisms in The Intermediate Language	40
4.5	A Need for Annotation	42
4.6	Limitations	42
5	Implementation	45
5.1	Overview	45
5.2	Scanner	46
5.3	Parser	47
5.3.1	Grammar	47
5.3.2	Code	47
5.4	Checker	49
5.5	Code generator	51
5.6	Not Implemented	54
6	Testing	55
6.1	Hello World	55
6.2	Multithreaded Hello World	55
6.3	Synchronized Integer	56
6.4	Functions, Actions, Recursion and the Limitations of Integers	57
6.5	Full Program Test With C++ Intermediate	59
6.6	Error messages	63
6.7	Performance	65
7	Conclusion, Discussion and Further Work	67
7.1	Conclusion	67
7.2	Discussion	67
7.3	Suggestions for Future Work	68

Bibliography	68
A System manual	72
B User manual	74
C Code listing	75
C.1 build.sbt	75
C.2 Main.scala	75
C.3 Ast.scala	78
C.4 Scanner.scala	80
C.5 Parser.scala	83
C.6 Typechecker.scala	86
C.7 CodeGenerator.scala	102
C.8 Error.scala	123

Chapter 1

Introduction

It is commonly understood that writing software is hard and that writing multi-threaded software is even harder. This report concerns itself with a new language - Fumurt - with a functional, though incomplete, compiler. This language is intended as a viability test of some new language semantics and a starting point for further development. The semantics of the language are intended to ease development of multithreaded real-time and reactive applications and produce programs which require less testing and have fewer bugs than the existing state of the art.

Specifying a language and implementing a compiler are inherently difficult tasks. The former is an exercise in subjective judgment and trade-offs and the latter is a challenging exercise in software engineering. When starting to work with this thesis, a language was envisioned that made manual scheduling of threads easier than before, but it was decided that this placed too heavy of a burden on the programmer. The ideas that culminated in this report are the result of several weeks of reconsideration.

Fumurt is a language built with the intention that the programmer shall never be surprised. It strives to make the least possible demands on programmers ability to build mental models and memorize. Therefore Fumurt strives to imbue its syntax with as much meaning as possible and to concentrate declaration of concurrent code in one place (fork-join concurrency not affected). Language design inherently necessitates compromise and Fumurt compromises minimally on readability and predictability, sacrificing instead keyboard typing and rapid iteration. It favors predictability over performance and explicitness over terseness.

1.1 Report Structure

The report is divided into chapters as follows:

- The Background chapter contains information needed to understand the rest of the report and a summary of the state of the art
- The specification loosely outlines how the language should look and behave

- Analysis and Design discusses the high-level choices taken during implementation and the limitations of the current design
- Implementation documents how the compiler is written
- Testing contains examples of input source code and resulting error messages or runtime behavior
- Conclusion, Discussion and Future Work evaluates and reflects on the work done and presents recommendations for future work

The appendices are as follows:

- System manual describes how to compile and run the compiler from source
- User manual describes how to run the compiler from Java bytecode
- Code listing contains the source code

The report layout adheres to a recommendation by University College London[4], modified in consultation with supervisor.

The citation style is that of the Association for Computing Machinery.

A Note on Terminology

During the writing of this report, a word that described both statements and definitions was needed, and it was decided to call them both expressions, despite this not being the usual way to use that word.

Chapter 2

Background

2.1 Author's Prior Knowledge

The inner workings of the compiler are heavily influenced by a course the author took on compilers at the Technische Universität Berlin under Peter Pepper and Judith Rohloff. While no code is reused, the structure of the compiler is very similar.

2.2 Concurrency Paradigms

It is commonly understood that writing software is hard. The development of programming languages is a response to this problem. The common pattern is that flexible features that are easily used to write code that is hard to reason about are replaced by, often several, less flexible features. After all, the less flexible a feature is, the more predictable its use is. Three examples:

- goto replaced by sequence, selection and iteration [8]
- pointers replaced by indexes and references
- mutable variables replaced by immutable values

Interestingly, one can observe that as each feature becomes easier to reason about, the total number of features increase. For example, to eliminate mutation, one needs to also eliminate iteration. One way to do this is by using recursion, which is a full replacement for iteration. But recursion, while allowing immutability, is often harder for humans to understand [22]. To ameliorate this problem, a variety of mechanisms have been implemented, for example map and fold, which performs common functions previously performed utilizing iteration. In this manner, the number of features often increase in the interest of analyzability. Is this generally true? And if so, at what point does the drawbacks of increasing feature number outweigh the benefit of increased analyzability and predictability? Answering

these questions is outside the scope of this report. Much progress has been made in making programs easier to understand and analyze in this fashion, yet there is always room for improvement. In later years, one feature in particular has risen to notability: Concurrency. In the past, concurrency has not been an issue for most programmers but as multi-processor (or multi-core) systems have gone mainstream, so has multithreaded programming[24]. The problems inherent to concurrency can roughly be divided into two categories: Communication and scheduling; making sure the correct information is shared between threads in a correct way and making sure tasks are done at correct times, respectively. One possibility is to let the programmer deal with these problems in an application-specific way. This is notoriously error-prone, however. Several abstractions have been devised for dealing with the two concurrency problems in a systematic manner, to the author's knowledge:

- Actors [16]
- Communicating sequential processes [17]
- Transactional memory[15]
- Synchronous programming[6]

Actors Actors are nondeterministic by definition. Each actor has a function that processes incoming messages. This function can run on its own thread, or a more lightweight system can be used. Regardless, the only way actors can exchange information is through messages. Each actor has a queue and each message received ends up at the back of this queue. When there is a message to process, the actor springs to life, processing messages (in the process probably sending some messages of its own) until its queue is empty again. Actors are similar to computers; actors are like processors with running software, and queues are like network buffers. This means that there is no need to adjust the actor system when there is a desire to spread the actors over several machines; communication over the network and communication over shared memory can both be abstracted away by the actor system. The two mainstream implementations of actors are Akka and Erlang.

Communicating Sequential Processes Communicating sequential processes (CSP) are also based on messages. The crucial difference is that in CSP there's no queue; the sending process blocks until the receiving process has received the message and, depending on whether this is desired, responded to the message. A very elegant property of CSP is that calling a function can be implemented as sending a message; the same syntax can be used for both. Two mainstream implementations of CSP are Go and Rust.

Transactional memory Transactional memory is implemented both in hardware and software, but hardware implementations are not widely available for consumer systems yet. The essence of transactional memory is the same that that of mutexes and locks - that is, shared memory where one prevent concurrent processes from accessing shared memory objects. The difference is that, where mutexes and

locks assume that there will be memory collisions, transactional memory assumes that the normal state of the system is that no two threads write to the same memory at the same time. From the perspective of a thread, it looks like this:

1. The thread locks the variable (O), and makes a read copy ($C1$)
2. The thread performs whatever calculations it wants to perform with the variable
3. If the calculation involves writing to the variable, then the result is written to a write copy ($C2$)
4. Now the thread locks the variable (O), and checks whether it is equal to the read copy ($C1$). If $O == C1$, then the result from the calculation is still valid. If $O != C1$ then the thread deletes the copies ($C1$ and $C2$) and reverts to step 1.
5. If the calculation involved writing to the variable (O), then the write copy is assigned to the variable ($O \leftarrow C2$)
6. The thread now unlocks the variable (O)

For big structures, such as arrays, the copies made are usually only of the read (read-set) and written (write-set) parts of the variable, which reduces copying, and lets several threads modify the same variable at the same time as long they do not affect the same parts of the variable. In transactional memory, it is desired to hold locks in as short time intervals as possible. Assuming the calculation takes a long time and no other thread tries to write to the same variables, the performance gains can be significant. And since protecting variables like this has such a low cost except under write contention, this technique can be applied to all memory the threads share. Transactional memory has many popular implementations, but it is particularly heavily used in Haskell and Clojure.

Synchronous Programming Synchronous programming provides a synchronicity abstraction, the same as is used for logical circuits: Time is discretized and all operations during a time step are done instantly and computed from memory as it was at the beginning of the time step. Notice that the operations done in a time step does not affect each other. Synchronous programming is thus *deterministic*. Synchronous programming is a rarity and the most mainstream implementation is Esterel, which is proprietary.

The Decision Made for This Thesis In the end a decision was made in favor of using a variation on the synchronous programming paradigm. There are trade-offs associated with choosing synchronous programming, but they were determined to be preferable to the alternatives. The main problems with synchronous programming are

1. Difficulty in scaling beyond one physical machine. The cost of global synchronization grows with latency.

2. Performance loss due to processing resources idling as the synchronization abstraction requires all operations to use the same amount of time.

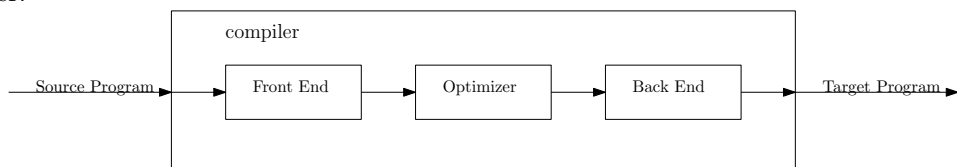
Synchronous programming therefore has substantial problems, yet for single-machine systems it presents a way to achieve near-multi-threaded performance and multi-threaded architecture but with single-threaded predictability and therefore debugability. While the other abstractions place some of the responsibility for correct concurrent behavior on the programmer, synchronous programming takes care of all of that and replaces it with the responsibility for performance, as the program performs best if all threads has an equal amount of work. Let us discuss the problems of the other abstractions:

- Actors assume infinite message queues, with the failure mode being a loss of information. In a producer-consumer relationship, producer actors can overwhelm consumer actors. Actors are designed to mimic distributed systems and create a unified abstraction over these, with the limited guarantees that requires. Distributed systems have to correctly handle hardware failures, so loss of information is an acceptable failure mode for actors. However, this makes actors unsuitable for real-time systems as recovering from data loss and unpredictable memory usage are unacceptable trade-offs. Ordering of IO is also unpredictable.
- CSP systems use synchronous communication and therefore avoid the message queue problem of actors entirely. In exchange, they are open to deadlock, and the ordering of IO is unpredictable. CSP therefore requires brute force search for deadlocks, and debugging is harder than for single-threaded systems. Despite this, it is regarded as a solid choice for real time systems.
- Transactional memory, though it makes it look as if thread communication is easy, has its own problems. The unpredictability of the sequence of writing is a problem, as well as the unpredictable time it takes. Again, IO order is unpredictable.

2.3 Compilers

A compiler is a program (one may regard it as a function) that accepts a program in a source format and outputs a corresponding program in a target format. The source and target format may differ in terms of encoding, language and any other way one may imagine.

This figure, reconstructed from [10], illustrates the structure of a typical compiler:

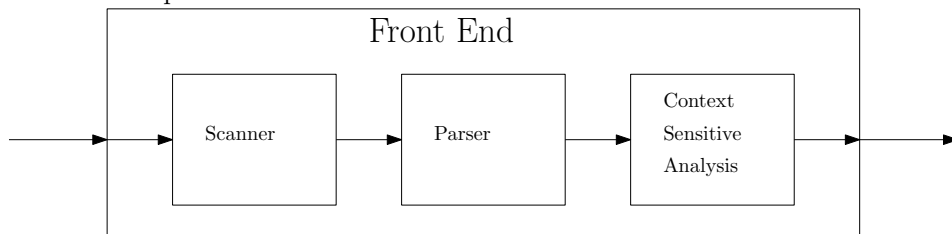


Consider the steps:

1. The front end accepts source text and transforms it into an intermediate representation that is easier to work with. It is generally independent of the target format.
2. The optimizer improves the code as encoded in the intermediate representation. The improvement is usually done with regards to performance, code size or memory usage .
3. The back end accepts the intermediate representation and outputs the the program encoded therein translated to the target format. It can be independent of the source format, depending on how general and flexible the intermediate representation is.

Since the Fumurt compiler (described in chapter 5) does not deal with optimization and conversion to binary itself, but rather outsources this to a C++ compiler, all of the difficult material on instruction selection, scheduling and register allocation is of no relevance. The parts of relevance to this report is the front end and a relatively simple back end.

Consider the parts of the front end:



- Scanner: Transforms source text into a list of tokens (simple objects), possibly ignoring some symbols (such as spaces, comments, indentation etc.)
- Parser: Transforms a list of tokens into an intermediate representation, usually an abstract syntax tree. In the process, it checks whether the syntax of the program is correct.
- Context Sensitive Analysis: Checks the correctness of program semantics. Most interpreted languages skip this step and deal with semantic errors at runtime. The correct time to do semantic analysis is not a settled matter, but in a static compiler such as the one for Fumurt it is done at compilation. This step may or may not emit a modified intermediate representation, but a case in which it would be expected to do so would be when the language has type inference.

The back end is composed of successive passes, of which every step transform the input intermediate representation into an output that is closer to the target format. The number of passes required vary greatly and depend on the differences between the source and output formats. In the trivial case, where the input and output format is identical (for example C to C), the number of necessary passes would be zero.

Grammars

A grammar is a formal and complete description of the syntax of a language. It is mostly used for programming languages. It consists of the confusingly named “production rules”. The standard used here is the Extended Backus-Naur Form of ISO/IEC 14977[25].

Example: Consider that a lower case letter can be described like this:

```
1 lower case letter = "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" |  
    "i" | "j" | "k" | "l" | "m" | "n" | "o" | "p" | "q" | "r" | "s" |  
    "t" | "u" | "v" | "w" | "x" | "y" | "z" ;
```

Where “=” signify the division of the two sides of the production rule, “|” signify alternation (intuitively “or”), quotes signify a string and “;” signify the end of the rule. Let us expand the example by describing a lower case word:

```
1 lower case word = lower case letter, {lower case letter};
```

Note that the correctness of the word as it pertains to English is ignored. The comma signify a sequence, and contents of curly brackets can be repeated from zero up to an infinite amount of times. A lower case word, as it has been defined here, is simply one lower case letter, followed by zero or more lower case letters. Next, the same is done for sentences, again ignoring rules for English:

```
1 lower case sentence = lower case word, {( " , " , lower case word ) | ( "  
    " , lower case word ) }, ". " ;
```

A lower case sentence is here a lower case word followed either by a comma plus space or just space, both followed by a new word. This is repeated as many times as desired and terminated by a period and a space.

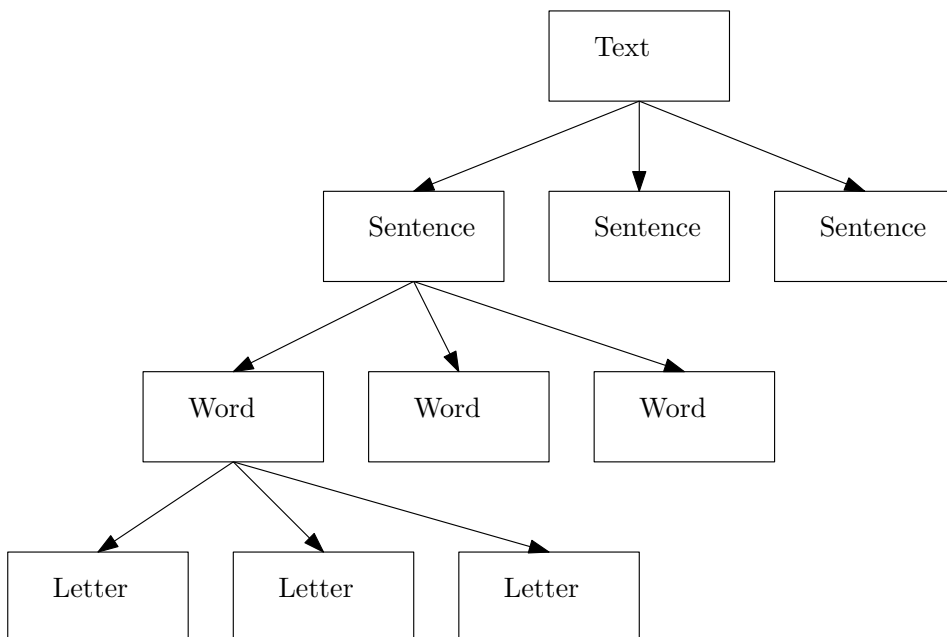
Parentheses allows grouping of sequences. Here, it allows us to alternate between sequences of symbols rather than just single symbols. Finally:

```
1 lower case text = lower case sentence, {lower case sentence};
```

The result is a very simple grammar, which allows us to partition up a text into sentences and words.

Abstract Syntax Trees

Now suppose it was desired to systematize a string of characters according to the grammar above. A data structure corresponding to the grammar would be appropriate. Consider the following figure:



This is an abstract syntax tree, often abbreviated AST. An abstract syntax tree is a tree, in the computer science sense, that represents the production of the source string from the grammar. In code:

```

1 class Text(val sentences:List[Sentence])
2 class Sentence(val words:List[Word])
3 class Word(val letters:List[Char])

```

2.4 Parser Combinators

A parser combinator is a higher order function that accepts parsers as input and returns a new parser[13]. The overall effect is similar to a domain specific language for constructing recursive descent parsers.

A parser is a function that converts one data structure to a more sensible data structure. Usually, the output data structure is more restricted and systematic than the input one.

Example: Consider a function that accepts the string “=” and returns an object of class equalToken or, if the string it is given is not “=”, returns an error object. Such a function is then a parser. Such parsers can be combined to form a larger parser that can work as a scanner, that is a parser that converts a list of characters to a list of tokens (very simple objects). Let the previously discussed function be called the equalParser. Let a parser that works exactly the same, save for exchanging “=” for “-” be called the minusParser and let it return a minusToken upon success. Consider combining the equalParser with the minusParser using

an *alternate parser combinator* (the “|” operator in 2.4.1). The resulting function would then first try the `equalParser`, and if that returned an error object, it would try the `minusParser`, returning an error object if both of these parsers fail. This new parser would not need to return a `minusToken` or `equalToken`, but can process the results from `equalParser` and `minusParser` into something new. In this example, two parsers have been formed and combined into a new parser using a parser combinator. This new parser can be part of a scanner. Indeed, the `Fumurt` scanner is formed like this (see ??).

A Note on Conflicting terminology: Unfortunately there is a case of conflicting terminology concerning the term “parser”. The parser is referred to in two senses:

1. The parser as defined above. A function that converts one data structure to a more sensible data structure.
2. A parser as a compilation step that converts a list of tokens into an abstract syntax tree

2.4.1 The Scala Standard Parser Combinator Library

All the information here is also available at [2].

The Scala Standard Parser Combinator Library introduces many parser combinators, most of whom are formulated as operators.

Let’s discuss these operators:

- `~` is used to combine parsers sequentially
- `~>` is used to combine parsers sequentially but ignore the result of the left parser
- `~!` is used to combine parsers sequentially but disallow backtracking.
- `*` applies the parser to the left as many times as it is successful, moving on at failure
- `+` applies the parser to the left as many times as it is successful, moving on at failure. Must be applied at least once
- `?` applies the parser to the left zero or one time
- `|` used to combine parsers in a manner similar to logical “||”. Tries to apply the left parser first. If the left parser fails, it will backtrack and attempt the right parser. If none work then an error is returned.
- `^^` is used to apply a function to the successful result of the parser.
- `^^^` is used to apply a function to the result of the parser, successful or not.

2.5 A Quick Tour of Scala, the Compiler Implementation Language

In order to understand the code in the compiler, which is included in appendix C, it is helpful to understand the language it is written in. This section gives a quick introduction to Scala.

2.5.1 Execution

There are three ways to execute Scala code:

1. In a read-evaluate-print loop (REPL).
2. Interpreted as a script.
3. As compiled Java bytecode.

The compiler is executed as compiled Java bytecode. Scala can look somewhat different when it is compiled versus when it is interpreted, due to the requirements imposed by the Java bytecode. As a result, methods need to be contained in an object if the code is intended for compilation, but in the REPL and in a script there are no such restrictions. In the REPL and script, statements are evaluated starting from the top, while a main method is required if the program is supposed to be compiled. This report uses only code meant to be compiled or code as it would look in a REPL. The two are easily distinguished by the latter's use of the "scala>" command prompt.

2.5.2 Hello World

A simple Hello World example illustrates some main concepts.

- A singleton is called an "object". These are sometimes called static classes in other languages
- Scope is demarcated using curly braces
- A method is defined using the "def" keyword
- Arguments are given using parentheses (separated by commas and identified by relative position)
- Types of values are written after the object name, separated with ":"
- Unit, as a return type, means the method returns nothing
- Some types are container types, such as List[Int] or Array[String]. These can hold any type through generics. In this case the square brackets means that args is an object of type Array, which in this case holds String. In other words, args is an array of strings.

- There are sequences, like Array or List
- lines need not be terminated with “;” (but it is optional)

```

1 object HelloWorld
2 {
3   def main(args:Array[String]):Unit =
4   {
5     println("Hello, world!")
6   }
7 }

```

2.5.3 Creating and Using Objects

- All values are objects, even native types
- Functions are objects, but methods are not
 - Internally, functions are objects that implement an interface, for example Function1 for functions with one argument. This interface has a method “apply” where the actual “function”, in the C sense of the word, is stored. This is completely transparent to the programmer, however.
- Var lets you create mutable references to objects
- Val lets you create immutable references to objects

```

1 scala> def int1 = 3
2 int1: Int
3
4 scala> val int2 = 2
5 int2: Int = 2
6
7 scala> var int3 = 7
8 int3: Int = 7
9
10 scala> //reassignment to a def is illegal
11
12 scala> int1 = int1+1
13 <console>:8: error: value int1_ = is not a member of object $iw
14     int1 = int1+1
15         ^
16
17 scala> //so is reassignment to val
18
19 scala> int2 = int2+1
20 <console>:8: error: reassignment to val
21     int2 = int2+1
22         ^
23
24 scala> //reassignment to var is completely ok
25
26 scala> int3 = int3+1

```

```

27 int3: Int = 8
28
29 scala> int1+int2+int3
30 res0: Int = 13
31
32 scala> //all values are objects
33
34 scala> int1.+(int2.+(int3))
35 res1: Int = 13
36
37 scala> //even functions
38
39 scala> val square = ((x:Int) => x*x)
40 square: Int => Int = <function1>
41
42 scala> square(3)
43 res2: Int = 9
44
45 scala> square.toString
46 res3: String = <function1>
47
48 scala> //methods are not objects
49
50 scala> def cube(x:Int) = x*x*x
51 cube: (x: Int)Int
52
53 scala> cube(3)
54 res4: Int = 27
55
56 scala> cube.toString
57 <console>:9: error: missing arguments for method cube;
58 follow this method with '_' if you want to treat it as a partially
59     applied function
60     cube.toString
        ^

```

2.5.4 Classes and Pattern Matching

- Classes work much like they do in Java
 - Case classes are different than normal classes.
 - Their constructors can be used like normal functions. The “new” keyword is not necessary
 - Their constructor parameters are exported
 - One can use pattern matching on them. Pattern matching allows one to test which type an object has and extract its values, a reference to it or both.
- * Pattern matching looks like this:

```

1  val x:String = input match
2  {
3    case TypeA("specific string") => "specific string"

```

```

4   case TypeB(anystring, otherstring) => anystring + " "
      + otherstring
5   case TypeB(_,otherstring) => "only care about
      "+otherstring
6   case TypeB(_,_) => "only care about type"
7   case reference:TypeA => "the object looks like this:
      "+reference.toString
8   case reference @ TypeA(str) => "both the reference and
      the constructor parameter"
9   }

```

- The wildcard “_” can be used to represent anything. In pattern matching it can be used much like “else” would in an if statement

```

1  scala>//classes in scala function much like classes in Java
2
3  scala> class A(int:Int, str:String)
4  defined class A
5
6  scala> val a = new A(3,"a string")
7  a: A = A@66ae2a84
8
9  scala>//case classes, on the other hand, have more functionality.
      Their constructors are called like normal functions
10
11 scala> case class B(str:String, int:Int)
12 defined class B
13
14 scala> val b = B("other string", 5)
15 b: B = B(other string,5)
16
17 scala>//and one can pattern match on them
18
19 scala> case class C(double:Double, int:Int)
20 defined class C
21
22 scala> val c = C(3.0, 3)
23 c: C = C(3.0,3)
24
25 scala> def matchfunc(in:Any):Unit = in match
26   | {
27   |   case B(string,integer) => println(string + integer.toString)
28   |   case x:C => println(x.double.toString+x.int.toString)
29   |   case _ => println("unknown type")
30   | }
31 matchfunc: (in: Any)Unit
32
33 scala> matchfunc(b)
34 other string5
35
36 scala> matchfunc(c)
37 3.03

```

2.5.5 Inheritance

- A trait is like an interface, a class with only abstract methods, but one that can also have default implementations of methods
- Classes and trait inherit from each other using “extends [first super] with [second super] with [third super]”
- A class can inherit multiple traits. In the case where two traits have the same signature for different method implementations, the last trait to be inherited is the one whose implementation will be used. Inheriting two classes is not allowed.

```
1 scala> trait Super
2 defined trait Super
3
4 scala> trait Side
5 defined trait Side
6
7 scala> trait Side2
8 defined trait Side2
9
10 scala> case class Sub(int:Int) extends Super with Side with Side2
11 defined class Sub
```

Inheritance is used very sparingly in this thesis.

2.5.6 Iteration

- While works like C while loops
- For is a sequence comprehension which works much like in Python.
 - The indices of sequences are represented by 32 bit integers so “for(x <- -1 until Int.MaxValue){println(x)}” won’t work since “-1 until Int.MaxValue” is a range with Int.MaxValue +1 elements. Put differently, the last element’s index in this case is higher than the maximum value of 32-bit integers, which is not allowed.
 - It is possible to iterate over any sequence with the for syntax
- FoldLeft, foldRight and fold allow combination of a sequence’s elements, going left to right, right to left and in an undefined direction, respectively
- Map and flatMap allows transformation of one sequence to another by applying a function to all elements. FlatMap allows the function to additionally eliminate elements whose results will thereby not be a part of the resulting list.

```

1 scala> var int = 0
2 int: Int = 0
3
4 scala> while(int<10){println(int); int=int+1}
5 0
6 1
7 2
8 3
9 4
10 5
11 6
12 7
13 8
14 9
15
16 scala> for(x <- 0 until 10){println(x)}
17 0
18 1
19 2
20 3
21 4
22 5
23 6
24 7
25 8
26 9
27
28 scala> val list = List(0,1,2,3,4,5,6,7,8,9)
29 list: List[Int] = List(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
30
31 scala> for(x <- list){println(x)}
32 0
33 1
34 2
35 3
36 4
37 5
38 6
39 7
40 8
41 9

```

Fold, foldLeft, foldRight, map and flatMap examples:

- fold is supplied with a function which produces a single value from two input values, all three of the same type, fold repeatedly uses this to produce a single value from a list. It takes two arguments using currying syntax; the first is the starting value and the second is the function used to fold two elements into one. It can be executed in parallel.

```

1 scala> (0 to 2).fold(0)((left,right) => left+right)
2 res1: Int = 3
3
4 scala> (0 to 2).par.fold(0)((left,right) => left+right)
5 res2: Int = 3

```


If the fold is executed in parallel, having the starting argument be non-zero or otherwise have consequence in application will give unexpected results:

```
1 scala> (0 to 2).fold(1)((left, right) => left+right)
2 res1: Int = 4
3
4 scala> (0 to 2).par.fold(1)((left, right) => left+right)
5 res3: Int = 6
6
7 scala> (0 to 2).par.fold(1)((left, right) => left+right)
8 res2: Int = 5
```

This is generally true: Mixing necessarily sequential operations and parallel collections is a bad idea, and a highly unfortunate pitfall for newcomers to Scala.

- `foldLeft` and `foldRight` are equivalent, except that the iteration over the list goes in opposite directions. In contrast to `fold`, the input and output types can be unequal. These can not be done in parallel.

```
1 scala> (0 to 9).foldLeft("numbers")((string, number) =>
2   string+number.toString)
3 res1: String = numbers0123456789
4
5 scala> (0 to 9).foldRight("numbers")((number, string) =>
6   number.toString+string)
7 res2: String = 0123456789numbers
```

- `map`

```
1 scala> (0 to 9).map(x=>x*x)
2 res1: scala.collection.immutable.IndexedSeq[Int] = Vector(0, 1,
3   4, 9, 16, 25, 36, 49, 64, 81)
4
5 scala> (0 to 9).par.map(x=>x*x)
6 res2: scala.collection.parallel.immutable.ParSeq[Int] =
7   ParVector(0, 1, 4, 9, 16, 25, 36, 49, 64, 81)
```

- `flatMap`

```
1 scala> (0 to 9).flatMap(x=>if(x%2==0){None}else{Some(x)})
2 res1: scala.collection.immutable.IndexedSeq[Int] = Vector(1, 3,
3   5, 7, 9)
4
5 scala> (0 to 9).flatMap(x=>if(x%2==0){None}else{Some(x*x)})
6 res2: scala.collection.immutable.IndexedSeq[Int] = Vector(1, 9,
7   25, 49, 81)
```

Together:

```
1 scala> (0 to 9).par.flatMap(x=>if(x%2==0){None}
2   else{Some(x*x)}).fold(0)((left, right)=>left+right)
3 res1: Int = 165
```

2.5.7 Container Types

Scala has several container types, some more exotic than others.

- Option allows handling of values which may or may not have any content. Both “Some(3)” and “None” can be passed as a parameter of type Option[Int]. Options can be mapped, in which case the unwrapping of the contents and subsequent re-wrapping is handled automatically.
- Either allows handling of values which are one of two types. It’s applicability is therefore a superset of that of Option. Left(3) and Right(“str”) can be passed as a parameter of type Either[Int, String]
- Sets are somewhat similar to arrays in that their size is fixed. However, each element can have a unique fixed type. So “(3, “str”, 5.0)” is a set with type (Int, String, Double). specific places in the set are accessed using “set._n”, where n is the 1-indexed index.

```
1  scala> //Option:
2
3  scala> def maybeSquare(in:Option[Int]):Option[Int] = in.map(x => x*x)
4  maybeSquare: (in: Option[Int])Option[Int]
5
6  scala> maybeSquare(Some(3))
7  res0: Option[Int] = Some(9)
8
9  scala> maybeSquare(None)
10 res1: Option[Int] = None
11
12 scala> //Either:
13
14 scala> def squareOrCube(in:Either[Int,Int]) = in match
15     | {
16     |   case Left(x) => x*x
17     |   case Right(x) => x*x*x
18     | }
19 squareOrCube: (in: Either[Int,Int])Int
20
21 scala> squareOrCube(Left(3))
22 res2: Int = 9
23
24 scala> squareOrCube(Right(3))
25 res3: Int = 27
26
27 scala> //set:
28
29 scala> def change(in:(Int, String, Double)):(Int, String, Double) =
30     (in._1*in._1, in._2+"ing", in._3)
31 change: (in: (Int, String, Double))(Int, String, Double)
32
33 scala> change((3, "str", 5.0))
34 res4: (Int, String, Double) = (9,string,5.0)
```

2.6 C++11

This section covers only the features needed in order to understand the C++ code the compiler generates. The features listed below may or may not have additional capabilities to those mentioned:

- `std::atomic` provides atomic transactions for integral types, boolean and pointers. This means a load and a store operations to this variable will never happen concurrently.
- `std::mutex` is a traditional mutual exclusion lock.
- `std::condition_variable` provides a variable that threads can wait on. Subsequently, one or all threads waiting can be awakened. A thread that wishes to use it must hold a `unique_lock` first. Also allows timeouts on waiting.
- `std::unique_lock` allows more sophisticated use of locks. It is not a mutex, but instead provides more ways to acquire and release locks on mutexes, including timed attempts at gaining locks and releasing locks when leaving the scope of the `unique_lock`.
- `std::thread` provides a standardized wrapping around Pthread and similar OS-specific thread libraries.

2.7 Regular Expressions

Regular expressions are programs used to match strings of text. More specifically, they are finite automata capable of parsing regular languages. In practice, what is called “regular expressions” are often capable of parsing more than just regular languages due to extra features. The IEEE POSIX standard specifies their syntax.

The following explains enough to understand their use in this thesis:

- Square brackets match a single character if that character is inside the square brackets. For instance, “[ab]” matches either “a” or “b”, while “[a-z]” matches all Latin lower case characters.
- A question mark (“?”) signifies that the preceding element can be matched one or zero times.
- Parentheses marks a subexpression.
- Backward slash (“\”) escapes the following character, allowing characters that would usually be interpreted as operators to be interpreted as actual character and vice versa. For instance, “\.” matches a period, while “\d” matches any digit.
- A plus sign (“+”) signifies that there are one or more of the preceding element
- A star (“*”) signifies that there are zero or more of the preceding element

- A vertical bar (“|”) signify that either the character to the left or the right is matched

Example

Integers are matched with this regular expression: “[+-]?(0|[1-9]\d*)”. First, there can optionally be a plus or minus sign, then comes the characters in the parenthesis: Either 0 or a number between 1 and 9 followed by a string of digits. “00201” will not match, but “201” and “-201” will match.

2.8 Deterministic Multithreading

All material here is based on [20] unless otherwise stated. This section is to be understood as a discussion of contemporary approaches to multithreading determinism - they have not influenced this thesis because their determinism models are generally less strict.

Deterministic multithreading is an active area of research. Two components are necessary for determinism:

- A deterministic logical clock, which orders synchronization operations deterministically
- A deterministic memory consistency model, which ensures unsynchronized load operations have deterministic results

2.8.1 Deterministic Logical Clock

There are two main approaches to this:

- Round-robin scheduling
- Instruction-count based scheduling[23]

Both concern themselves with which thread’s turn it is to do synchronization calls. In normal pthread systems, it is the thread which calls first that, say, acquires the lock. Round robin scheduling means that it is the thread that has gotten it last that will get it next. In instruction-count scheduling, the next recipient of the lock is determined by which thread has completed the least amount of instructions, with a tie-breaker. Notice that in the latter model, synchronization call order is not necessarily robust in the face of changing inputs, as some inputs may require more instructions to be performed than others.

2.8.2 Deterministic Memory Consistency Model

The memory consistency model concerns itself with making guarantees about the determinism of memory access. Total Store Order (TSO) guarantees that all writes are globally visible in deterministic order, yet makes no guarantees about when.

In both Dthreads and Consequence described below, synchronization of memory is done whenever there is a synchronization (for example *lock()*) operation, which ensures determinism since the position of these operations are determined by the logical clock. This read determinism is then a result of implementation; TSO does not require it. Alternatives to TSO (for example DRF0[12] and LRC[19]) relax the total store order requirements to guaranteeing that a write with respect to a synchronization object is only visible to the next thread that holds the synchronization object. While the computational result is the same, the total store order requires less memory than relaxed models, since all shared memory needs only one copy and thread-local writes will always be applied and memory freed. In relaxed models, memory copies will have to be made whenever a thread releases a synchronization object and freed when the synchronization object is locked by a new thread. This means memory use scales with the amount of synchronization objects. That said, relaxed models can be faster than TSO since individual threads can be isolated until they need updates, even while other threads synchronize memory among themselves.

2.8.3 Dthreads

Dthreads[18] is a deterministic replacement for Pthreads, using round-robin scheduling and total store order. DTHREADS work by giving each thread, as declared in C/C++, its own process and then cleverly hiding this by reimplementing functions such as *getpid()* to give the same answer for all processes that make up the program. All threads do work in a parallel phase, and upon an event that triggers synchronization, for instance the acquisition of a lock, a serial phase is entered. The updates that any single thread applies to shared memory will be applied in deterministic order in the serial phase.

2.8.4 Consequence

Consequence[20] is, like Dthreads, an deterministic implementation for C/C++. It uses instruction-count based scheduling and provides total store order. Instruction-count based scheduling allows Consequence to be faster than Dthreads. The downside is that Consequence is nondeterministic in the face of changing input and also changes behavior when inserting debugging operations like print-statements. Consequence relies on Conversion, a kernel-implemented version control system, for the memory consistency model. Like in Dthreads, each thread is actually run in its own process.

2.9 Related Work

There are related work which also tries to make multithreaded programs easier to work with, using custom compilers or languages extensions. Among these are CoreDet and Deterministic Parallel Java. CoreDet is a custom compiler for C/C++ made by modifying LLVM. Deterministic Parallel Java is a language extension for

Java. Given that they are about making multithreading easier they are worth mentioning, even though they have not influenced this thesis.

2.9.1 CoreDet

CoreDet is “[a] compiler and runtime system that runs arbitrary multithreaded C/C++ POSIX Threads programs deterministically”[5].

2.9.2 Deterministic Parallel Java

DPJ extends Java with a deterministic features. It is built on the idea of *regions*. The programmer divides memory into regions by annotating classes, and thereafter annotates methods with effect summaries stating which regions are read and written by a method. “The compiler uses the class types and method effect summaries to check that all concurrent (read, write) and (write, write) pairs of accesses to the same region are disjoint” [7][1].

Chapter 3

Specification

It was decided that an approach belonging to the tradition of synchronous programming would be the best choice for this thesis, as described in the background chapter. Given the importance of a familiar superficialities for language adoption[21], it was decided that the language should have a familiar C/Algol-style syntax, rather than invent or adopt something less common.

A Note on the Finality of This Specification

Much of what has been specified has not been implemented. While everything has been given thought, this thought has not been distributed equally, but concentrated on the basic things and that which has been implemented. Everything herein, especially that which is left unimplemented, is to be considered preliminary. Future work should not give this specification undue consideration.

3.1 Language Design Goals

It is the goals of Fumurt to aid in producing correct programs suitable for real-time applications in general, and such multithreaded programs in particular.

3.2 Runtime Execution Model

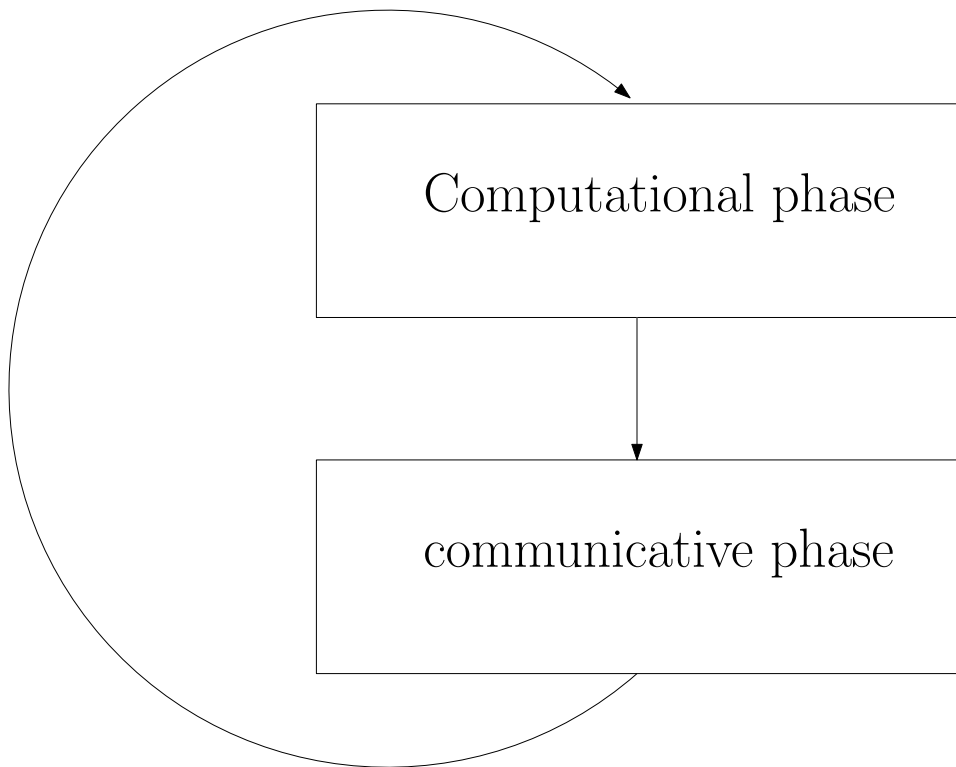
The goal of the programming language is to make a multithreaded program behave as predictably as were it single-threaded and, more generally, to help create reliable applications. A corollary of this is that only changes of state that are visible to a single thread can happen concurrently. All IO and inter-thread communication are required happen in a statically determined sequence. One way to do this is to have the program have two alternating phases:

- Computational phase: In which computations local to a thread are performed.

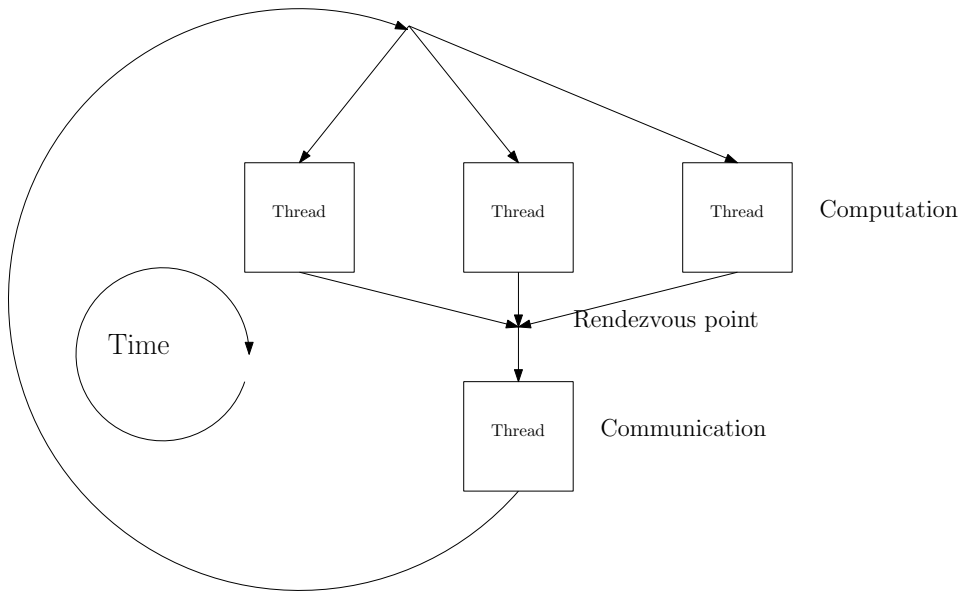
- Communicative phase: In which IO is effected and shared variables are updated, all in a single-threaded manner.

In the computational phase, the order in which computations are performed on the processor is irrelevant as nothing is shared between the thread and the rest of the world. Since the threads have no effect on each other or the outside world in this phase, the only difference between concurrent execution and sequential execution is speed. In the communicative phase, however, execution has to be single threaded. This is similar to Dthreads[18], except here IO sequence is also deterministic and only one thread per synchronized variable have write rights to a synchronized variable. Having only one thread have write privileges means last-writer-wins semantics are avoided, in which everything but the last write since the start of the computational phase will never be visible to other threads. This seems like it would very rarely be the programmer's intended behavior.

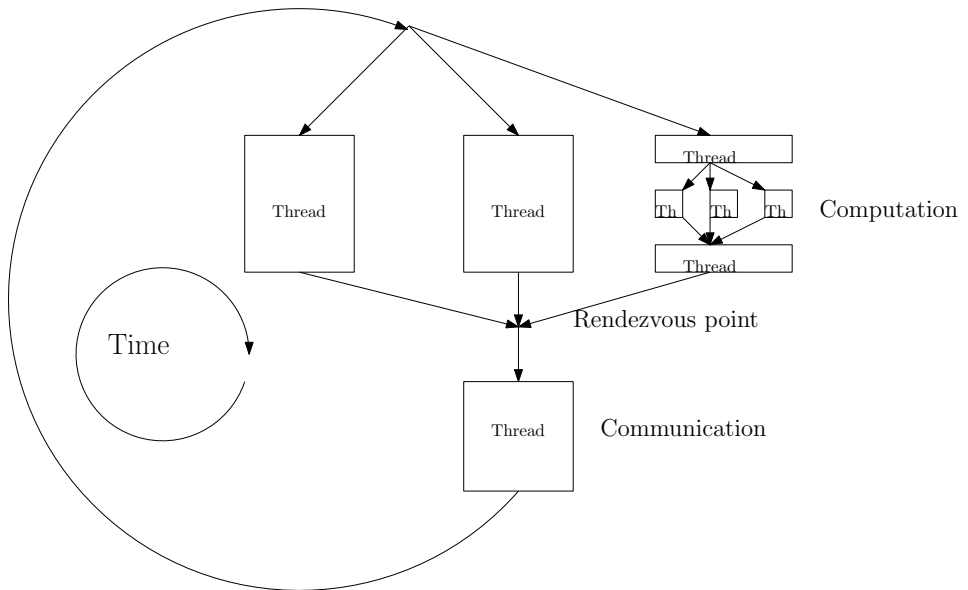
Using this scheme, the application appears to be single threaded both to itself and to the rest of the world, all the while enabling separation of concerns and better utilization of multi-core systems. The following figure illustrates the principle:



In terms of the actual execution a more detailed figure is offered:



Observe that in the computational stage parallel list transformations like map and fold or even futures can be made available, without affecting the outward behavior of the system, except for performance:



Futures and parallel list comprehensions are together applicable to all problems which can be divided into subproblems that can be done in parallel without communication. Futures are a bit of extra work to deal with, but the map-and-fold pattern, sometimes called mapReduce[11], is easy to use and widely applicable to many problems.[9] Indeed, map-and-fold is intensely used in the Fumurt com-

piler. Supporting map-and-fold and futures reduces the performance problems of all threads waiting on each other significantly as long as it can be applied to the most time-consuming task.

The overall effect of this execution model is that phases per second becomes an important measure of responsiveness of the system.

3.3 Inter-thread communication

Inter-thread communication is provided by synchronized variables. These are variables to which one thread has write rights, while all threads has read rights. The writes to a synchronized variable are effected so that all threads can read them during the communication phase. Having only one thread have write rights circumvents the entire problem of store order, and makes sure the programmer doesn't have to worry about the order in which a synchronized variable is written to by the different threads.

3.4 Input and Output

Input and output sequence must be deterministic. To achieve this, output requests are placed in a queue, and are written to the output devices during the communicative phase. Whenever input is desired, the thread will immediately pause and wait for the communicative phase, during which the input will be returned to the thread. In the case where several pieces of input and output do not depend on each other, they can be combined to a composite IO action, whose effects will be performed in the same communicative phase. The syntax for this is left unspecified.

3.5 Syntax

Syntax is by definition somewhat arbitrary but, as Brainfuck demonstrates, some syntaxes are better than others. The following goals were decided upon:

- Look modern and familiar. This is supposed to make it easier to learn, as well as more appealing to someone evaluating whether to learn it.
- Be simple. For ease of implementation.
- Be predictable, and aid the programmer in the understanding of the program.

A language made as part of a master thesis is simple by necessity. The two other goals require more explanation:

3.5.1 Modern and Familiar

Fumurt adopts several conventions from contemporary languages:

- Separating expressions with line endings instead of special characters (for example semicolon).
- Employ “instanceOfType:Type” instead of “Type instanceOfType” when declaring the type of something.
- “=” is used to perform definitions and mark the boundaries of blocks with brackets

This results in syntax with a distinctly modern look:

```
1 function integerIdentity(x:Integer):Integer = {x}
```

One might wish for brackets to be optional in such one-liners, though,

3.5.2 Predictable and Helpful

Although modern languages and their type systems have made the use of functions safe, the syntax of modern languages insufficiently aid the programmer in understanding what a function does, as it is called:

- Functions that perform IO or mutate shared variables are called actions and their names must begin with “action”, like so:

```
1 action actionPrintFoo:Nothing =
2 {
3   actionPrint("  FOO  ")
4 }
```

Similarly thread names begin with “thread” and synchronized variable names begin with “synchronized”. This means that one can observe much about the properties of a call or a variable where it is used without looking up the definitions.

- Function arguments, if there are more than one, are distinguished not by relative position, but by name (as is optionally available in Python). Here is presented a call to the if function and some calls to the toString function:

```
1 if(condition=true, then=toString(1), else=toString(0))
```

Type classes are an alternative to named arguments, the idea being that you have one type per role a variable can play. There are multiple problems with this:

- It’s unnecessarily verbose. Worst case, you’ll end up with one type class declaration per argument.
- Because it’s unnecessarily verbose, the temptation will be to use the same type class everywhere or just use a base class (like Integer) instead. Which would mean that we’re back at square one.

3.6 Scope

Among the goals of this programming language is to help the programmer understand the program. One way this is done is to make dependencies between functions explicit via *inclusions*. It is common among languages for changes in one function to affect the correctness of seemingly unrelated parts of the program. In the following example, changing the definition of function `c` affects the output of function `a`:

```
1  action actionA:Nothing =
2  {
3    b()
4  }
5  action actionB:Nothing =
6  {
7    c()
8  }
9  action actionC:Nothing =
10 {
11   actionPrint("string")
12 }
```

While the above example is a bit contrived, it illustrates the problem. Using inclusions, the dependencies become explicit:

```
1  action actionA(b:Inclusion, c:Inclusion):Nothing =
2  {
3    b(c=c)
4  }
5  action actionB(c:Inclusion):Nothing =
6  {
7    c()
8  }
9  action actionC:Nothing =
10 {
11   actionPrint("string")
12 }
```

Note that inclusions are not functions as arguments - the passed function and the name of the inclusion must have the same name; it is simply there to make dependencies between functions explicit.

In keeping with the goal of being modern and familiar, definitions of functions inside other definitions of functions are allowed. Recursive function definitions, that is. This means that developers can hide functions inside other functions when they are not needed outside them. Inclusions are not needed when functions are defined inside each other, as dependence is implied.

3.7 Pointers

There are no pointers in the language. Any pointers required by C++ is to be hidden by Fumurt. This is because pointers are colloquially as well as formally[22] known to be hard to understand. Programmers should be able to specify specific

memory ranges that can later be written to with functions. These are intended to be used only when the programmer needs to store data to specific addresses.

3.8 Operators

Operators are functions with two arguments and the function name in between the arguments. There are multiple problems with them:

1. Convention suggests that their names should be information-anemically short, often one character. This is obviously problematic
2. It can be hard to figure out what role the different arguments have
3. Operator precedence for user-defined operators is tricky. For math operators there's convention, but for user-defined ones this may be confusing for users of those operators

A prime example of unhelpful operator names can be found in section 2.4.1.

Any good solutions to this have not been found, to the author's knowledge, but it's hard to argue with the convenience of operators. Some predictability to operators are provided by enforcing the following rules:

1. Either the types of the two arguments has to be the same or one of the types have to be a container type of the other. For example `Int` and `Int` or `List[Int]` and `Int`.
2. There's no operator precedence, it has to be defined on a use-by-use basis using parentheses. Ambiguous use of operators are not allowed.

3.9 Immutability

Mutable variables are a major source of bugs, and even experienced developers create bugs when a variable that would have held the correct information previously no longer holds that information. At the same time mutable variables are needed in order to share information across threads. Therefore mutable variables are disallowed, except the synchronized variables that are shared across threads.

3.9.1 Loops

Loops are familiar for many people, yet are usually not included in languages with only immutable values, because their utility is pretty limited. However, they are convenient and they are equivalent to tail-recursion. The major advantages of tail recursion over looping is that the assignment and dependencies are explicit. And yet loops are far easier to understand[22]. Loops that are as safe as tail recursion while being almost as friendly as common loops are possible:

```

1 value y:Int = 5
2
3 value x:Int = loop(y=y,x=y)
4 {
5   if(
6     condition=(y>0),
7     then=
8     {
9       x = x*y
10      y = y-1
11      continue
12    },
13    else=break)
14 }

```

All variables passed to the loop would then need to be copied. In the example above, the `y` modified inside the loop cannot be the same that is defined outside it. Such scoping of variables are common in function calls, and a similar mechanism can be used for loops.

An additional benefit of loops is that their use has constant memory consumption independent of number of iterations. While the same can be achieved for recursion using tail recursion with optimizing compilers, such compilers are still not the norm. Mutual tail recursion optimization is particularly rare. Since optimizations are not an immediate goal for the Fumurt compiler, loops would offer an important guarantee for the programmer.

3.10 Types

3.10.1 Classes

In trying to be familiar, it is desirable to provide types along with their popular object oriented nomenclature. So classes are present, just that they are immutable. They are defined by their constructors, optionally with extra static methods:

```

1 class IntAndString(int:Integer, string:String) =
2 {
3   function combine:String = {concatenate(left=toString(int),
4     right=string)}
5 }
6 value x = IntAndString(int=3, string="something")
7 actionPrint(x.combine)
8 actionPrint("==")
9 actionPrint(concatenate(left=toString(x.int), right=x.string))

```

Fumurt does not have inheritance, because while inheritance means you get code reuse, it also obscures the class that inherits. When one class inherits from a hierarchy, one needs to understand not only what's written about that class but also the entire hierarchy in order to understand the end result.

In order to aid the programmer in understanding their own and others' code, the names of types always lead with a capital letter. Conversely, leading with a

capital letter for anything else is illegal.

3.10.2 Interfaces

All classes are interfaces, but one can also create interfaces that aren't classes using the "interface" keyword. When implementing an interface one explicitly have to note what interfaces the class is implementing.

```
1 interface IntAndString(int:Integer, string:String)
2 //or
3 class IntAndString(int:Integer, string:String)
4
5 class IntAndStringAndBool(int:Integer, string:String, bool:Boolean)
   implements IntAndString
```

3.10.3 Modules

Modules are singletons containing only immutable values, actions and functions. They can therefore serve as libraries. Their scope is handled the same way functions' scope is. This avoids the problem where singletons are global entities and functions' dependence on them are completely obscure.

3.11 Program Declaration

The program declaration is meant to give a high level overview of the behavior of the program. It declares what threads are spawned, in what sequence their IO should be enacted, which synchronized variables exist and which threads have write permission to which variable.

3.12 Built-in Functions

Fumurt provides the following built-in functions:

- `toString(x)` gives a string representation of `x`
- `actionPrint(x)` prints the string `x`
- `actionMutate(variable, newValue)` assigns the `newValue` to the synchronized variable
- `if(condition, then, else)` returns result of *then* if *condition* is true and returns *else* if it is not
- `plus(left, right)` returns $left + right$
- `minus(left, right)` returns $left - right$
- `divide(left, right)` returns $\frac{left}{right}$

- `multiply(left, right)` returns $left * right$
- `equal(left, right)` returns $left == right$

Chapter 4

Analysis and Design

4.1 Choice of Intermediate Target

For easy debugging and wide selection of binary targets it was decided to first compile to an intermediate language and then let an external compiler perform the final transformation to binary form. This is a well-trodden path[14], and C is often used. Though many modern languages would be suitable for this, a wish list of features determined which language to choose:

1. No garbage collection or other other source of run-to-run variability.
2. Wide selection of final targets, including embedded.
3. Low overhead, whether in performance or memory.
4. A solid set of features to make transformation into the language easier.
5. Mature standard that is unlikely to break backwards compatibility.
6. One, preferably more, good and mature open source implementations available.
7. Possibility of running without an operating system.

C++ seems to satisfy all these criteria, and were therefore selected as the intermediate language. Its main competitor, C, has too few features, which means a compiler would have to make more difficult transformations and/or things like linked lists would have to be manually implemented. Such difficulties seem unnecessary.

4.2 Choice of Compiler Implementation Language

Scala was chosen as the implementation language for the compiler partly because it's what the author used in the TU Berlin compiler bau course (see 2.1) and already had lots of experience in, but it also has some highly attractive qualities for making a compiler:

- Solid type checking which makes the code easier to work with, especially when refactoring
- A wide selection of functional abstractions, which allows compact code and eliminates simple but irritating bugs as well as access to imperative constructs like loops etc. when this is more convenient
- A parser combinator library
- Fast execution time

Other languages under consideration were C, C++ and Haskell. C has inadequate abstractions and lackluster type checking. While C++ has much better abstractions, its type checking is still not strict enough to prevent many of the errors that would undoubtedly have been made during development. Haskell has all the features necessary, but the author had previously had problems learning it. It was also a concern that Haskell does not provide non-functional mechanisms, even when these are the best solution to a problem.

4.3 Choice of C++ Compiler

There were two compilers under consideration: GCC and Clang. While Clang is in many ways the better compiler, GCC is installed by default on most Unix systems. That leaves Windows. After some trial and error, it was found that installing a C++11 compliant standard library was difficult on Windows, and that the by far easiest solution on Windows is to install Visual Studio and use the Microsoft Visual C++ compiler. In the end it was decided that the Fumurt compiler will compile the C++ code using GCC, unless it is run on Windows, in which case it will ask the user to compile the C++ code using Visual C++ manually. This is quite clearly the lesser evil, rather than a particularly good solution.

4.4 Synchronization Mechanisms in The Intermediate Language

Our execution model formulated in 3.2 needs to be formulated in the compiled C++ code.

- Each thread gets its own `printList` (type `std::list<std::string>`), and `actionPrints` are translated into `printList.push_back`. The same principle can be used for future output as well. When the threads are finished with the computational phase, the last thread to finish will print `printList.pop_front` until the `printList` is empty. The thread started first in the program statement gets its `printList` emptied first, and so on.
- A rendezvous pattern is used:

1. A macro NUMTOPTHREADS, with the number of threads defined in the program statement is defined
2. A static `std::atomic<int> rendezvousCounter`, which holds the number of threads that have arrived at the rendezvous point is defined.
3. A static `std::mutex rendezvousSyncMutex` and a static `std::condition_variable cv` are defined.
4. For each synchronized variable in the source code, one variable which holds the global state of this variable and one which holds the local state of this variable in the thread that is allowed to write to it is defined.
5. A `[[noreturn]]` static void `threadName()` is defined for each thread, holding its values. All arguments to `thread` in the source code are converted to static global variables. If the platform is Windows, “`__declspec(noreturn)`” is used instead of “`[[noreturn]]`”, since Microsoft Visual C++ does not support C++11 syntax for attributes.
6. A main function is defined, inside of which:
 - (a) `rendezvousCounter` is set to 0, threads (`std::thread`) are started with the thread functions (defined in previous step) as arguments and finally the main function enters a loop executing `std::this_thread::sleep_for(std::chrono::seconds(1))`.
7. static void `waitForRendezvous(std::string name)` which a thread calls when it is ready to wait, is defined. Inside of which:
 - (a) The thread locks the `rendezvousSyncMutex`
 - (b) Increments the `rendezvousCounter`
 - (c) If the value in the `rendezvousCounter` is less than NUMTOPTHREADS, the thread waits using `cv.wait`, at which point `rendezvousSyncMutex` will be automatically unlocked. If the `rendezvousCounter` equals NUMTOPTHREADS, the thread prints all strings held in the `printLists` as described above, sets any global synchronized variables to its writer-local values, sets `rendezvousCounter` to 0 and finally notifies all other threads using `cv.notify_all` before exiting the function. `rendezvousSyncMutex` is unlocked on function exit. Example of a generated `waitForRendezvous` function:

```

1  static void waitForRendezvous(std::string name)
2  {
3      std::unique_lock<std::mutex> lk(rendezvousSyncMutex);
4      ++rendezvousCounter;
5      if (rendezvousCounter.load() < NUMTOPTHREADS)
6      {
7          cv.wait(lk);
8      }
9      else if (rendezvousCounter.load() == NUMTOPTHREADS)
10     {
11         while(!printthreadPrintHello.empty())
12         {
13             std::cout << printthreadPrintHello.front();

```

```

14     printthreadPrintHello.pop_front();
15     }
16     /*similarly for other thread print lists*/
17     synchronizedNumber = writeSynchronizedNumber;
18     //where synchronizedNumber is the name of a
19     //synchronized variable
20     rendezvousCounter.store(0);
21     cv.notify_all();
22     }
23     /*abnormal situation diagnostics mechanism here*/

```

4.5 A Need for Annotation

Technically, the finished code can always be determined directly from the AST, but it was discovered that in order to do this in the Fumurt case, the same rules would have to be encoded into the code in several different places. In the current state of implementation, there are two rules that require annotation. The first was the rule for determining the C++ names of function and the second is the rule for naming arguments to threads. In both cases, Fumurt's semantics are very different from C++'s. There are four aspects to the naming:

1. Actions and functions that are in other functions need to get new names because the hierarchy needs to be flattened
2. Actions need to be demultiplexed, as the C++ code they contain needs to be different depending on which thread calls that action. For instance, an actionPrint needs to be transformed to a push to a list whose name depends on the calling thread
3. Function calls need to be changed so they refer to the new names
4. Arguments to threads need to have new C++ names that will be globally unique.

This can be accomplished by doing two passes over the AST. In the first pass, all function definitions and thread arguments are annotated with their C++ names. In the last pass, all function calls are annotated with the C++ name of the function they call, copying from the annotation done in pass one.

4.6 Limitations

While the specification and design is satisfactory, there are many ways in which it could be improved:

- There are no compound statements, except in the right hand side of definitions

- Definition right hand demarcation of the `begin..end [function/x]` type (for example `begin..end loop`) should be optional, as it can be helpful when reading and writing deeply nested expressions, where exactly what it is that is ending can often be unclear.
- Performance of the current execution model may be a concern for some applications. Allowing programmer-defined synchronization intervals would allow for greater performance without sacrificing predictability. The programmer could then specify that computation-heavy threads participate in only every Nth communication phase. In cases where the appropriate performance and responsiveness requires sacrifices to predictability, it seems prudent to evaluate the possibility of using an instruction-based logical clock system when the programmer specifies it. Systems such as Consequence[20] may make it possible to obtain greater performance in cases where the programmer can allow predictability requirements to be relaxed. Likewise, software transactional memory could be interesting, particularly when a thread needs to wait on input from an unpredictable source, like a human, while the rest of the threads needs to be responsive.
- The design of Fumurt centers around predictability, but in order to guarantee any predictability we have to assume correctness of the underlying hardware. Fumurt is by design not fault-tolerant, because fault tolerance deals with, and causes, unpredictability. This is in many cases insufficient. It would be beneficial if it was possible to construct some system wherein multiple computers or chips running Fumurt code could be coordinated by a system that does deal with fault-tolerance. Erlang with OTP is often used for such applications, but no study has been carried out regarding how to combine Erlang and Fumurt.
- As it is, the design of Fumurt has some, but very little empirical underpinnings. User surveys concerning how the various aspects of the language are received, particularly by novice programmers, would shed light on whether all the ideas introduced in this report are actually good ideas.
- There is no appropriate response in the cases where the IO buffers can no longer fit in memory. A solution which would degrade performance but otherwise work well, would be to pause all threads trying to put IO into a full memory while letting the thread whose IO are to be effected first write directly to IO. Once that first thread is finished, the second thread whose IO shall be effected can write directly to IO and so on until all threads are ready to enter the communicative phase. This will serialize execution, which can degrade performance. In those cases where responsiveness is more important than strict IO sequentiality, special mechanisms may be provided whereby the programmer can specify that in such cases IO buffers shall be emptied to IO during the computational phase.
- Recursion can cause a stack overflow, leading to a segmentation fault. With the exception of the recursion happening when a thread recurses on itself,

no recursion is optimized away. This is problematic in a critical-application system. Some types of recursion are easy to optimize away, some less so. The appropriate behavior for the compiler towards recursion it can't optimize away is undetermined.

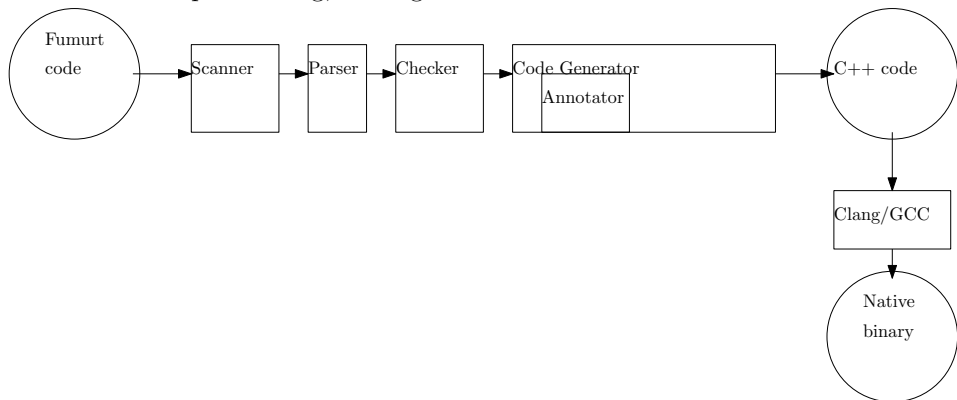
- There is no mechanism for direct access to memory, which is often needed in embedded programming
- There are no lists, arrays or similar sequences. Likewise, loops, values, operators and user-defined types are missing.

Chapter 5

Implementation

5.1 Overview

The compiler consists of four parts: The scanner, parser, checker and code generator. There is no optimizer, although the requirement for no dynamic destruction or creation of threads allows us to use a loop in threads instead of just recursion. This is necessary because neither Clang nor GCC could correctly optimize that tail recursion into a loop in testing, leading to an inevitable stack overflow.



Consider the steps taken by the compiler:

1. The code is scanned. If there is an error it's printed and compilation ended. Note that neither scanner nor parser are advanced enough to detect more than one error at a time.
2. The tokens from the scanner is parsed. If there is an error it's printed and compilation ended.
3. The AST from the parser is handed to the checker, which looks for any semantic errors. If there are any, they are printed out and compilation ended.

4. The AST from the parser is given to the code generator, which produces C++ code conforming to the C++11 standard.
5. GCC is used to compile the C++ code to native binaries.

5.2 Scanner

Scanners, it should be noted, are sometimes called lexers. Drawing on experience from the TU Berlin course (see 2.1), the Scala Standard Parser Combinator Library was chosen.

Parsers for individual tokens are formed like this:

```

1 def intParser: Parser[IntegerT] = positioned( new
   Regex("""(0|[1-9]\d*)""") ^^ {x => IntegerT(x.toInt)} )
2 def equalParser: Parser[EqualT] = positioned( new Regex("=") ^^ {x =>
   EqualT()} )

```

The parsers are then combined into the final scanner using the alternate operator (“|”)[2].

It all goes into a list of tokens. The tokens are defined like this:

```

1 abstract class Token() extends Positional
2 abstract class DefDescriptionT() extends Token
3 abstract class BasicValueT() extends Token
4 abstract class SyntaxT() extends Token
5
6 case class TrueT() extends BasicValueT {override def toString =
   "true"}

```

Positional[3] is a trait that gives the token a Position. The “positioned” call in the parsers assigns the Position to the token. This is all inherited from the parser combinator library, so it’s hard to understand what’s going on from looking at the source alone. The “positioned” call assigns the source code position of the input text to the token object produced by the parser, which allows us to output really nice error messages later on.

Function List

- *scan(in:String):Either[NoSuccess, List[Token]]* takes the source file as a string and either outputs a list of tokens or an error message
- *scanInternal:Parser[Token]* is the internal scanner. The parser combinator library will use this to create a parser to serve as scanner at compile time
- *xParser: Parser[XT]* parses that particular type of token, for example *newlineParser: Parser[NewlineT]*

Classes

- The scanner uses token classes. These are held in Ast.scala

5.3 Parser

Like in the scanner, the Scala Standard Parser Combinator Library was used. Unfortunately, the tasks of the parser is a bit more complicated than those of the scanner, and the code reflects this.

5.3.1 Grammar

The grammar serves as a formal definition of the language. Though not needed in order to understand the language, it is included for completeness. Here's the EBNF ([25]) for the grammar, as implemented:

```
1 prog = paddedDef, {paddedDef}, EOF;
2 paddedDef = {"\n"}, def, {"\n"};
3 def = deflhs, "=", {"\n"}, defrhs;
4 deflhs = defdescription, id, args, ":" type;
5 args = ("(", id, ":", type, {subsequentArg}) | "";
6 subsequentArg = ",", id, ":", type;
7 defrhs = "{", {"\n"}, expression, {"\n", {"\n"}, expression},
8 {"\n"}, " ";
9 expression = def | statement;
10 statement = functionCall | basicStatement | identifierStatement;
11 callargs = "(", (namedcallargs|callarg), ")";
12 callarg = statement | "";
13 namedcallargs = namedcallarg, subsequentnamedcallarg,
14 {subsequentnamedcallarg};
15 subsequentnamedcallarg = ",", namedcallarg;
16 namedcallarg = id, "=", callarg;
17 functionCall = id, callargs;
18 identifierStatement = id;
19 defdescription = "program" | "action" | "thread" | "function" |
20 "value";
21 basicStatement = boolean | string | integer | float;
22 float = integer, ".", digit, {digit};
23 integer = "0" | (digit excluding zero, {digit});
24 digit excluding zero = "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8"
25 | "9" ;
26 digit = "0" | digit excluding zero ;
27 upper case = "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" |
28 "J" | "K" | "L" | "M" | "N" | "O" | "P" | "Q" | "R" | "S" | "T" |
29 "U" | "V" | "W" | "X" | "Y" | "Z" ;
30 lower case = "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" |
31 "j" | "k" | "l" | "m" | "n" | "o" | "p" | "q" | "r" | "s" | "t" |
32 "u" | "v" | "w" | "x" | "y" | "z" ;
33 id = lower case, {(upper case | lower case)}
34 type = upper case, {(upper case | lower case)}
```

For help understanding this, see section 2.3.

5.3.2 Code

This is where the grammar is encoded into the program:

```
1 def progParser: Parser[List[Definition]] = (paddedDefParser.+) <~
2 eofParser
```

```

2 def paddedDefParser:Parser[Definition] = { newlineParser.* ~>
  defParser <~ newlineParser.* }
3 /*more here*/

```

The relevant values are extracted from the result by using the “`._x`” methods, where `x` is a number. This is because the result of several consecutive parsers are combined into sets. “`._1`” is then the first value of the set, etc. The structure of these sets are sometimes not immediately obvious. For the operators refer back to 2.4.1.

There are also a number of somewhat less exciting helper parsers, of which an example is provided:

```

1 def equalParser:Parser[Token] = accept(EqualT())
2 def basicStatementParser:Parser[BasicValueStatement] =
  accept("expected string, integer, boolean or float", {
3 case StringT(value) => StringStatement(value);
4 case IntegerT(value)=> IntegerStatement(value)
5 case TrueT() => TrueStatement()
6 })

```

This shows how the parser error messages are generated.

The entirety produces an abstract syntax tree. Both the checker and the code generator operates on this AST, and it is the centerpiece of the implementation. Without understanding the AST, the rest of the implementation will appear cryptic at best:

```

1 class Expression() extends Positional
2 trait Callarg extends Positional
3 trait Statement extends Expression
4 trait BasicValueStatement extends Statement with Callarg with
  aCallarg with aStatement
5
6 case class Definition(val leftside:DefLhs, val rightside:DefRhs)
  extends Expression
7 case class DefLhs(val description:DefDescriptionT, val id:IdT, val
  args:Option[Arguments], val returnType:TypeT)
8 case class Arguments(val args:List[Argument])
9 case class Argument(val id:IdT, val typestr:TypeT)
10 case class DefRhs(val expressions:List[Expression] )
11 case class Empty();
12 case class DefDescription(val value:Token)
13 case class NamedCallarg(id:IdT, argument:Callarg)
14 case class NamedCallargs(val value:List[NamedCallarg])
15 case class NoArgs() extends Callarg with aCallarg
16
17 case class StringStatement(val value:String) extends
  BasicValueStatement
18 case class IntegerStatement(val value:Int) extends BasicValueStatement
19 case class DoubleStatement(val value:Double) extends
  BasicValueStatement
20 case class TrueStatement() extends BasicValueStatement
21 case class FalseStatement() extends BasicValueStatement
22 case class IdentifierStatement(val value:String) extends Statement
  with Callarg with aCallarg with aStatement

```

```
case class FunctionCallStatement(val functionIdentifier:String, val
    args:Either[Callarg,NamedCallargs]) extends Statement with Callarg
```

Function List

- *parse(in:List[Token]):Either[NoSuccess, List[Definition]]* takes a list of tokens and returns either an error message or an AST
- *progParser: Parser[List[Definition]]* is the first of the parsers, from which the parser combinator library will generate the final parser
- *xParser:Parser[X]* parses that particular kind of AST node, for example *defParser:Parser[Definition]*. Can often be a bit indirect. For example, *paddedDefParser:Parser[Definition]* parses a definition with newlines around it, but uses *defParser:Parser[Definition]* to parse the definition part of that.

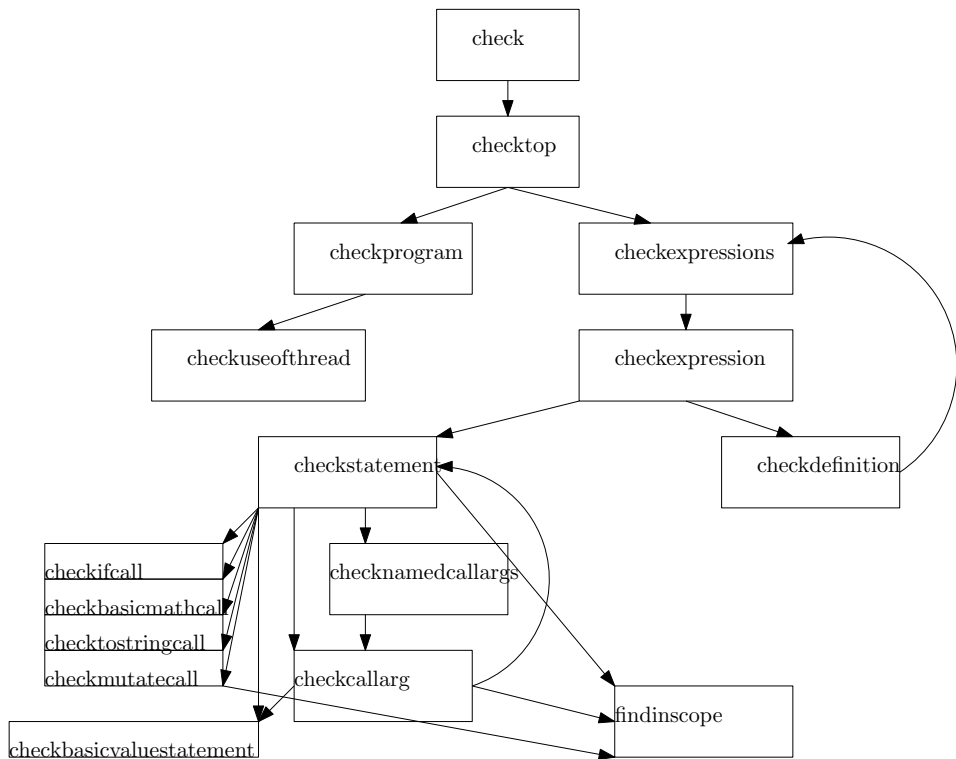
Classes

- *Class TokenReader* is a wrapping around the list of tokens. It is required by the parser combinator library and implements the Reader interface. It has the following functions:
 - *atEnd* which returns true if the list of tokens is empty
 - *first*, which returns the current first element in the list
 - *pos*, which returns the source text position of the first element in the list
 - *rest*, which returns a new TokenReader wrapping all elements except the first in the list
- The parser uses AST and token classes. These are in the Ast.scala file.

5.4 Checker

The checker, contrary to its in-source name (FumurtTypeChecker) checks more than types. It does not modify, annotate or otherwise change the abstract syntax tree. It simply returns errors found or returns nothing. When the implementation of the checker began it was envisaged that the basic functions would be treated equally with user defined functions, but due to the lack of generics and other abstraction mechanisms, most of the basic functions still needed special treatment, with “actionPrint” being the notable exception.

This graphic illustrates how the functions in the checker call each other:



Function List

- *check* is the interface to the rest of the program. Takes in an AST and returns a list of errors, if there are any.
- *checktop* checks the top level of the program. The top is special because it contains threads and the program statement, though only the program statement need special treatment.
- *checkprogram* checks the program statement. Uses *checkuseofthread* and checks whether there are any calls to non-threads or definition of non-synchronized variables.
 - *checkuseofthread* checks that the thread given is actually called in the program statement. Declaring a thread and failing to call it is an error.
- *checkexpressions* checks a list of expressions, such as might be found in the right-hand side of a definition. Uses *indexleft* to get new in-scope definitions and passes them to *checkexpression*
- *checkexpression* checks an individual expression. Determines if the expression is a statement or a definition, and subsequently uses *checkstatement* and *checkdefinition*

- *checkstatement* checks a statement. If it's an identifierStatement, checks that its return value is as expected. Uses *checkbasicvaluestatement* for the same for basic values. If it's a function call, then it either uses special case functions, such as *checkifcall* or finds the function in scope and uses a general approach using *checknamedcallargs* and/or *checkcallarg*
- *checkifcall* checks calls to if. Makes sure the return type of then and else is the same and that condition is a boolean. Also checks naming.
- *checkmutatecall* checks that the variable is a synchronized variable and otherwise has the same type as the new value
- *checkbasicmathcall* checks the four basic math operators, with special attention to the return type when double and int are mixed
- *checktostringcall* checks that there is only one argument and that the expected type is String
- *checknamedcallargs* checks named call arguments. Checks that the correct names are used, that the correct number of arguments are given and uses *checkCallarg* to check each argument individually.
- *checkCallarg* checks a call argument. Makes sure the type is correct. Uses *checkbasicvaluestatement* and *checkstatement*.
- *checkbasicvaluestatement* checks that the type of the basic value is correct.
- *checkdefinition* checks a definition. makes sure the return type is the one specified, that an action is not defined or used from inside a function etc.
- *indexlefts(in:List[Expression]):List[DefLhs]* takes a list of expressions and returns a list of all the left sides of definitions in that list.
- *findinscope* finds a left side of the definition in the current scope with the same name as that which is searched for.

5.5 Code generator

The code generator can best be explained step by step:

1. First the C++ include statements are determined. These are currently hand-written.
2. We scan the program declaration and find the threads that will be started in the main thread. The statements for those are found in the program declaration.
3. The main function is determined from the list of thread statements and their arguments

4. The print list declarations are determined from the list of thread statements.
5. The NUMTHREADS macro is determined from the length of the list of threads.
6. The abstract syntax tree and a list of the threads are passed to the annotator, which returns an annotated tree.
 - (a) The definitions are annotated with their C++ names, and actions called by several threads are demultiplexed into one per calling thread. Inclusion arguments are removed from the signatures. Thread arguments are annotated with their C++ names.
 - (b) The calls to functions and actions are annotated with the correct C++ name, and inclusion call arguments are removed.
7. The C++ equivalent of the threads, actions and functions are constructed along from the annotated tree, along with their forward declarations.
8. The global synchronization variables for use in the runtime (for example rendezvousCounter) are generated. This is currently handwritten.
9. The synchronized variables are found in the program declaration and the C++ equivalents are later determined. These are later put in the global scope of the C++ program.
10. The synchronizer function (waitForRendezvous) is constructed from the synchronized variables and the thread list.

Function List

- *generate* generates the final C++ code from the Fumurt AST
- *getAnnotatedTree* Returns an annotated version of the supplied AST. This version has the final C++ names for functions and their arguments and function calls
- *getCallsAnnotatedTreeInternal* returns an annotated version of the AST with final C++ names for function calls. Requires that function names have been annotated first
- *annotateFunctionCall* annotates a single function call
 - *annotateCallargs* annotates that function calls call arguments. Since call arguments can be function calls, this is often recursive.
 - *removeInclusions* removes inclusion arguments from functions, since these have no purpose in C++
- *indexlefts* indexes DefLhs's like in the checker, but with the annotated types.

- *findinscope* same as the version in the checker, but with annotated types.
- *getAnnotatedTreeInternal* returns an AST with with final C++ names for functions
- *getFunctionDeclarations* gets the functions, in C++, from the annotated AST
 - *actfunrecursivetranslate* gets function body and signature of a function corresponding to the arguments as well as all functions defined in the body of the definition.
 - *changeNamesToCppOnes* changes all identifiers which are arguments to a thread to their C++ names throughout the thread.
- *getFunctionSignature* constructs a C++ function signature from the arguments
 - *argtranslator* translates an argument as used in defining a function
- *typetranslator* translates Fumurt types to their C++ equivalents. Currently there are no user-defined types, so only basic types need to be translated.
- *callargTranslator* translates a call argument to the C++ equivalent
- *functioncalltranslator* translates function calls to C++ syntax
- *basicmathcalltranslator* translates calls to plus, minus, divide and multiply into +, -, /, and *
- *gettopthreadstatements* gets the C++ statements spawning the threads.
- *getprintlistdeclarations* gets the printList declarations. These are lists in which strings to be printed are kept. One for each thread
- *getmain* gets the main function. The main function only spawns the threads and then goes to sleep
- *getsynchronizerfunction* gets the mostly static and hand-written function that performs all actions during the communication phase
- *getGlobalSynchVariableDeclarations* gets the C++ declarations of the synchronized variables
- *getsynchronizedvariables* gets the definitions of the synchronized variables, so that they can later be used in *getGlobalSynchVariableDeclarations*

Classes The generator uses classes needed to annotate the AST, for example *class aDefinition(val leftside:aDefLhs, val rightside:aDefRhs)*. Existing AST classes are used unless extra information needs to be held or it is a parent of such a class. The most dramatic example is *class aDefLhs(val description:DefDescriptionT, val id:IdT, val cppid:IdT, val callingthread:String, val args:Option[Arguments], val returntype:TypeT)*. Here, we see the new C++ name, as well as which thread is meant to call the function. These are in the *Ast.scala* file.

5.6 Not Implemented

Considering the nature of languages, the amount left undone could very well be infinite. The following list are for things that make the current implementation feel incomplete.

- Loops
- User-defined types
- Boolean functions
- Comparison functions (beside equal)
- Exit function. This is not particularly important, as the systems Fumurt is made for are not expected to ever exit
- A check that only the thread with write rights to a synchronized variable is allowed to write to that variable
- Some checks for the *equal* function
- Any other IO than print to console

Chapter 6

Testing

6.1 Hello World

A simple repeating Hello World is written like this:

```
1 program helloworld:Nothing =
2 {
3     threadPrintHelloWorld()
4 }
5
6 thread threadPrintHelloWorld:Nothing =
7 {
8     actionPrint("Hello World\n")
9     threadPrintHelloWorld()
10 }
```

Which prints Hello World forever:

```
1 Hello World
2 Hello World
3 Hello World
4 Hello World
5 Hello World
6 Hello World
7 /*and so on*/
```

6.2 Multithreaded Hello World

A dualthreaded hello World is written like this:

```
1 program helloworld:Nothing =
2 {
3     threadPrintHello()
4     threadPrintWorld()
5 }
6
7 thread threadPrintWorld:Nothing =
```

```

8 {
9   actionPrint("World\n")
10  threadPrintWorld()
11 }
12
13 thread threadPrintHello:Nothing =
14 {
15   actionPrint("Hello ")
16   threadPrintHello()
17 }

```

Which also prints Hello World forever:

```

1 Hello World
2 Hello World
3 Hello World
4 Hello World
5 Hello World
6 Hello World
7 /*and so on*/

```

Note there is absolutely no performance benefits to dualthreading this, as the IO is sequential and this program does nothing but IO.

6.3 Synchronized Integer

Synchronized variables are the same in all threads, and mutations are published in the communicative phase.

```

1 program helloworld:Nothing =
2 {
3   synchronized variable synchronizedCounter:Integer =
4     {synchronized(variable=0, writer=threadC)}
5   threadA(synchronizedCounter)
6   threadB(synchronizedCounter)
7   threadC(synchronizedCounter)
8 }
9
10 thread threadA(synchronizedCounter:Integer):Nothing =
11 {
12   actionPrint(toString(synchronizedCounter))
13   actionPrint(" == ")
14   threadA(synchronizedCounter)
15 }
16
17 thread threadB(synchronizedCounter:Integer):Nothing =
18 {
19   actionPrint(toString(synchronizedCounter))
20   actionPrint("\n")
21   threadB(synchronizedCounter)
22 }
23
24 thread threadC(synchronizedCounter:Integer):Nothing =
25 {
26   actionMutate(newValue=plus(left=synchronizedCounter, right=1),
27     variable=synchronizedCounter)

```

```
26 threadC(synchronizedCounter)
27 }
```

And we can see that the number is consistent across threads:

```
1 0 == 0
2 1 == 1
3 2 == 2
4 3 == 3
5 4 == 4
6 5 == 5
7 6 == 6
8 7 == 7
9 8 == 8
10 9 == 9
11 /*and so on*/
```

6.4 Functions, Actions, Recursion and the Limitations of Integers

An example with a single thread, a square and a factorial function and an action is presented below.

```
1 program helloworld:Nothing =
2 {
3   threadA(d=1.0, i=1, actionPrintSquare=actionPrintSquare)
4 }
5
6 thread threadA(d:Double, i:Integer,
7   actionPrintSquare:Inclusion):Nothing =
8 {
9   function factorial(i:Integer):Integer =
10  {
11    if(condition=equal(left=1, right=i), then=1,
12      else=multiply(left=i, right=factorial(minus(left=i,
13        right=1))))
14  }
15  actionPrint("The factorial of ")
16  actionPrint(toString(i))
17  actionPrint(" is ")
18  actionPrint(toString(factorial(i)))
19  actionPrint(" ")
20  actionPrintSquare(d)
21  threadA(d = plus(left=d, right=0.5), i = plus(left=i, right=1),
22    actionPrintSquare=actionPrintSquare)
23 }
24
25 action actionPrintSquare(d:Double):Nothing =
26 {
27   function square(x:Double):Double = {multiply(left=x, right=x)}
28   actionPrint("The square of ")
29   actionPrint(toString(d))
30   actionPrint(" is ")
31   actionPrint(toString(square(d)))
32 }
```

```
28     actionPrint("\n")
29 }
```

When run, this example gives the following output:

```
1 The factorial of 1 is 1      The square of 1.000000 is 1.000000
2 The factorial of 2 is 2      The square of 1.500000 is 2.250000
3 The factorial of 3 is 6      The square of 2.000000 is 4.000000
4 The factorial of 4 is 24     The square of 2.500000 is 6.250000
5 The factorial of 5 is 120    The square of 3.000000 is 9.000000
6 The factorial of 6 is 720    The square of 3.500000 is 12.250000
7 The factorial of 7 is 5040   The square of 4.000000 is 16.000000
8 The factorial of 8 is 40320  The square of 4.500000 is 20.250000
9 The factorial of 9 is 362880 The square of 5.000000 is 25.000000
10 The factorial of 10 is 3628800 The square of 5.500000 is 30.250000
11 The factorial of 11 is 39916800 The square of 6.000000 is 36.000000
12 The factorial of 12 is 479001600 The square of 6.500000 is
    42.250000
13 The factorial of 13 is 1932053504 The square of 7.000000 is
    49.000000
14 The factorial of 14 is 1278945280 The square of 7.500000 is
    56.250000
15 The factorial of 15 is 2004310016 The square of 8.000000 is
    64.000000
16 The factorial of 16 is 2004189184 The square of 8.500000 is
    72.250000
17 The factorial of 17 is -288522240 The square of 9.000000 is
    81.000000
18 The factorial of 18 is -898433024 The square of 9.500000 is
    90.250000
19 The factorial of 19 is 109641728 The square of 10.000000 is
    100.000000
20 The factorial of 20 is -2102132736 The square of 10.500000 is
    110.250000
21 The factorial of 21 is -1195114496 The square of 11.000000 is
    121.000000
22 The factorial of 22 is -522715136 The square of 11.500000 is
    132.250000
23 The factorial of 23 is 862453760 The square of 12.000000 is
    144.000000
24 The factorial of 24 is -775946240 The square of 12.500000 is
    156.250000
25 The factorial of 25 is 2076180480 The square of 13.000000 is
    169.000000
26 The factorial of 26 is -1853882368 The square of 13.500000 is
    182.250000
27 The factorial of 27 is 1484783616 The square of 14.000000 is
    196.000000
28 The factorial of 28 is -1375731712 The square of 14.500000 is
    210.250000
29 The factorial of 29 is -1241513984 The square of 15.000000 is
    225.000000
30 The factorial of 30 is 1409286144 The square of 15.500000 is
    240.250000
31 The factorial of 31 is 738197504 The square of 16.000000 is
    256.000000
32 The factorial of 32 is -2147483648 The square of 16.500000 is
    272.250000
```

```

33 The factorial of 33 is -2147483648      The square of 17.000000 is
    289.000000
34 The factorial of 34 is 0      The square of 17.500000 is 306.250000
35 The factorial of 35 is 0      The square of 18.000000 is 324.000000
36 /*and so on*/

```

Here we see a problem with relying on integers of limited size. 32-bit integer is clearly inadequate for the factorial calculation. As for the eventual answer to the factorial calculation being zero, this seems to be a result of the C++ compiler's optimizations. No optimization gives the stack overflow we expect; running the binary results in a segmentation fault when compiled with GCC with `-O0` or `-O1` or Clang with `-O0`. Though there are problems with integer wrap-around and stack overflow, a recursive factorial function is a classic way to demonstrate the syntax of a language.

6.5 Full Program Test With C++ Intermediate

The following Fumurt code:

```

1  program p:Nothing =
2  {
3      synchronized variable synchronizedNumber:Integer =
          {synchronized(variable=0, writer=threadPrintHello)}
4      threadPrintHello(synchronizedNumber)
5      threadPrintWorld(synchronizedNumber)
6      threadPrintBaz(actionPrintFoo=actionPrintFoo, counter=0.0,
          integerIdentity=integerIdentity)
7  }
8
9  thread threadPrintWorld(synchronizedNumber:Integer):Nothing =
10 {
11     actionPrint("world ")
12     actionPrint(toString(synchronizedNumber))
13     threadPrintWorld(synchronizedNumber)
14 }
15
16 thread threadPrintHello(synchronizedNumber:Integer):Nothing =
17 {
18     actionPrint(toString(synchronizedNumber))
19     actionPrint(" Hello ")
20     actionMutate(variable=synchronizedNumber,
          newValue=plus(left=synchronizedNumber, right=1))
21     threadPrintHello(synchronizedNumber)
22 }
23
24 thread threadPrintBaz(actionPrintFoo:Inclusion,
          integerIdentity:Inclusion, counter:Double):Nothing =
25 {
26     action actionPrintBaz(counter:Double):Nothing =
27     {
28         actionPrint(" BAZ ")
29         actionPrint(toString(counter))
30         actionPrint(" ")
31     }

```

```

32     actionPrintBaz(counter)
33     actionPrintFoo(integerIdentity)
34     threadPrintBaz(counter=minus(right=1.0, left=counter),
35         actionPrintFoo=actionPrintFoo, integerIdentity=integerIdentity)
36 }
37
38 action actionPrintFoo(integerIdentity:Inclusion):Nothing =
39 {
40     action actionPrintFoo0:Nothing =
41     {
42         actionPrint(" F000 ")
43     }
44     actionPrint(" F00 ")
45     actionPrintFoo()
46     actionPrint(toString(integerIdentity(5)))
47     actionPrint(" ")
48     actionPrint(if(condition=true, then=toString(6), else=toString(3)))
49     actionPrint("\n")
50 }
51
52 function integerIdentity(x:Integer):Integer = {x}

```

The program gets compiled to the following C++11 code:

```

1  #include <iostream>
2  #include <thread>
3  #include <string>
4  #include <atomic>
5  #include <condition_variable>
6  #include <list>
7  #include <chrono>
8
9
10 #define NUMTOPTHREADS 3
11
12 [[noreturn]] static void threadPrintWorld();
13 [[noreturn]] static void threadPrintHello();
14 [[noreturn]] static void threadPrintBaz();
15 void actionPrintBaz(threadPrintBaz(double counter);
16 int integerIdentity(int x);
17 void actionPrintFoo(threadPrintBaz());
18 void actionPrintFoo0(threadPrintBazactionPrintFoo());
19
20 static int synchronizedNumber = 0;
21 static int writeSynchronizedNumber = 0;
22 static std::list<std::string> printthreadPrintHello;
23 static std::list<std::string> printthreadPrintWorld;
24 static std::list<std::string> printthreadPrintBaz;
25 static std::atomic<int> rendezvousCounter;
26 static std::mutex rendezvousSyncMutex;
27 static std::condition_variable cv;
28 static double threadPrintBaz$counter;
29 static void waitForRendezvous(std::string name)
30 {
31     std::unique_lock<std::mutex> lk(rendezvousSyncMutex);
32     ++rendezvousCounter;
33     if(rendezvousCounter.load() < NUMTOPTHREADS)

```

```

34     {
35         cv.wait(lk);
36     }
37     else if (rendezvousCounter.load() == NUMTOPTHREADS)
38     {
39         while(!printthreadPrintHello.empty()){
40             std::cout << printthreadPrintHello.front();
41             printthreadPrintHello.pop_front();
42         }
43         while(!printthreadPrintWorld.empty()){
44             std::cout << printthreadPrintWorld.front();
45             printthreadPrintWorld.pop_front();
46         }
47         while(!printthreadPrintBaz.empty()){
48             std::cout << printthreadPrintBaz.front();
49             printthreadPrintBaz.pop_front();
50         }
51         synchronizedNumber = writeSynchronizedNumber;
52
53         {
54             rendezvousCounter.store(0);
55             cv.notify_all();
56         }
57     }
58     else
59     {
60         std::cout << "error in wait for " << name << ". Rendezvouscounter
61             out of bounds. RendezvousCounter = " <<
62             rendezvousCounter.load() << "\n";
63         exit(0);
64     }
65 }
66
67 [[noreturn]] static void threadPrintWorld()
68 {while(true)
69 {
70     printthreadPrintWorld.push_back("world ");
71     printthreadPrintWorld.push_back(std::to_string(synchronizedNumber));
72     waitforRendezvous("threadPrintWorld");
73     continue;
74 }
75 }
76
77 [[noreturn]] static void threadPrintHello()
78 {while(true)
79 {
80     printthreadPrintHello.push_back(std::to_string(synchronizedNumber));
81     printthreadPrintHello.push_back(" Hello ");
82     writeSynchronizedNumber = (synchronizedNumber + 1);
83     waitforRendezvous("threadPrintHello");
84     continue;
85 }
86 }
87
88 [[noreturn]] static void threadPrintBaz()

```

```

89 {while(true)
90 {
91     actionPrintBaz$threadPrintBaz(threadPrintBaz$counter);
92     actionPrintFoo$threadPrintBaz();
93     waitForRendezvous("threadPrintBaz");
94     threadPrintBaz$counter = (threadPrintBaz$counter - 1.0);
95
96     continue;
97 }
98 }
99
100 void actionPrintBaz$threadPrintBaz(double counter)
101 {
102     printthreadPrintBaz.push_back(" BAZ ");
103     printthreadPrintBaz.push_back(std::to_string(counter));
104     printthreadPrintBaz.push_back(" ");
105 }
106
107 int integerIdentity$(int x)
108 {
109     return x;
110 }
111
112 void actionPrintFoo$threadPrintBaz()
113 {
114     printthreadPrintBaz.push_back(" FOO ");
115     actionPrintFoo$threadPrintBazactionPrintFoo();
116     printthreadPrintBaz.push_back(std::to_string(integerIdentity$(5)));
117     printthreadPrintBaz.push_back(" ");
118     printthreadPrintBaz.push_back(std::to_string(6));
119     printthreadPrintBaz.push_back("\n");
120 }
121
122 void actionPrintFoo$threadPrintBazactionPrintFoo()
123 {
124     printthreadPrintBaz.push_back(" F000 ");
125 }
126
127
128 int main()
129 {
130     rendezvousCounter.store(0);
131
132     threadPrintBaz$counter = 0.0;
133     std::thread tthreadPrintHello (threadPrintHello);
134     std::thread tthreadPrintWorld (threadPrintWorld);
135     std::thread tthreadPrintBaz (threadPrintBaz);
136     while(true)
137     {
138         std::this_thread::sleep_for(std::chrono::seconds(1));
139     }
140 }

```

When run in a terminal, this results in the following output:

```

1 0 Hello world 0 BAZ 0.000000 F00 F000 5 6
2 1 Hello world 1 BAZ -1.000000 F00 F000 5 6
3 2 Hello world 2 BAZ -2.000000 F00 F000 5 6

```



```

4 3 Hello world 3  BAZ -3.000000    F00    F000  5  6
5 4 Hello world 4  BAZ -4.000000    F00    F000  5  6
6 /*and so on...*/

```

6.6 Error messages

Error messages are useful to detect errors in the program at compile time. Changing the source in 6.5 to the following erroneous program allow us to test them:

```

1 program p:Nothing =
2 {
3   synchronized variable synchronizedNumber:Integer =
4     {synchronized(variable=0, writer=threadPrintHello)}
5     threadPrintWorld(synchronizedNumber)
6     threadPrintLol(actionPrintFoo=integerIdentity,
7       integerIdentity=integerIdentityyy)
8 }
9
10 thread threadPrintWorld(synchronizedNumber:Integer):Nothing =
11 {
12   actionPrint("world ")
13   actionPrint(toString(synchronizedNumber))
14   threadPrintWorld(synchronizedNumber)
15 }
16
17 thread threadPrintHello(synchronizedNumber:Integer):Nothing =
18 {
19   actionPrint(synchronizedNumber)
20   actionPrint(" Hello ")
21   actionMutate(variable=synchronizedNumber,
22     newValue=plus(left=synchronizedNumber, right=1))
23   threadPrintHello(synchronizedNumber)
24 }
25
26 thread threadPrintLol(actionPrintFoo:Inclusion,
27   integerIdentity:Inclusion):Nothing =
28 {
29   action actionPrintLol:Nothing =
30   {
31     actionPrint(" LOL ")
32   }
33
34   actionPrintLol()
35   actionPrintFoo(integerIdentity)
36   threadPrintLol(actionPrintFoo=actionPrintFoo,
37     integerIdentity=integerIdentity)
38 }
39
40 function printFoo(integerIdentity:Inclusion):Nothing =
41 {
42   action actionPrintFooo:Nothin =
43   {
44     actionPrint(" F000 ")
45   }
46   actionPrint(" F00 ")

```

```

42     actionPrintFoo()
43     actionPrint(toString(integerIdentity(5.0)))
44     actionPrint(" ")
45     actionPrint(if(condition=0, then=6, else=toString(3)))
46     actionPrint(toString(if(condition=false, then=6, else=3)))
47     actionPrint("\n")
48 }
49
50 function integerIdentity(x:Integer):Integer = {multiply(left=x,
    right=1.0)}

```

This causes the Fumurt checker to produce the following errors:

```

1 0.0: thread threadPrintHello is declared but not used
2 global position
3 ^
4
5 5.33: Passed inclusion must be the same as the one referenced inside
6 the function
7     threadPrintLol(actionPrintFoo=integerIdentity,
8         integerIdentity=integerIdentityyy)
9
10 5.66: integerIdentityyy not found
11     threadPrintLol(actionPrintFoo=integerIdentity,
12         integerIdentity=integerIdentityyy)
13
14
15 17.15: Expected type String. Got Integer
16     actionPrint(synchronizedNumber)
17
18
19
20 31.3: actionPrintFoo not found
21     actionPrintFoo(integerIdentity)
22
23
24
25 32.33: actionPrintFoo not found
26     threadPrintLol(actionPrintFoo=actionPrintFoo,
27         integerIdentity=integerIdentity)
28
29
30 39.5: Expected return type Nothin. Got Nothing
31     actionPrint(" F000 ")
32
33
34
35 37.3: actions cannot be defined in functions
36     action actionPrintFoo:Nothin =
37
38
39
40 42.3: Expected return type Nothing. Got Nothin
41     actionPrintFoo()

```

```

42
43
44
45 43.40: Call argument type should be Integer. Call argument type was
      Double
46     actionPrint(toString(integerIdentity(5.0)))
47
48
49
50 45.28: Call argument type should be Boolean. Call argument type was
      Integer
51     actionPrint(if(condition=0, then=6, else=toString(3)))
52
53
54
55 45.36: Call argument type should be String. Call argument type was
      Integer
56     actionPrint(if(condition=0, then=6, else=toString(3)))
57
58
59
60 50.48: This call to multiply returns a Double not an Integer
61 function integerIdentity(x:Integer):Integer = {multiply(left=x,
      right=1.0)}
62
63
64
65 13 errors found

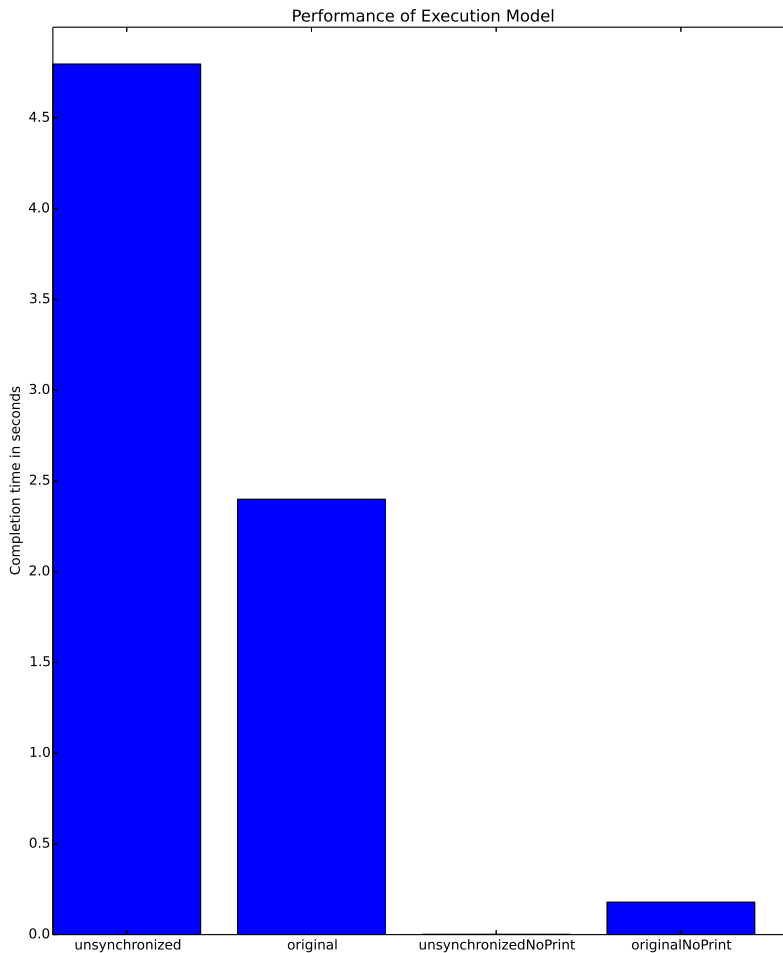
```

6.7 Performance

In order to understand the cost of the synchronization in the execution model, a test was performed. The C++ code generated in 6.5 was modified to exit when `synchronizedNumber` was equal to or bigger than 20000. Let this be the *original*. Then all synchronization mechanisms was removed. Let this be the *unsynchronized*. Then the print statements of both was removed, as if the original Fumurt program had had no calls to `actionPrint`. Let these be *originalNoPrint* and *unsynchronizedNoPrint*. The times taken until completion was then measured using the Unix *time* utility. The optimizations used were “-O3 -march=native” on an Intel i5-2500 CPU. The results were very interesting:

Code	Time until completion (in seconds)
original	2.399
unsynchronized	4.797
originalNoPrint	0.179
unsynchronizedNoPrint	0.002

The same results are visualized in the plot below:



Two things can be concluded from these measurements:

1. The execution model incurs considerable cost
2. The execution model can achieve superior performance compared to an unsynchronized model when the program is dominated by access to terminal output. One may speculate that this is due to resource contention and applies equally to all inherently sequential IO

Chapter 7

Conclusion, Discussion and Further Work

7.1 Conclusion

During the writing of this thesis, a deterministic multithreaded language has been designed and a compiler has been built for it. In this report it has been shown that creating a programming language that eliminates almost all of the difficulties of multithreaded programming is possible, while maintaining some of the architectural and performance benefits of multithreading. Fumurt also presents some new ideas regarding the ways in which code should be structured, possibly making it easier to maintain large software projects. Yet Fumurt is not near being a usable language, and many questions remain unanswered.

7.2 Discussion

In hindsight, the code generator could have been better written. Adding an additional two steps with the annotator was a fairly late decision, and the architecture of the module suffered for it. There's also numerous bugs and lacking features, as well as corner cases where the appropriate behavior simply has not been determined. The various features of the language included with the intent of easing the maintenance of large programs are not rooted in empirical studies, which is clearly unfortunate. In the case where a computational phase runs for a long time, the IO buffers may grow to be too large to be stored in memory. While this is not an issue for desktop and laptop computers where filling up the memory takes so long that the program's unresponsiveness is the bigger issue, it's a bigger problem for microcontrollers. Fixing this problem means that the sequential IO and inter-thread communication abstraction which the programming language provides can in extreme cases require that the execution itself becomes sequential. It seems intuitively possible that this is simply a necessity when providing such an abstraction.

In situations where performance is more important than predictability, mechanisms need to be provided to the programmer so that determinism requirements can be relaxed. Similarly, some kinds of recursion have memory use requirements which are hard to optimize away. The correct way to handle this is yet to be determined.

More fundamentally, the literature concerning multithreading seems divided over what should be required to be deterministic by the language and what should require programmer intervention if a deterministic sequence is required. It is unclear whether this thesis has the best approach.

7.3 Suggestions for Future Work

It is common for programming languages to need a decade of intensive development by several contributors before it is ready for serious usage. It is therefore not hard to come up with ways in which Fumurt could be improved. For ideas, see section 4.6 and 5.6. But not all improvements to Fumurt are of academic interest; much of the work is simply implementation of pretty mundane things. Improvements to the execution model might be more interesting. There are many ideas in section 4.6 about how the model might be refined. Investigating solutions to employing deterministic Fumurt or Fumurt-like systems while accommodating hardware faults and distributed systems is another possibility. An Erlang/OTP system might be able to serve as a supervisor for several networked systems running deterministic code. Lastly, there seems to be little empirical work when it comes to programming language design. Performing empirical studies among programmers investigating what programming language ideas are actually helpful seems like a good idea.

Bibliography

- [1] Deterministic parallel java: Current design. http://dpj.cs.illinois.edu/DPJ/Current_Design.html. Accessed: 2015-06-03.
- [2] Scala parser combinator documentation. <http://www.scala-lang.org/files/archive/api/2.11.x/scala-parser-combinators/index.html#scala.util.parsing.combinator.Parsers>. Accessed: 2015-05-18.
- [3] Scala parser combinator positional documentation. <http://www.scala-lang.org/files/archive/api/2.11.x/scala-parser-combinators/index.html#scala.util.parsing.input.Positional>. Accessed: 2015-05-18.
- [4] University college london suggested project report structure for msc computer graphics, vision and imaging. http://www.cs.ucl.ac.uk/teaching_learning/msc_cgvi/projects/project_report_structure/. Accessed: 2015-05-20.
- [5] BERGAN, T., ANDERSON, O., DEVIETTI, J., CEZE, L., AND GROSSMAN, D. Coredet: a compiler and runtime system for deterministic multithreaded execution. In *ACM SIGARCH Computer Architecture News* (2010), vol. 38, ACM, pp. 53–64.
- [6] BERRY, G., AND GONTHIER, G. The esterel synchronous programming language: Design, semantics, implementation. *Science of computer programming* 19, 2 (1992), 87–152.
- [7] BOCCHINO, R., ADVE, V., ADVE, S., AND SNIR, M. Parallel programming must be deterministic by default. In *Proceedings of the First USENIX conference on Hot topics in parallelism* (2009), pp. 4–4.
- [8] BÖHM, C., AND JACOPINI, G. Flow diagrams, turing machines and languages with only two formation rules. *Communications of the ACM* 9, 5 (1966), 366–371.
- [9] BORTHAKUR, D., GRAY, J., SARMA, J. S., MUTHUKKARUPPAN, K., SPIEGELBERG, N., KUANG, H., RANGANATHAN, K., MOLKOV, D., MENON, A., RASH, S., ET AL. Apache hadoop goes realtime at facebook. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data* (2011), ACM, pp. 1071–1080.

- [10] COOPER, K., AND TORCZON, L. *Engineering a compiler*. Elsevier, 2011.
- [11] DEAN, J., AND GHEMAWAT, S. Mapreduce: simplified data processing on large clusters. *Communications of the ACM* 51, 1 (2008), 107–113.
- [12] DEVIETTI, J., NELSON, J., BERGAN, T., CEZE, L., AND GROSSMAN, D. Redc: a relaxed consistency deterministic computer. *ACM SIGPLAN Notices* 46, 3 (2011), 67–78.
- [13] FROST, R., AND LAUNCHBURY, J. Constructing natural language interpreters in a lazy functional language. *The Computer Journal* 32, 2 (1989), 108–121.
- [14] HALL, C., HAMMOND, K., PARTAIN, W., JONES, S. L. P., AND WADLER, P. The glasgow haskell compiler: a retrospective. In *Functional Programming, Glasgow 1992*. Springer, 1993, pp. 62–71.
- [15] HERLIHY, M., AND MOSS, J. E. B. *Transactional memory: Architectural support for lock-free data structures*, vol. 21. ACM, 1993.
- [16] HEWITT, C., BISHOP, P., AND STEIGER, R. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd international joint conference on Artificial intelligence (1973)*, Morgan Kaufmann Publishers Inc., pp. 235–245.
- [17] HOARE, C. A. R. Communicating sequential processes. *Communications of the ACM* 21, 8 (1978), 666–677.
- [18] LIU, T., CURTSINGER, C., AND BERGER, E. D. Dthreads: efficient deterministic multithreading. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (2011)*, ACM, pp. 327–336.
- [19] LU, K., ZHOU, X., BERGAN, T., AND WANG, X. Efficient deterministic multithreading without global barriers. In *Proceedings of the 19th ACM SIGPLAN symposium on Principles and practice of parallel programming (2014)*, ACM, pp. 287–300.
- [20] MERRIFIELD, T., DEVIETTI, J., AND ERIKSSON, J. High-performance determinism with total store order consistency. In *Proceedings of the Tenth European Conference on Computer Systems (2015)*, ACM, p. 31.
- [21] MEYEROVICH, L. A., AND RABKIN, A. S. Empirical analysis of programming language adoption. In *ACM SIGPLAN Notices (2013)*, vol. 48, ACM, pp. 1–18.
- [22] MILNE, I., AND ROWE, G. Difficulties in learning and teaching programming - views of students and tutors. *Education and Information technologies* 7, 1 (2002), 55–66.
- [23] OLSZEWSKI, M., ANSEL, J., AND AMARASINGHE, S. Kendo: efficient deterministic multithreading in software. *ACM Sigplan Notices* 44, 3 (2009), 97–108.

- [24] SUTTER, H. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobbs's journal* 30, 3 (2005), 202–210.
- [25] WIRTH, N. Extended backus-naur form (ebnf). *ISO/IEC 14977* (1996), 2996.

Appendix A

System manual

To avoid confusion when discussing compiling the compiler, the Fumurt compiler will be referred to as “the program”.

To compile this code you need the Simple Build Tool (SBT), available at <http://www.scala-sbt.org/>. SBT will download the dependencies required including the compiler and the parser combinator library. It will also allow you to run the program. Depending on the way you install SBT and on which platform, you may have to install a Java runtime environment in order to run SBT

To compile the code using SBT, a certain directory hierarchy is required. The directory in which you run SBT must be the same directory that the “build.sbt” file and “src” directory is in. “build.sbt” holds dependency and compilation options for SBT. The “src” directory holds all the source code for the project in a structure. Since there’s only Scala code in this project, the source files shall be in “src/main/scala”.

Once SBT is installed and the directory structure conforms to SBT rules, SBT can be started in the directory by using the “sbt” command in a terminal in the directory holding “build.sbt” file and “src”. SBT will then download the files needed to compile and run the program. This usually takes a long while, depending on your Internet connection. Once this is done, SBT will present a command prompt. The program can then be compiled and run from this SBT command prompt using the “run [name of Fumurt file]” command. The compilation (of the compiler) also usually takes a while. Note that the Fumurt source file must be in the same directory that you launch SBT in, as the the Fumurt compiler does not handle file paths in its input.

The Fumurt compiler uses GCC, and GCC must therefore be installed. If GCC is not installed, the user may compile the generated C++ from the generated “generated.cpp” file. The options “-pthread” and “-std=c++11” are required when using Clang/GCC on Linux, but not using the Microsoft Visual C++ compiler on Windows.

If everything goes well, the output will be a binary executable named “generated”, with “-O3” and “march=native” options.

Example:

```
1 $ ls
2 build.sbt src test.fumurt
3 $ sbt
4 [info] /*current sbt state*/
5 > run test.fumurt
6 [info] Running fumurtCompiler.Main test.fumurt
7 [success] Total time: 2 s, completed May 29, 2015 5:42:18 PM
8 > /*ctrl+c*/
9 $ ls
10 generated.cpp build.sbt generated src test.fumurt
11 $ ./generated
12 /*program output here*/
```

Appendix B

User manual

You need to have Scala installed (<http://www.scala-lang.org/download/>) to run the Fumurt compiler from compiled bytecode. The current directory must be the one *above* the “.class” bytecode files. Because the starting point of the program is function “main” in object “Main” in package “fumurtCompiler”, the folder containing the bytecode files must be “fumurtCompiler” (i.e. the name of the package), and the command to run must be “scala fumurtCompiler.Main [fumurt source file here]”.

The Fumurt compiler uses GCC, and GCC must therefore be installed. If GCC is not installed, the user may compile the generated C++ from the generated “generated.cpp” file. The options “-pthread” and “-std=c++11” are required when using Clang/GCC on Linux, but not using the Microsoft Visual C++ compiler on Windows.

Example:

```
1 $ ls
2 fumurtCompiler  test.fumurt
3 $ ls fumurtCompiler
4 aCallarg.class
5 FumurtParser$$anonfun$subsequentArgsParser$1$$anonfun$apply$11.class
6 ActionT.class
7 FumurtParser$$anonfun$subsequentArgsParser$1.class
8 ActionT$.class
9 FumurtParser$$anonfun$subsequentArgsParser$2.class
10 /*more bytecode files here*/
11 $
12 $ scala fumurtCompiler.Main test.fumurt
13 $ ./generated
14 /*program output here*/
```

Appendix C

Code listing

Comments and commented-out diagnostics are left in, as they might prove useful to anyone improving on the work.

C.1 build.sbt

```
1 name := "solution"
2
3 organization := "NTNU ITK"
4
5 version := "0.1.0"
6
7 scalaVersion := "2.11.6"
8
9 scalacOptions += Seq("-feature", "-optimise", "-Xlint",
10     "-Xfatal-warnings", "-deprecation", "-Ywarn-unused",
11     "-Ywarn-infer-any", "-Ywarn-unused-import", "-Ywarn-dead-code",
12     "-Ywarn-inaccessible", "-Ywarn-numeric-widen",
13     "-Ywarn-nullary-override", "-Ywarn-nullary-unit",
14     "-Ywarn-adapted-args")
15
16 libraryDependencies += Seq( "org.scala-lang.modules" %%
17     "scala-parser-combinators" % "1.0.3")
```

C.2 Main.scala

```
1 package fumurtCompiler
2
3 import scala.io.Source._
4 import scala.util.parsing.input.Positional
5
6 object CompileTypeOption extends Enumeration
7 {
8     type CompileTypeOption = Value
```

```

9   val compiledToGo, compiledToC, compiledToCpp, interpreted = Value
10  }
11
12  import CompileTypeOption._
13
14  object Main
15  {
16  def main(args: Array[String]) :Unit = {
17    if(args.length <1)
18      {
19        println("no file found in arguments\n")
20      }
21    else
22      {
23      val parts = args(0).split("\\.")
24      if(parts.length==2)
25        {
26          if(parts(1)=="fumurt")
27            {
28              compile(getOptions(args.drop(1), args(0)))
29            }
30          else
31            {
32              println("unknown file ending: " + parts(1) + "\n")
33            }
34          }
35        else
36          {
37            println("too many arguments\n")
38          }
39        }
40      }
41
42  def getOptions(args:Array[String],file:String): Options =
43  {
44    //println(args.toString)
45    new Options(CompileTypeOption.interpreted, true, file)
46  }
47
48  def compile(opts:Options):Unit =
49  {
50    //println("Now compiling!")
51    val sourcestring = fromFile(opts.file).mkString
52    FumurtScanner.scan(sourcestring) match
53    {
54      case Left(error) => println("Error in scanner: " +
55        error.toString)
56      case Right(tokens) =>
57      {
58        //println("successful scan. Tokens: "+tokens.toString+"\n")
59        FumurtParser.parse(tokens) match
60        {
61          case Left(error) => println("Error in parser: " +
62            error.toString)
63          case Right(ast) =>
64          {
65            //println("Success in parser: " + ast.toString)

```

```

64 FumurtTypeChecker.check(ast) match
65 {
66   case Some(errors) =>
67     {
68       errors.map(x=>println(x))
69       val errornum:String = errors.length match
70         {
71           case 1 => "one"
72           case 2 => "two"
73           case 3 => "three"
74           case 4 => "four"
75           case 5 => "five"
76           case 6 => "six"
77           case 7 => "seven"
78           case 8 => "eight"
79           case 9 => "nine"
80           case x => x.toString
81         }
82       val singularplural:String = if(errors.length==1){
83         error}else{" errors"}
84       println(errornum.capitalize + singularplural + "
85         found")
86     }
87   case None =>
88     {
89       //println("\nNo errors in checker")
90       val generatedcode = FumurtCodeGenerator.generate(ast)
91       //println("\ncode generated: \n" + generatedcode)
92       import java.nio.file.{Paths, Files}
93       import java.nio.charset.StandardCharsets
94       val outname = "generated"
95       val fileending = ".cpp"
96       Files.write(Paths.get("./"+outname+fileending),
97         generatedcode.getBytes(StandardCharsets.UTF_8))
98       val options = " -pthread -std=c++11 -O3 -march=native"
99       //println("\n\n===Starting GCC cpp compilation===")
100      //println("options = " + options)
101      import scala.sys.process._
102      val command = "g++ " + outname + fileending + options
103      + " -o " + outname
104      //println(command)
105      if( System.getProperty( "os.name"
106        ).startsWith("Windows") )
107      {
108        println("OS identified as Windows. Please use the
109          Microsoft Visual C++ compiler (included in
110          Visual Studio) to compile the generated
111          \"generated.cpp\" file")
112      }
113      else
114      {
115        (command).!
116      }
117    }
118  }
119 }

```

```

113     }
114   }
115
116   }
117
118 }
119
120 class Options(val compileTypeOption:CompileTypeOption, val
      debug:Boolean, val file:String)

```

C.3 Ast.scala

```

1 package fumurtCompiler
2
3 import scala.util.parsing.input.Positional
4
5
6 abstract class Token() extends Positional
7 abstract class DefDescriptionT() extends Token
8 abstract class BasicValueT() extends Token
9 abstract class SyntaxT() extends Token
10
11 case class EmptyT() extends Token
12 case class TrueT() extends BasicValueT {override def toString =
13   "true"}
13 case class FalseT() extends BasicValueT {override def toString =
14   "false"}
14 case class ProgramT() extends DefDescriptionT {override def toString
15   = "program"}
15 case class ActionT() extends DefDescriptionT {override def toString =
16   "action"}
16 case class ThreadT() extends DefDescriptionT {override def toString =
17   "thread"}
17 case class FunctionT() extends DefDescriptionT {override def toString
18   = "function"}
18 case class ValueT() extends DefDescriptionT {override def toString =
19   "value"}
19 case class SynchronizedVariableT() extends DefDescriptionT {override
20   def toString = "synchronized variable"}
20 case class OpenParenthesisT() extends SyntaxT {override def toString
21   = "\"(\""}
21 case class CloseParenthesisT() extends SyntaxT {override def toString
22   = "\"}\""}
22 case class OpenCurlyBracketT() extends SyntaxT {override def toString
23   = "\"{""}
23 case class CloseCurlyBracketT() extends SyntaxT {override def
24   toString = "\"}\""}
24 case class DoubleT(val value:Double) extends BasicValueT {override
25   def toString = "decimal number"}
25 case class IntegerT(val value:Int) extends BasicValueT {override def
26   toString = "integer"}
26 case class EqualT() extends SyntaxT {override def toString = "\"=\""}
27 case class ColonT() extends SyntaxT {override def toString = "\":""}
28 case class CommaT() extends SyntaxT {override def toString = "\",\""}

```



```

29 | case class NewlineT() extends SyntaxT {override def toString =
    | "newline"}
30 | case class IdT(val value:String) extends Token {override def toString
    | = "identifier(\""+value+"\")}
31 | case class TypeT(val value:String) extends Token {override def
    | toString = "type(\""+value+"\")}
32 | case class StringT(val value:String) extends BasicValueT {override
    | def toString = "string"}
33 | case class SpaceT() extends SyntaxT
34 | case class DummyT() extends Token
35 | case class EofT() extends SyntaxT {override def toString = "end of
    | file"}
36 |
37 |
38 |
39 |
40 |
41 |
42 |
43 |
44 |
45 |
46 |
47 |
48 |
49 |
50 |
51 |
52 |
53 |
54 | class Expression() extends Positional
55 | trait Callarg extends Positional
56 | trait Statement extends Expression
57 | trait BasicValueStatement extends Statement with Callarg with
    | aCallarg with aStatement
58 |
59 | case class Definition(val leftside:DefLhs, val rightside:DefRhs)
    | extends Expression
60 | case class DefLhs(val description:DefDescriptionT, val id:IdT, val
    | args:Option[Arguments], val returntype:TypeT)
61 | /*case class Arguments(val id:IdT, val typestr:TypeT, val
    | args2:Option[Arguments2])
62 | case class Arguments2(val id:IdT, val typestr:TypeT, val
    | args2:Option[Arguments2])*/
63 | case class Arguments(val args:List[Argument])
64 | case class Argument(val id:IdT, val typestr:TypeT)
65 | case class DefRhs(val expressions:List[Expression] )
66 | case class Empty();
67 | case class DefDescription(val value:Token)
68 | case class NamedCallarg(id:IdT, argument:Callarg) //extends Callarg
69 | case class NamedCallargs(val value:List[NamedCallarg])
70 | case class NoArgs() extends Callarg with aCallarg
71 |
72 | case class StringStatement(val value:String) extends
    | BasicValueStatement
73 | case class IntegerStatement(val value:Int) extends BasicValueStatement

```

```

74 case class DoubleStatement(val value:Double) extends
    BasicValueStatement
75 case class TrueStatement() extends BasicValueStatement
76 case class FalseStatement() extends BasicValueStatement
77 case class IdentifierStatement(val value:String) extends Statement
    with Callarg with aCallarg with aStatement
78 case class FunctionCallStatement(val functionidentifier:String, val
    args:Either[Callarg,NamedCallargs]) extends Statement with Callarg
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95 trait aExpression
96 trait aCallarg extends Callarg with aStatement
97 trait aStatement extends aExpression
98
99 case class aDefinition(val leftside:aDefLhs, val rightside:aDefRhs)
    extends aExpression
100 case class aDefLhs(val description:DefDescriptionT, val id:IdT, val
    cppid:IdT, val callingthread:String, val args:Option[aArguments],
    val returntype:TypeT)
101 case class aArguments(val args>List[aArgument])
102 case class aArgument(val id:IdT, cppid:IdT, val typestr:TypeT)
103 case class aDefRhs(val expressions>List[aExpression] )
104 case class aNamedCallarg(id:IdT, argument:aCallarg) //extends Callarg
105 case class aNamedCallargs(val value>List[aNamedCallarg])
106
107 case class aFunctionCallStatement(val functionidentifier:String, val
    cppfunctionidentifier:String, val
    args:Either[aCallarg,aNamedCallargs], val returntype:String)
    extends aStatement with aCallarg

```

C.4 Scanner.scala

```

1
2
3 package fumurtCompiler
4
5 import scala.language.implicitConversions
6 import scala.util.parsing.combinator.RegexParsers
7 import scala.util.matching.Regex
8 import scala.language.postfixOps

```

```

9  //import scala.util.parsing.combinator.lexical._
10 import scala.util.parsing.input.Positional
11
12 object FumurtScanner extends RegexParsers /*with Parsers*/
13 {
14   override val skipWhitespace = false
15
16   def scan(in:String):Either[NoSuccess, List[Token]] =
17   {
18     //println(in)
19
20     parseAll((scanInternal*), in) match
21     {
22       case Success(result, _) =>
23       {
24         val tokens = result.filter(x=>x match{case SpaceT() => false;
25           case _ => true}) :+ EofT()
26         Right(tokens)
27       }
28       case f:Failure => Left(f)
29       case e:Error => Left(e)
30       //case Failure(message, reader) => Left(new
31         FumurtError(reader.pos, "Failure: "+message+"\n" +
32         in.lines.toList(reader.pos.line) +"\n"))
33       //case Error(message, _) => Left(new FumurtError(Global, "Error:
34         " + message, ""))
35     }
36   }
37
38   def spaceParser:Parser[SpaceT] = positioned( new Regex(" ") ^^
39     {x => /*println("scanned space");*/SpaceT() } )
40   def programStrParser: Parser[ProgramT] = positioned( new
41     Regex("program ") ^^ {x => /*println("scanned program
42     "+x.toString);*/ProgramT() } )
43   def functionParser: Parser[FunctionT] = positioned( new
44     Regex("function ") ^^ {x => /*println("scanned function
45     "+x.toString);*/FunctionT() } )
46   def threadParser: Parser[ThreadT] = positioned( new Regex("thread
47     ") ^^ {x => /*println("scanned thread
48     "+x.toString);*/ThreadT() } )
49   def synchronizedVariableParser: Parser[SynchronizedVariableT] =
50     positioned(new Regex("synchronized variable ") ^^ {x =>
51     /*println("scanned synchronized variable "+x.toString);*/
52     SynchronizedVariableT()})
53   def valueParser: Parser[ValueT] = positioned( new Regex("value ")
54     ^^ {x => /*println("scanned unsafe value
55     "+x.toString);*/ValueT() } )
56   def actionParser: Parser>ActionT] = positioned( new Regex("action
57     ") ^^ {x => /*println("scanned action
58     "+x.toString);*/ActionT() } )
59   def trueParser: Parser[TrueT] = positioned( new Regex("true") ^^ {x
60     => /*println("scanned true "+x.toString);*/TrueT() } )
61   def falseParser: Parser[FalseT] = positioned( new Regex("false") ^^
62     {x => /*println("scanned false "+x.toString);*/FalseT() } )

```

```

46 def openParenthesisParser: Parser[OpenParenthesisT] = positioned(
    new Regex("""\""")) ^^ {x => /*println("scanned (
47 "+x.toString);*/OpenParenthesisT() )
48 def closeParenthesisParser: Parser[CloseParenthesisT] = positioned(
    new Regex("""\""")) ^^ {x => /*println("scanned )
49 "+x.toString);*/CloseParenthesisT() )
48 def openCurlyBracketParser: Parser[OpenCurlyBracketT] = positioned(
    new Regex("""\""")) ^^ {x => /*println("scanned {
50 "+x.toString);*/OpenCurlyBracketT() )
49 def closeCurlyBracketParser: Parser[CloseCurlyBracketT] =
    positioned( new Regex("""\""")) ^^ {x => /*println("scanned }
51 "+x.toString);*/CloseCurlyBracketT() )
50 def doubleParser: Parser[DoubleT] = positioned( new
    Regex("""[-+]?[0-9]*\.[0-9]+""") ^^ {x => /*println("scanned
52 double "+x.toString);*/DoubleT(x.toDouble)} )
51 def intParser: Parser[IntegerT] = positioned( new
    Regex("""[-+]?[0]([1-9]\d*)""") ^^ {x => /*println("scanned
53 integer "+x.toString);*/IntegerT(x.toInt)} )
52 def equalParser: Parser[EqualT] = positioned( new Regex("=") ^^ {x
53 => /*println("scanned = "+x.toString);*/EqualT() )
53 def colonParser: Parser[ColonT] = positioned( new Regex(":") ^^ {x
54 => /*println("scanned : "+x.toString);*/ColonT() )
54 def commaParser: Parser[CommaT] = positioned( new Regex(",") ^^ {x
55 => /*println("scanned , "+x.toString);*/CommaT() )
55 //def emptyParser: Parser[EmptyT] = new Regex("") ^^ {x =>
    println("scanned empty");EmptyT() }
56 def newlineParser: Parser[NewlineT] = positioned( new Regex("\n")
    ^^ {x => /*println("scanned newline ");*/NewlineT() } )
57 def idParser: Parser[IdT] = positioned( new
    Regex("[a-z]+[a-zA-Z]*") ^^ {x => /*println("scanned id
58 "+x.toString);*/IdT(x.toString)} )
59 def stringParser: Parser[StringT] = positioned( new
    Regex("""("[^"]*"")""") ^^ {x => /*println("scanned string
60 "+x.toString);*/StringT(x.toString)} )
61 def typeParser: Parser[TypeT] = positioned( new
    Regex("[A-Z][a-zA-Z]*") ^^ {x => /*println("scanned type
62 "+x.toString);*/TypeT(x.toString)} )
63
64 def scanInternal: Parser[Token] =
65 {
66 (
67     spaceParser           |
68     programStrParser      |
69     threadParser          |
70     actionParser          |
71     synchronizedVariableParser |
72     functionParser        |
73     trueParser            |
74     falseParser           |
75     openParenthesisParser |
76     closeParenthesisParser |
77     openCurlyBracketParser |
78     closeCurlyBracketParser |
79     doubleParser          |
80     intParser              |
81     equalParser            |

```

```

80     colonParser           |
81     commaParser          |
82     //emptyParser        |
83     newlineParser        |
84     stringParser         |
85     idParser             |
86     typeParser           |
87   )
88 }
89
90
91
92 }
```

C.5 Parser.scala

```

1  package fumurtCompiler
2
3  //import scala.util.parsing._
4  import scala.language.postfixOps
5  import scala.language.implicitConversions
6  import scala.util.parsing.input._
7  import scala.util.parsing.combinator._
8  //import scala.util.parsing.combinator.PackratParsers.PackratReader
9  //import scala.util.parsing.combinator.syntactical._
10 import scala.util.parsing.combinator.PackratParsers
11
12 object FumurtParser extends Parsers //with PackratParsers
13 {
14   override type Elem = Token
15   //type Tokens = Token
16   //type Token = Elem
17
18   def parse(in:List[Token]):Either[NoSuccess, List[Definition]]=
19   {
20     //val ast = parseAll((progParser), in)
21     val res = progParser(new TokenReader(in))
22     res match
23     {
24       case ns:NoSuccess=>
25       {
26         println(res+"\n")
27         //Left(new FumurtError(ns.next.pos, ns.msg, ""))
28         Left(ns)
29       }
30       case _=>
31       {
32         val ast = res.get
33         //println("\n")
34         //println(ast.toString+"\n")
35         Right(ast)
36       }
37     }
38   }
```

```

39
40
41 def progParser: Parser[List[Definition]] = (paddedDefParser.+ ) <~
eofParser
42 def paddedDefParser: Parser[Definition] =
  { /*println("paddeddefparser");*/ newlineParser.* ~> defParser
  <~ newlineParser.* }
43 def defParser: Parser[Definition] = { /*println("defparser");*/
  positioned((deflhsParser <~ equalParser ~! newlineParser.*) ~!
  defrhsParser ^^ {x=>Definition(x._1,x._2)}) }
44 def deflhsParser: Parser[DefLhs] = { /*println("deflhsparser");*/
  (defdescriptionParser ~ idParser ~ argsParser ~! (colonParser
  ~> typeParser)) ^^ {x=>DefLhs(x._1._1._1, x._1._1._2, x._1._2,
  x._2)} }
45 def argsParser: Parser[Option[Arguments]] =
  { /*println("argsparser");*/ openParenthesisParser ~> ((idParser
  <~ colonParser) ~ typeParser ~ subsequentArgsParser.*) <~
  closeParenthesisParser ^^ {x=>Some(Arguments( (Argument(x._1._1,
  x._1._2) +:
  x._2).sortWith((left,right)=>left.id.value<right.id.value) ))}
  | emptyParser ^^ {x=>None} }
46 def subsequentArgsParser: Parser[Argument] =
  { /*println("args2parserparser");*/ commaParser ~> (idParser <~
  colonParser) ~ typeParser ^^ {x=>Argument(x._1, x._2)} }
47
48 def defrhsParser: Parser[DefRhs] = { /*println("-defrhspartner");*/
  (openCurlyBracketParser ~ newlineParser.* ~> expressionParser ~
  (newlineParser.+ ~> expressionParser).*) <~ newlineParser.* ~
  closeCurlyBracketParser ^^ {x=>DefRhs(x._1 +: x._2)} }
49 def expressionParser: Parser[Expression] =
  { /*println("expressionparser");*/ positioned(defParser |
  statementParser) }
50
51 /*
52 def defrhsParser: Parser[DefRhs] = {println("-defrhspartner");
  (openCurlyBracketParser ~> expressionParser.+ ) <~
  newlineParser.* ~ closeCurlyBracketParser ^^ {x=>DefRhs(x)} }
53 def expressionParser: Parser[Expression] =
  {println("expressionparser"); newlineParser.+ ~>
  positioned(defParser | statementParser) }
54 */
55 def statementParser: Parser[Statement] =
  { /*println("statementparser");*/ functionCallParser |
  basicStatementParser | identifierStatementParser }
56 def callargsParser: Parser[Either[Callarg,NamedCallargs]] =
  { /*println("callargspartner");*/ openParenthesisParser ~>
  (namedcallargsParser | callargParser) <~ closeParenthesisParser
  ^^ {x=>x match{case x:Callarg => Left(x); case
  x:NamedCallargs=>Right(x)}} }
57 def callargParser: Parser[Callarg] = { /*println("callargspartner");*/
  positioned(functionCallParser | identifierStatementParser |
  basicStatementParser | success(NoArgs())) }
58 def namedcallargsParser: Parser[NamedCallargs] =
  { /*println("namedcallargspartner");*/ namedcallargParser ~
  subsequentnamedcallargsParser.+ ^^ {x => NamedCallargs((x._1 +:
  x._2).sortWith((left,right)=>left.id.value<right.id.value))} }
59 def subsequentnamedcallargsParser: Parser[NamedCallarg] =
  { /*println("subsequentnamedcallargspartner");*/ (commaParser ~!

```

```

59     success(Unit)) ~> namedcallargParser }
60 def namedcallargParser:Parser[NamedCallarg] =
    { /*println("namedcallargparser");*/ (idParser <~ equalParser) ~
      callargParser ^^ {x=>NamedCallarg(x._1, x._2)} }
61
62 def functionCallParser:Parser[FunctionCallStatement] =
63     { /*println("functioncallparser");*/ idParser ~ callargsParser
64     ^^ {x=>FunctionCallStatement(x._1 match{case IdT(str)=>str},
        x._2)} }
65
66 /*
67 def argsParser: Parser[Option[Arguments]] = {println("argsparser");
68     openParenthesisParser ~> ((idParser <~ colonParser) ~
69     typeParser ~ args2Parser) <~ closeParenthesisParser
70     ^^ {x=>Some(Arguments(x._1._1, x._1._2, x._2))} | emptyParser ^^
71     {x=>None} }
72
73 def args2Parser: Parser[Option[Arguments2]] =
74     {println("args2parserparser"); commaParser ~> (idParser <~
75     colonParser) ~ typeParser ~ args2Parser
76     ^^ {x=>Some(Arguments2(x._1._1, x._1._2, x._2))} | emptyParser
77     ^^ {None} }
78
79 */
80
81 def equalParser:Parser[Token] = accept(EqualT())
82 def colonParser:Parser[Elem] = accept(ColonT())
83 def commaParser:Parser[Elem] = accept(CommaT())
84 def newlineParser:Parser[Elem] = accept(NewlineT())
85 def emptyParser:Parser[Empty] = success(Empty())
86 def openParenthesisParser:Parser[Elem] = accept(OpenParenthesisT())
87 def closeParenthesisParser:Parser[Elem] =
    accept(CloseParenthesisT())
88
89 def openCurlyBracketParser:Parser[Elem] =
    accept(OpenCurlyBracketT())
90
91 def closeCurlyBracketParser:Parser[Elem] =
    accept(CloseCurlyBracketT())
92
93 def programStrParser:Parser[Elem] = accept(ProgramT())
94 def actionParser:Parser[Elem] = accept(ActionT())
95 def threadParser:Parser[Elem] = accept(ThreadT())
96 def functionParser:Parser[DefDescription] = accept("function",
    {case FunctionT() => DefDescription(FunctionT())})
97
98 def eofParser:Parser[Elem] = accept(EofT())
99 def idParser:Parser[IdT] = accept("identifier", {case IdT(value) =>
    { /*println("idparser accepted "+value);*/ IdT(value)}})
100
101 def trueParser:Parser[Elem] = accept(TrueT())
102 def falseParser:Parser[Elem] = accept(FalseT())
103 def identifierStatementParser:Parser[IdentifierStatement]
    = { /*println("identifierstatementparser");*/
    accept("identifier", {case
    IdT(str)=> { /*println("identifierstatementparser accepted
    "+str);*/ IdentifierStatement(str)}}) }
104
105 def basicStatementParser:Parser[BasicValueStatement] =
    accept("expected string, integer, boolean or float", {case
    StringT(value) => StringStatement(value);

```

```

88
89
90
91
92 def typeParser:Parser[TypeT] = accept("expected type. Types are
93   written with a leading capital letter", {case x:TypeT => x})
94 def intParser:Parser[Elem] = accept("integer", {case x:IntegerT =>
95   x})
96 def doubleParser:Parser[Elem] = accept("double", {case x:DoubleT =>
97   x})
98 def defdescriptionParser: Parser[DefDescriptionT] =
99   {/*println("defdescriptionParser");*/ accept("expected
100   function, action, thread or program", {case x:DefDescriptionT
101     => x}) }
102
103 class TokenReader(in:List[Token]) extends Reader[Elem]
104 {
105   def atEnd:Boolean = in.isEmpty
106   def first:Elem = in.head
107   def pos:Position = in.head.pos;
108   def rest = new TokenReader(in.tail)
109 }
110 }

```

C.6 Typechecker.scala

```

1 package fumurtCompiler
2 import scala.collection.mutable.ListBuffer
3
4 object FumurtTypeChecker
5 {
6   def check(in:List[Definition]):Option[List[FumurtError]] =
7     {
8       val print = DefLhs(ActionT(), IdT("actionPrint"),
9         Some(Arguments(List(Argument(IdT("toPrint"),
10           TypeT("String"))))), TypeT("Nothing"))
11       val basicfunctions = List(print)

```



```

12 //all standard library functions available everywhere (maybe also
13 //actions).
14 //checkexpression(in, DefLhs(UnsafeActionT(), IdT(""), None,
15 //TypeT("Nothing")), None, List(List():List[Definition]),
16 //basics, List():List[DefLhs], List():List[FumurtErrors])
17
18 //println()
19 val errors = checktop(in, basicfunctions)
20 //println()
21 if (errors.isEmpty)
22 {
23     None
24 }
25 else
26 {
27     Some(errors)
28 }
29 }
30
31 def checktop(in:List[Definition], basicFunctions:List[DefLhs]):
32 List[FumurtError]=
33 {
34     val topdefs = indexlefts(in)
35     val programs = in.filter(x=>(x.leftside.description match {case
36     ProgramT() => true; case _=> false}))
37     val implicitargs = topdefs.filter(x=>(x.description match {case
38     ProgramT() => false; case _=> true}))
39     //println("\nimPLICITargs is: "+implicitargs)
40     val programerrors = if(programs.length==1)
41     {
42         checkprogram(programs(0), implicitargs, basicFunctions)
43     }
44     else {List(FumurtError(Global, "There must be exactly one program
45     definition. "+programs.length+" program definitions
46     detected"))}
47     val program = programs(0)
48     //val synchronizedvars = program.rightside.expressions.filter(x=>
49     x match {case
50     Definition(DefLhs(SynchronizedVariableT(),_,_,_),_)=>true;
51     case _=>false}):List[Definition]
52     val synchronizedvars = program.rightside.expressions.flatMap(x=>
53     x match
54     {
55         case deff:Definition=>if(deff.leftside.description ==
56         SynchronizedVariableT()) {Some(deff.leftside)} else
57         {None};
58         case _=>None
59     }
60     ):List[DefLhs]
61     val nonProgramDefs = in.filter(x=>(x.leftside.description match
62     {case ProgramT() => false; case _=> true}))
63     val othererrors = checkexpressions(nonProgramDefs, None,
64     Some(implicitargs++synchronizedvars), basicFunctions)
65
66     programerrors++othererrors
67 }

```

```

53 def checkprogram(program:Definition, topleveldefs:List[DefLhs],
54   basicFunctions:List[DefLhs]): List[FumurtError]=
55 {
56   def checkuseofthread(program:Definition,
57     thread:DefLhs):List[FumurtError]=
58   {
59     thread.description match
60     {
61       case ThreadT() => program.rightside.expressions.find(y=>y
62         match{case FunctionCallStatement(thread.id.value, _) =>
63           true; case _=>false})
64         match
65         {
66           case Some(_)=> List();
67           case None=> List(FumurtError(Global, "thread
68             "+thread.id.value+" is declared but not used"))
69         }
70       case _=> List()
71     }
72   }
73   val unusedthreaderrors:List[FumurtError] =
74     topleveldefs.foldLeft(List():
75       List[FumurtError])((x:List[FumurtError], y:DefLhs)=>
76         x++checkuseofthread(program,y)
77     ):List[FumurtError]
78
79   val lefts = indexlefts(program.rightside.expressions)
80   val unsuitableexpressions =
81     program.rightside.expressions.foldLeft(List():
82       List[FumurtError])( (x,y)=>
83     y match
84     {
85       case z:Definition=>
86       {
87         z.leftside.description match
88         {
89           case SynchronizedVariableT() =>
90           {
91             if(z.rightside.expressions.length !=
92               1){x++List(FumurtError(z.pos, "only single call to
93                 synchronized permitted"))}
94             else
95             {
96               val synchcall = z.rightside.expressions(0)
97               val signatureerror =
98                 synchcall match
99                 {
100                   case FunctionCallStatement( "synchronized",
101                     Right(NamedCallargs(List(
102                       NamedCallarg(IdT("variable"),
103                         variablearg:Callarg),
104                       NamedCallarg(IdT("writer"),writerarg:
105                         Callarg)))))) =>
106                   {
107                     x++checkCallarg(z.leftside.returntype,
108                       variablearg, IdT("variable"),
109                       program.leftside, None, basicFunctions,

```

```

92         List()) //TODO: make sure that writer is a
93         }
94         case _=>x++List(FumurtError(synchcall.pos, "must be
95         call to synchronized with \"variable\" and
96         \"writer\" arguments"))
97     }
98     x++signatureerror
99 }
100 }
101 }
102 case z:FunctionCallStatement=>
103 {
104     if(!z.functionidentifier.startsWith("thread")) {x ++
105         List(FumurtError(z.pos, "Only threads can be called in
106         Program"))}
107     else
108     {
109         x++checkstatement(z, program.leftside, None,
110         basicFunctions, lefts++topleveldefs, TypeT("Nothing"))
111     }
112     case z:Expression=>x++List(FumurtError(z.pos, "Only
113     definitions and thread start statements allowed in
114     Program"))
115 }
116 )
117 //println(program.rightside.expressions)
118 //println("unsuit "+(unusedthreaderrors ++
119     unsuitabledefinitions.toList))
120 (unusedthreaderrors ++
121     unsuitableexpressions.toList):List[FumurtError]
122 }
123 def checkexpressions(tree>List[Expression],
124     containingdefinition:Option[Definition],
125     containingdefinitionarguments:Option[List[DefLhs]],
126     basicFunctions>List[DefLhs]):List[FumurtError]=
127 {
128     val insamedefinition = indexlefts(tree)
129     //println("\nin checkexpressions:  insamedefinition is
130     "+insamedefinition+" containingdefinition is
131     "+containingdefinition)
132     tree.foldLeft(List():List[FumurtError])((x,y) =>x
133     ++checkexpression(y, containingdefinition,
134     containingdefinitionarguments, basicFunctions,
135     insamedefinition))
136 }

```

```

127 def checkexpression(tocheck:Expression,
128 containingdefinition:Option[Definition],
129 arguments:Option[List[DefLhs]], basicFunctions:List[DefLhs],
130 inSameDefinition:List[DefLhs]):List[FumurtError] =
131 {
132 //println("\nIn checkexpression: tocheck:
133 "+tocheck+"containingdefinition: "+containingdefinition+"
134 arguments: "+arguments)
135 tocheck match
136 {
137 case x:Definition=>
138 {
139 val (newargs, argpropagationerrors) = x.leftside.args match
140 {
141 case None => (List(), List())
142 case Some(Arguments(args)) =>
143 {
144 val hits = arguments match
145 {
146 case Some(contargs) => args.flatMap(arg =>
147 (contargs++inSameDefinition).find(y =>
148 y.id.value==arg.id.value))
149 case None => args.flatMap(arg =>
150 inSameDefinition.find(y =>
151 y.id.value==arg.id.value))
152 }
153 if (hits.length == args.length) //used to be !=.
154 Don't know why. bug?
155 {
156 (hits, List())
157 }
158 else
159 {
160 // (hits, List(FumurtError(x.pos, "One or more arguments
161 not found in local scope"))) TODO: Find better
162 solution than just abandoning compile time
163 dependent checking. Checking for each function call
164 might be possible...
165 (hits, List())
166 }
167 }
168 }
169 }
170 checkdefinition(x, containingdefinition.map(x=>x.leftside),
171 Some(newargs), basicFunctions) ++ argpropagationerrors
172 }
173 case x:Statement => containingdefinition match
174 {
175 case None => List(FumurtError(x.pos, "Statements must be
176 enclosed in either Program or another definition"))
177 case Some(contdef) => /*println("\n"+x);*/ checkstatement(x,
178 contdef.leftside, arguments, basicFunctions,
179 inSameDefinition, contdef.leftside.returntype)
180 }
181 }
182 }
183 }
184 }
185 }

```

```

166 def checkstatement(tocheck:Statement, containingdefinition:DefLhs,
167 arguments:Option[List[DefLhs]], basicFunctions>List[DefLhs],
168 inSameDefinition>List[DefLhs], expectedreturn:TypeT):
List[FumurtError]=
169 {
170 //println("\nIn checkstatement: tocheck:
171 "+tocheck+"containingdefinition: "+containingdefinition+"
172 arguments: "+arguments)
173 tocheck match
174 {
175 case b:BasicValueStatement=>
176 checkbasicvaluestatement(expectedreturn, b, "Return")
177 case b:IdentifierStatement=>
178 {
179 val statedvalue = findinscope(arguments, inSameDefinition,
180 basicFunctions, Some(containingdefinition), b.value)
181 statedvalue match
182 {
183 case Left(string) => List(FumurtError(b.pos, /*"in
184 checkstatement "+*/string))
185 case Right(deflhs) =>
186 {
187 if(containingdefinition.returntype.value !=
188 deflhs.returntype.value)
189 {
190 List(FumurtError(b.pos, "expected: "
191 +expectedreturn.value+ ". Got: "
192 +deflhs.returntype.value))
193 }
194 else
195 {
196 List()
197 }
198 }
199 }
200 }
201 case y:FunctionCallStatement=>
202 {
203 //println("found "+y)
204 if (y.functionidentifier=="if")
205 {
206 checkifcall(y, expectedreturn, containingdefinition,
207 arguments, basicFunctions, inSameDefinition)
208 }
209 else if (y.functionidentifier=="plus" ||
210 y.functionidentifier=="minus" ||
211 y.functionidentifier=="multiply" ||
212 y.functionidentifier=="divide")
213 {
214 checkbasicmathcall(y, expectedreturn, containingdefinition,
215 arguments, basicFunctions, inSameDefinition)
216 }
217 else if (y.functionidentifier=="toString")
218 {
219 checktostringcall(y, expectedreturn, containingdefinition,
220 arguments, basicFunctions, inSameDefinition)
221 }
222 }

```

```

206     else if (y.functionidentifier=="actionMutate")
207     {
208         checkmutatecall(y, expectedreturn, containingdefinition,
209             arguments, basicFunctions, inSameDefinition)
210     }
211     else if (y.functionidentifier=="equal")
212     {
213         val reterror = if(expectedreturn!=TypeT("Boolean"))
214             {List(FumurtError(tocheck.pos, "Call to equal always
215                 returns boolean, not
216                 "+expectedreturn.value))}else{List()}
217         val argerrors = y.args match
218         {
219             case Right(NamedCallargs(List(NamedCallarg(IdT("left"),
220                 leftargument), NamedCallarg(IdT("right"),
221                 rightargument)))) =>
222             {
223                 List()
224             }
225             case _=>List(FumurtError(tocheck.pos, "Call to equal
226                 requires two arguments named left and right"))
227         }
228         reterror++argerrors
229     }
230     /*else if (y.functionidentifier=="lessThan" || "biggerthan")
231     {
232         val reterror = if(expectedreturn!=TypeT("Boolean"))
233             {List(FumurtError(ifcall.pos, "Call to
234                 "+y.functionidentifier+" always returns boolean, not
235                 "+expectedreturn.value))} else{List()}
236     }
237     else if (y.functionidentifier=="not")
238     {
239         val reterror = if(expectedreturn!=TypeT("Boolean"))
240             {List(FumurtError(ifcall.pos, "Call to not always
241                 returns boolean, not "+expectedreturn.value))}
242             else{List()}
243     }*/
244     else
245     {
246         findinscope(arguments, inSameDefinition, basicFunctions,
247             Some(containingdefinition), y.functionidentifier) match
248         {
249             case Left(string) => List(FumurtError(y.pos, /*"in
250                 checkstatement_2 "+*/string))
251             case Right(calledfunction) =>
252             {
253                 val argumenterrors:List[FumurtError] = y.args match
254                 {
255                     case Left(NoArgs()) => calledfunction.args match
256                     {
257                         case None => List()
258                         case Some(_) => List(FumurtError(y.pos, "expected
259                             arguments, but none were given"))
260                     }
261                 }

```

```

246         case Left(callarg) => calledfunction.args match
           //checkCallarg(, callarg, containingdefinition,
           arguments, basicFunctions, inSameDefinition)
247     {
248         case Some(Arguments(args)) =>
249     {
250         if (args.length != 1) { List(FumurtError(y.pos,
           "expected "+args.length+" arguments, but only
           one was given")) }
251         else { checkCallarg(args(0).typestr, callarg,
           args(0).id, containingdefinition, arguments,
           basicFunctions, inSameDefinition) }
252     }
253     case None => List(FumurtError(y.pos, "expected no
           arguments, but some were given"))
254     }
255     case Right(NamedCallargs(value)) =>
256     {
257         //println("checking namedcallargs "+value)
258         checknamedcallargs(calledfunction, value,
           containingdefinition, arguments,
           basicFunctions, inSameDefinition)
259     }
260     }
261     val returnerror:List[FumurtError] = if (expectedreturn
           != calledfunction.returntype)
262     {
263         List(FumurtError(y.pos, "Expected return type
           "+expectedreturn.value+". Got
           "+calledfunction.returntype.value/*+".
           containingdefinition is"+containingdefinition*/)
264     }
265     else {List()}
266     returnerror ++ argumenterrors
267     }
268     }
269     }
270     }
271     }
272     }
273     }
274     }
275     def checkifcall(ifcall:FunctionCallStatement, expectedtype:TypeT,
           containingdefinition:DefLhs, arguments:Option[List[DefLhs]],
           basicFunctions:List[DefLhs],
           inSameDefinition:List[DefLhs]):List[FumurtError] =
276     {
277         ifcall.args match
278     {
279         case Left(callarg) => List(FumurtError(ifcall.pos, "Call to if
           needs three arguments"))
280         case Right(NamedCallargs(arglist))=>
281     {
282         if (arglist.length != 3)
283     {
284         List(FumurtError(ifcall.pos, "Call to if needs three
           arguments"))

```

```

285     }
286     else
287     {
288         ( if(arglist(0).id.value !=
                "condition"){List(FumurtError(ifcall.pos, "Call to if
                needs a condition argument"))} else {List()} )++
289         ( if(arglist(1).id.value !=
                "else"){List(FumurtError(ifcall.pos, "Call to if needs
                an else argument"))} else {List()} )++
290         ( if(arglist(2).id.value !=
                "then"){List(FumurtError(ifcall.pos, "Call to if needs
                a then argument"))} else {List()} )++
291         checkCallarg(TypeT("Boolean"), arglist(0).argument,
                IdT("condition"), containingdefinition, arguments,
                basicFunctions, inSameDefinition)++
292         (checkCallarg(expectedtype, arglist(1).argument,
                IdT("else"), containingdefinition, arguments,
                basicFunctions, inSameDefinition)++
293         (checkCallarg(expectedtype, arglist(2).argument,
                IdT("then"), containingdefinition, arguments,
                basicFunctions, inSameDefinition))
294     }
295 }
296 }
297 }
298
299 def checkmutatecall(call:FunctionCallStatement, expectedtype:TypeT,
    containingdefinition:DefLhs, arguments:Option[List[DefLhs]],
    basicFunctions>List[DefLhs],
    inSameDefinition>List[DefLhs]):List[FumurtError] =
300 {
301     //println("mutate call "+call)
302     call.args match
303     {
304         case Left=>List(FumurtError(call.pos, "call to mutate requires
                both a variable, and a new value to assign to that
                variable"))
305         case Right(NamedCallargs(List(value:NamedCallarg,
                variable:NamedCallarg)))=>
306         {
307             val firstnameerror = if(value.id.value !=
                "newValue"){List(FumurtError(call.pos, "call to mutate
                requires argument \"newValue\")} else{List()}
308             val lastnameerror = if(variable.id.value !=
                "variable"){List(FumurtError(call.pos, "call to mutate
                requires argument \"variable\")} else{List()}
309             val variabletypeerror = variable.argument match
310             {
311                 case z:IdentifierStatement=>
312                 {
313                     findinscope(arguments, inSameDefinition, basicFunctions,
                Some(containingdefinition), z.value) match
314                     {
315                         case Left(str) => List(FumurtError(z.pos, str))
316                         case Right(defl)=>
317                     {

```



```

318         (if (defl.description != SynchronizedVariableT())
319             {List(FumurtError(call.pos, "Variable must be
synchronized"))}else{List()})++
320         (checkCallarg(defl.returntype, value.argument,
321             IdT("variable"), containingdefinition, arguments,
322             basicFunctions, inSameDefinition))
323     }
324 }
325 }
326 case z:Expression=>List(FumurtError(call.pos, "variable
327 argument must be an identifier"))
328 }
329 }
330 }
331 }
332 }
333 }
334 }
335 }
336 }
337 }
338 }
339 }
340 }
341 }
342 }
343 }
344 }
345 }
346 }
347 }
348 }
349 }
350 }
351 }
352 }
353 }
354 }
355 }
356 }
357 }
358 }
359 }
360 }
361 }
362 }
363 }
364 }
365 }
366 }
367 }
368 }
369 }
370 }
371 }
372 }
373 }
374 }
375 }
376 }
377 }
378 }
379 }
380 }
381 }
382 }
383 }
384 }
385 }
386 }
387 }
388 }
389 }
390 }
391 }
392 }
393 }
394 }
395 }
396 }
397 }
398 }
399 }
400 }
401 }
402 }
403 }
404 }
405 }
406 }
407 }
408 }
409 }
410 }
411 }
412 }
413 }
414 }
415 }
416 }
417 }
418 }
419 }
420 }
421 }
422 }
423 }
424 }
425 }
426 }
427 }
428 }
429 }
430 }
431 }
432 }
433 }
434 }
435 }
436 }
437 }
438 }
439 }
440 }
441 }
442 }
443 }
444 }
445 }
446 }
447 }
448 }
449 }
450 }
451 }
452 }
453 }
454 }
455 }
456 }
457 }
458 }
459 }
460 }
461 }
462 }
463 }
464 }
465 }
466 }
467 }
468 }
469 }
470 }
471 }
472 }
473 }
474 }
475 }
476 }
477 }
478 }
479 }
480 }
481 }
482 }
483 }
484 }
485 }
486 }
487 }
488 }
489 }
490 }
491 }
492 }
493 }
494 }
495 }
496 }
497 }
498 }
499 }
500 }
501 }
502 }
503 }
504 }
505 }
506 }
507 }
508 }
509 }
510 }
511 }
512 }
513 }
514 }
515 }
516 }
517 }
518 }
519 }
520 }
521 }
522 }
523 }
524 }
525 }
526 }
527 }
528 }
529 }
530 }
531 }
532 }
533 }
534 }
535 }
536 }
537 }
538 }
539 }
540 }
541 }
542 }
543 }
544 }
545 }
546 }
547 }
548 }
549 }
550 }
551 }
552 }
553 }
554 }
555 }
556 }
557 }
558 }
559 }
560 }
561 }
562 }
563 }
564 }
565 }
566 }
567 }
568 }
569 }
570 }
571 }
572 }
573 }
574 }
575 }
576 }
577 }
578 }
579 }
580 }
581 }
582 }
583 }
584 }
585 }
586 }
587 }
588 }
589 }
590 }
591 }
592 }
593 }
594 }
595 }
596 }
597 }
598 }
599 }
600 }
601 }
602 }
603 }
604 }
605 }
606 }
607 }
608 }
609 }
610 }
611 }
612 }
613 }
614 }
615 }
616 }
617 }
618 }
619 }
620 }
621 }
622 }
623 }
624 }
625 }
626 }
627 }
628 }
629 }
630 }
631 }
632 }
633 }
634 }
635 }
636 }
637 }
638 }
639 }
640 }
641 }
642 }
643 }
644 }
645 }
646 }
647 }
648 }
649 }
650 }
651 }
652 }
653 }
654 }
655 }
656 }
657 }
658 }
659 }
660 }
661 }
662 }
663 }
664 }
665 }
666 }
667 }
668 }
669 }
670 }
671 }
672 }
673 }
674 }
675 }
676 }
677 }
678 }
679 }
680 }
681 }
682 }
683 }
684 }
685 }
686 }
687 }
688 }
689 }
690 }
691 }
692 }
693 }
694 }
695 }
696 }
697 }
698 }
699 }
700 }
701 }
702 }
703 }
704 }
705 }
706 }
707 }
708 }
709 }
710 }
711 }
712 }
713 }
714 }
715 }
716 }
717 }
718 }
719 }
720 }
721 }
722 }
723 }
724 }
725 }
726 }
727 }
728 }
729 }
730 }
731 }
732 }
733 }
734 }
735 }
736 }
737 }
738 }
739 }
740 }
741 }
742 }
743 }
744 }
745 }
746 }
747 }
748 }
749 }
750 }
751 }
752 }
753 }
754 }
755 }
756 }
757 }
758 }
759 }
760 }
761 }
762 }
763 }
764 }
765 }
766 }
767 }
768 }
769 }
770 }
771 }
772 }
773 }
774 }
775 }
776 }
777 }
778 }
779 }
780 }
781 }
782 }
783 }
784 }
785 }
786 }
787 }
788 }
789 }
790 }
791 }
792 }
793 }
794 }
795 }
796 }
797 }
798 }
799 }
800 }
801 }
802 }
803 }
804 }
805 }
806 }
807 }
808 }
809 }
810 }
811 }
812 }
813 }
814 }
815 }
816 }
817 }
818 }
819 }
820 }
821 }
822 }
823 }
824 }
825 }
826 }
827 }
828 }
829 }
830 }
831 }
832 }
833 }
834 }
835 }
836 }
837 }
838 }
839 }
840 }
841 }
842 }
843 }
844 }
845 }
846 }
847 }
848 }
849 }
850 }
851 }
852 }
853 }
854 }
855 }
856 }
857 }
858 }
859 }
860 }
861 }
862 }
863 }
864 }
865 }
866 }
867 }
868 }
869 }
870 }
871 }
872 }
873 }
874 }
875 }
876 }
877 }
878 }
879 }
880 }
881 }
882 }
883 }
884 }
885 }
886 }
887 }
888 }
889 }
890 }
891 }
892 }
893 }
894 }
895 }
896 }
897 }
898 }
899 }
900 }
901 }
902 }
903 }
904 }
905 }
906 }
907 }
908 }
909 }
910 }
911 }
912 }
913 }
914 }
915 }
916 }
917 }
918 }
919 }
920 }
921 }
922 }
923 }
924 }
925 }
926 }
927 }
928 }
929 }
930 }
931 }
932 }
933 }
934 }
935 }
936 }
937 }
938 }
939 }
940 }
941 }
942 }
943 }
944 }
945 }
946 }
947 }
948 }
949 }
950 }
951 }
952 }
953 }
954 }
955 }
956 }
957 }
958 }
959 }
960 }
961 }
962 }
963 }
964 }
965 }
966 }
967 }
968 }
969 }
970 }
971 }
972 }
973 }
974 }
975 }
976 }
977 }
978 }
979 }
980 }
981 }
982 }
983 }
984 }
985 }
986 }
987 }
988 }
989 }
990 }
991 }
992 }
993 }
994 }
995 }
996 }
997 }
998 }
999 }
1000 }

```

```

351     val returnsdouble = leftrightdouble || rightrightdouble
352     ( if(arglist(0).id.value !=
      "left"){List(FumurtError(call.pos, "Call to
      "+call.functionidentifier+" needs a left argument"))}
      else {List()} )++)
353     ( if(arglist(1).id.value !=
      "right"){List(FumurtError(call.pos, "Call to
      "+call.functionidentifier+" needs a right argument"))}
      else {List()} )++)
354     ( lefterrors )++
355     ( righterrors )++
356     ( expectedtype match
357     {
358     case TypeT("Double")=>List();
359     case TypeT("Integer") =>
      if(returnsdouble){List(FumurtError(call.pos, "This
      call to "+call.functionidentifier+" returns a
      Double not an Integer"))} else{List()}
360     case TypeT(str)=>
361     {
362     if(returnsdouble){List(FumurtError(call.pos, "This
      call to "+call.functionidentifier+" returns a
      Double not "+str))}
363     else{List(FumurtError(call.pos, "This call to
      "+call.functionidentifier+" returns an Integer
      not "+str))}
364     }
365     }
366     )
367     }
368     }
369     }
370     }
371
372     def checktostringcall(call:FunctionCallStatement,
      expectedtype:TypeT, containingdefinition:DefLhs,
      arguments:Option[List[DefLhs]], basicFunctions>List[DefLhs],
      inSameDefinition>List[DefLhs]):List[FumurtError] =
373     {
374     call.args match
375     {
376     case Left(callarg) =>
377     {
378     val integererrors = checkCallarg(TypeT("Integer"), callarg,
      IdT("none needed as not user defined and single
      argument"), containingdefinition, arguments,
      basicFunctions, inSameDefinition)
379     val doubleerrors = checkCallarg(TypeT("Double"), callarg,
      IdT("none needed as not user defined and single
      argument"), containingdefinition, arguments,
      basicFunctions, inSameDefinition)
380     val argumenterrors = if(integererrors.length <
      doubleerrors.length){integererrors} else{doubleerrors}
381     val outerrors = expectedtype match{ case
      TypeT("String")=>List(); case
      TypeT(str)=>List(FumurtError(call.pos, "toString returns
      String not "+str))}

```

```

382     argumenterrors++outerrors
383   }
384   case Right(NamedCallargs(arglist))=>List(FumurtError(call.pos,
385     "Call to toString needs one argument"))
386 }
387 }
388 def checknamedcallargs(calledfunction:DefLhs,
389   namedcallargs:List[NamedCallarg], containingdefinition:DefLhs,
390   arguments:Option[List[DefLhs]], basicFunctions:List[DefLhs],
391   inSameDefinition:List[DefLhs]):List[FumurtError] =
392 {
393   calledfunction.args match
394   {
395     case None => List(FumurtError(namedcallargs(0).id.pos, "No
396       arguments expected, but "+namedcallargs.length+" were
397       given"))
398     case Some(Arguments(defargs)) =>
399     {
400       if (defargs.length != namedcallargs.length)
401       {
402         List(FumurtError(namedcallargs(0).id.pos, "expected
403           "+defargs.length+" arguments. Got
404           "+namedcallargs.length+" arguments"))
405       }
406       else
407       {
408         if( !namedcallargs.groupBy(x => x.id.value).filter(y =>
409           y._2.length>1).isEmpty ) //ensure uniqueness of
410           arguments
411         {
412           List(FumurtError(namedcallargs(0).id.pos, "two or more
413             arguments were given with the same name"))
414         }
415         else
416         {
417           val individualargumenterrors =
418             ListBuffer():ListBuffer[FumurtError]
419           for(i<-0 until namedcallargs.length)
420           {
421             individualargumenterrors +=
422               (if(namedcallargs(i).id.value !=
423                 defargs(i).id.value)
424               {
425                 //println("FOUND INCORRECT NAMES")
426                 List(FumurtError(namedcallargs(i).id.pos, "Wrong
427                   argument name. Argument in definition named
428                   "+defargs(i).id.value+". In calling named
429                   "+namedcallargs(i).id.value ))
430               }
431             else
432             {
433               checkCallarg(defargs(i).typestr,
434                 namedcallargs(i).argument, defargs(i).id,
435                 containingdefinition,
436                 arguments:Option[List[DefLhs]],
437                 basicFunctions:List[DefLhs],

```

```

418         inSameDefinition:List[DefLhs])
419     }
420 }
421
422     //println("individualargumenterrors.toList:
423     "+individualargumenterrors.toList)
424     individualargumenterrors.toList
425 }
426 }
427 }
428 }
429
430 def checkCallarg(expectedtype:TypeT, arg:Callarg, id:IdT,
431     containingdefinition:DefLhs, arguments:Option[List[DefLhs]],
432     basicFunctions:List[DefLhs],
433     inSameDefinition:List[DefLhs]):List[FumurtError] =
434 {
435     //println("in checkCallarg. arg is "+arg)
436     arg match
437     {
438         case c:BasicValueStatement=>
439             checkbasicvaluestatement(expectedtype, c, "Call argument")
440         case c:NoArgs =>
441         {
442             println("NoArgs got checked by checkCallarg. This is better
443             checked in checkstatement"); scala.sys.exit()
444         }
445         case c:IdentifierStatement =>
446         {
447             findinscope(arguments, inSameDefinition, basicFunctions,
448                 Some(containingdefinition), c.value) match
449             {
450                 case Left(str) => List(FumurtError(c.pos, /*"in
451                 checkcallarg "+*/str))
452                 case Right(thingdef) =>
453                 {
454                     if(expectedtype.value == "Inclusion")
455                     {
456                         if(thingdef.id.value != id.value)
457                         {
458                             List(FumurtError(c.pos, "Passed inclusion must be the
459                             same as the one referenced inside the function"))
460                         }
461                     }
462                     else{List()}
463                 }
464                 else if(expectedtype.value != thingdef.returntype.value)
465                 {
466                     List(FumurtError(c.pos, "Expected type
467                     "+expectedtype.value+". Got
468                     "+thingdef.returntype.value))
469                 }
470                 else {List()}
471             }
472         }
473     }
474 }

```

```

463     }
464     case c:FunctionCallStatement =>
465     {
466         //check that call end result is correct
467
468         //check that call itself is correct
469         val callerrors = checkstatement(c, containingdefinition,
470             arguments, basicFunctions, inSameDefinition, expectedtype)
471         callerrors //++ resulterrors
472     }
473 }
474
475 def checkbasicvaluestatement(expectedtype:TypeT,
476     basicstatement:BasicValueStatement,
477     role:String):List[FumurtError] =
478 {
479     basicstatement match
480     {
481         case c:StringStatement => {if (expectedtype.value != "String")
482             List(FumurtError(c.pos, role+" type should be
483                 "+expectedtype.value+". "+role+" type was String")) else
484             List()}
485         case c:IntegerStatement => {if (expectedtype.value !=
486             "Integer") List(FumurtError(c.pos, role+" type should be
487                 "+expectedtype.value+". "+role+" type was Integer")) else
488             List()}
489         case c:DoubleStatement => {if (expectedtype.value != "Double")
490             List(FumurtError(c.pos, role+" type should be
491                 "+expectedtype.value+". "+role+" type was Double")) else
492             List()}
493         case c:TrueStatement => {if (expectedtype.value != "Boolean")
494             List(FumurtError(c.pos, role+" type should be
495                 "+expectedtype.value+". "+role+" type was Boolean")) else
496             List()}
497         case c:FalseStatement => {if (expectedtype.value != "Boolean")
498             List(FumurtError(c.pos, role+" type should be
499                 "+expectedtype.value+". "+role+" type was Boolean")) else
500             List()}
501     }
502 }
503
504 def checkdefinition(tocheck:Definition,
505     containingdefinition:Option[DefLhs],
506     arguments:Option[List[DefLhs]], basicFunctions:List[DefLhs]):
507     List[FumurtError]=
508 {
509     //println("\nIn checkdefinition:  tocheck:
510         "+tocheck+"containingdefinition: "+containingdefinition+"
511         arguments: "+arguments)
512     val undererrors = checkexpressions(tocheck.rightside.expressions,
513         Some(tocheck), arguments, basicFunctions)
514     val threaderror:List[FumurtError] =
515         tocheck.leftside.description match
516     {
517         case ThreadT() => tocheck.rightside.expressions.last match
518         {

```

```

495     case FunctionCallStatement(functionidentifier,_) =>
496     {
497         if(functionidentifier != tocheck.leftside.id.value)
498         {
499             List(FumurtError(tocheck.rightside.expressions.last.pos,
500                 "A thread must recurse on itself (at least until
501                 exit() is implemented)")
502         }
503         else
504         {
505             List()
506         }
507     }
508     case _ =>
509     List(FumurtError(tocheck.rightside.expressions.last.pos,
510         "A thread must recurse on itself (at least until exit()
511         is implemented)")
512     )
513     case _ => List()
514 }
515 val nameerror = tocheck.leftside.description match
516 {
517     case ActionT() =>
518     if(!tocheck.leftside.id.value.startsWith("action"))
519     {List(FumurtError(tocheck.pos, "Name of action is not
520     prefixed with \"action\")")} else{List()}
521     case ThreadT() =>
522     if(!tocheck.leftside.id.value.startsWith("thread"))
523     {List(FumurtError(tocheck.pos, "Name of thread is not
524     prefixed with \"thread\")")} else{List()}
525     case FunctionT() => List()
526     case ValueT() => List()
527     case ProgramT() => println("Program got checked by
528     checkdefinition. This is better checked in checkprogram");
529     scala.sys.exit()
530 }
531 val permissionerror = tocheck.leftside.description match
532 {
533     case ActionT() => containingdefinition match
534     {
535         case None=>List()
536         case Some(DefLhs(ValueT(),_,_,_))=>
537         List(FumurtError(tocheck.pos, "actions cannot be defined
538         in values"))
539         case Some(DefLhs(FunctionT(),_,_,_))=>
540         List(FumurtError(tocheck.pos, "actions cannot be defined
541         in functions"))
542         case Some(something) => List()
543     }
544     case ThreadT() => containingdefinition match{ case None =>
545     List(); case Some(_)=>List(FumurtError(tocheck.pos,
546     "threads must be defined on top "+containingdefinition))}
547     case FunctionT() => containingdefinition match{ case
548     Some(DefLhs(ValueT(),_,_,_)) =>
549     List(FumurtError(tocheck.pos, "functions cannot be defined
550     in values")); case _=> List()}

```

```

529     case SynchronizedVariableT() => List(FumurtError(tocheck.pos,
530         "synchronized variables must be defined in Program
531         definition"))
532     case ValueT() => List()
533     case ProgramT() => println("Program got checked by
534         checkdefinition. This is better checked in checkprogram");
535         scala.sys.exit()
536 }
537
538 undererrors.toList ++ nameerror ++ permissionerror ++
539 threadenderror
540 }
541
542 def indexlefts(in:List[Expression]):List[DefLhs]=
543 {
544     in.foldLeft(List():List[DefLhs]) ((list,y)=> y match
545     {
546         case Definition(leftside, _)=>list :+ leftside;
547         case _:Statement=> list
548     }
549     )
550 }
551
552 def findinscope(arguments:Option[List[DefLhs]],
553     inSameDefinition:List[DefLhs], basicfunctions:List[DefLhs],
554     enclosingDefinition:Option[DefLhs],
555     searchFor:String):Either[String, DefLhs]=
556 {
557     val argsres = arguments match{ case
558         Some(args)=>args.filter(y=>y.id.value==searchFor); case
559         None=>List():List[DefLhs]}
560     val inscoperes = inSameDefinition.filter(x=>x.id.value==searchFor)
561     //println()
562     //println(basicfunctions)
563     //println()
564     val basicfunctionres =
565         basicfunctions.filter(x=>x.id.value==searchFor)
566
567     val enclosingres = enclosingDefinition match
568     {
569         case None => List()
570         case Some(deff) => if (deff.id.value == searchFor) {List(deff)}
571             else {List()}
572     }
573
574     val res = argsres ++ inscoperes ++ basicfunctionres ++
575         enclosingres
576
577     if(res.length == 1)
578     {
579         Right(res.head)
580     }
581     else if(res.length>1)
582     {
583         Left("Ambiguous reference to "+searchFor)
584     }
585 }

```

```

573     else if(res.length == 0)
574     {
575         enclosingDefinition match
576         {
577             case None=>Left(searchFor+" not found" /*+" arguments is:
578                 "+arguments+". insamedefinition is "+inSameDefinition*/)
579             case Some(DefLhs(_,_,Some(Arguments(internalargs)),_))=>
580             {
581                 internalargs.find(x=>x.id.value==searchFor) match
582                 {
583                     case Some(Argument(id, TypeT("Inclusion")))=>
584                         Left(searchFor+" not found" /*+" arguments is:
585                             "+arguments+". insamedefinition is
586                             "+inSameDefinition*/)
587                     case Some(Argument(id, typestr))=>
588                         Right(DefLhs(ValueT(),id,None,typestr))
589                     case None=>Left(searchFor+" not found" /*+" arguments is:
590                         "+arguments+". insamedefinition is
591                         "+inSameDefinition*/)
592                 }
593             }
594             case Some(_)=> Left(searchFor+" not found" /*+" arguments is:
595                 "+arguments+". insamedefinition is "+inSameDefinition*/)
596         }
597     }
598     else
599     {
600         Left("error in search for "+searchFor)
601     }
602 }
603
604 //TODO: add type for synchronized variables and use it to pass them
605         around, so that it can be controlled that a thread calling
606         actionMutate has write rights

```

C.7 CodeGenerator.scala

```

1 package fumurtCompiler
2
3 import scala.collection.mutable.ListBuffer
4
5 object FumurtCodeGenerator
6 {
7     def generate(ast:List[Definition]):String =
8     {
9         val includestatement = "#include <iostream>\n#include
10             <thread>\n#include <string>\n#include <atomic>\n#include
11             <condition_variable>\n#include <list>\n#include
12             <chrono>\n\n\n"

```



```

10     val topthreads = gettopthreadstatements(ast)
11     val atree = getAnnotatedTree(ast, topthreads)
12     //println(atree)
13     val numtopthreads = topthreads.length
14     val synchronizationGlobalVars = "static std::atomic<int>
        rendezvousCounter;\nstatic std::mutex
        rendezvousSyncMutex;\nstatic std::condition_variable cv;"
15     val main = getmain(topthreads, atree)
16     val synchvars = getsynchronizedvariables(ast)
17     val syncfunc = getsynchronizerfunction(synchvars, topthreads)
18     val synchvardeclarations =
        getGlobalSynchVariableDeclarations(synchvars)
19     val printdecs = getprintlistdeclarations(topthreads)
20     //val topthreaddeclarations = gettopthreaddeclarations(ast)
21     val (funSignatures, funDeclarations) =
        getFunctionDeclarations(atree)
22     val staticthreadargs =
        getStaticThreadArgs(atree>List[aExpression])
23     val topThreadNumMacroIn = "#define NUMTOPTHREADS " +
        numtopthreads.toString + "\n"
24
25     //println(funSignatures)
26
27     includestatement + topThreadNumMacroIn + funSignatures + "\n" +
        synchvardeclarations + printdecs + "\n" +
        synchronizationGlobalVars + staticthreadargs + syncfunc +
        "\n\n" /*+ topthreaddeclarations*/ + "\n"+ funDeclarations +
        "\n\n" + main
28 }
29
30
31 def getAnnotatedTree(ast>List[Expression],
    topthreadcalls>List[FunctionCallStatement]):List[aExpression] =
32 {
33     val treeWithAnnotatedDefinitions =
        getAnnotatedTreeInternal(ast,topthreadcalls,"", None)
34     getCallsAnnotatedTreeInternal(treeWithAnnotatedDefinitions,
        List(), None)
35 }
36
37 def getCallsAnnotatedTreeInternal(ast>List[aExpression],
    arguments>List[aDefLhs],
    containingDefinition:Option[aDefinition]):List[aExpression] =
38 {
39     val inSameDefinition = indexlefts(ast)
40
41     ast.flatMap(node=>node match
42     {
43         case deff @ aDefinition(aDefLhs(desc, id, cppid,
            callingthread, args, returntype), aDefRhs(expressions))=>
44         {
45             val argumentsToDef = args match
46             {
47                 case None => List()
48                 case Some(aArguments(arglist)) => arglist.flatMap(arg =>
49                 {

```

```

50         val fromargs =
51             arguments.find(x=>x.id.value==arg.id.value)
52         val fromSame = inSameDefinition.find(x =>
53             x.id.value == arg.id.value)
54         fromargs match
55         {
56             case Some(_)=>fromargs
57             case None=>fromSame
58         }
59     }
60     //println("\n{deff: "+deff+"\nargumentsToDef:
61         "+argumentsToDef+"\nargs: "+args+"\n\nast:
62         "+ast+"\n\narguments: "+arguments+"}\n\n\n")
63     val aexpressions =
64         getCallsAnnotatedTreeInternal(expressions,
65             argumentsToDef, Some(deff))
66     Some(aDefinition(aDefLhs(desc, id, cppid, callingthread,
67         args, returntype), aDefRhs(aexpressions)))
68 }
69 case call @ aFunctionCallStatement(fid,_,args,_) =>
70     Some(annotateFunctionCall(call, arguments,
71         inSameDefinition, containingDefinition))
72 case z:IdentifierStatement=>Some(z)
73 case z:BasicValueStatement=>Some(z)
74 }
75 )
76 }
77
78 def annotateFunctionCall( functioncall:aFunctionCallStatement,
79     arguments:List[aDefLhs], inSameDefinition:List[aDefLhs],
80     containingDefinition:Option[aDefinition] ):
81     aFunctionCallStatement=
82 {
83
84     def annotateCallargs(args: Either[aCallarg,aNamedCallargs],
85         arguments:List[aDefLhs], inSameDefinition:List[aDefLhs],
86         containingDefinition:Option[aDefinition]):
87         Either[aCallarg,aNamedCallargs] =
88 {
89     args match
90     {
91         case Left(callarg)=>callarg match
92         {
93             case z:aFunctionCallStatement=>Left(annotateFunctionCall(z,
94                 arguments, inSameDefinition, containingDefinition))
95             case z:aStatement=>Left(z)
96         }
97         case Right(aNamedCallargs(callargs)) =>
98             Right(aNamedCallargs(callargs.map(namedcallarg =>
99                 namedcallarg.argument match
100                 {
101                     case z:aFunctionCallStatement =>
102                         aNamedCallarg(namedcallarg.id,
103                             annotateFunctionCall(z, arguments, inSameDefinition,
104                                 containingDefinition))

```

```

86         case aCallarg=>namedcallarg:aNamedCallarg
87     }
88 )))
89 }
90 }
91
92 val fid = functioncall.functionidentifier
93 val args = functioncall.args
94 if(fid=="actionPrint" || fid=="toString" || fid=="actionMutate")
95 {
96     val newargs = annotateCallargs(args, arguments,
97         inSameDefinition, containingDefinition)
98     aFunctionCallStatement(fid,fid,newargs,"Nothing")
99 }
100 else if(fid=="plus" || fid=="minus" || fid=="multiply" ||
101     fid=="divide")
102 {
103     val newargs = annotateCallargs(args, arguments,
104         inSameDefinition, containingDefinition)
105     aFunctionCallStatement(fid,fid,newargs,"Number") //TODO: Find
106         actual type like in typechecker. As it is, it only matters
107         if it is Nothing or not.
108 }
109 else if(fid=="equal" || fid=="lessThan")
110 {
111     val newargs = annotateCallargs(args, arguments,
112         inSameDefinition, containingDefinition)
113     aFunctionCallStatement(fid,fid,newargs,"Boolean")
114 }
115 else if(fid=="if")
116 {
117     val newargs = annotateCallargs(args, arguments,
118         inSameDefinition, containingDefinition)
119     aFunctionCallStatement(fid,fid,newargs,"Something") //TODO:
120         Find actual type like in typechecker. As it is, it only
121         matters if it is Nothing or not.
122 }
123 else
124 {
125     def removeInclusions(args: Either[aCallarg,aNamedCallargs],
126         ldeffargs:Option[aArguments]):
127         Either[aCallarg,aNamedCallargs] = args match
128     {
129         case Left(callarg)=>
130         {
131             ldeffargs match
132             {
133                 case Some(aArguments(defargs))=>
134                 {
135                     if(defargs.head.typestr.value == "Inclusion")
136                     {Left(NoArgs())}
137                     else
138                     {
139                         args
140                     }
141                 }
142             }
143         }
144         case None=>Left(NoArgs())
145     }

```

```

131         //case _=>Left(NoArgs())
132     }
133 }
134 case Right(aNamedCallargs(namedcallargs))=>
135 {
136     ldeffargs match
137     {
138         case Some(aArguments(defargs))=>
139         {
140             val mnewargs = ListBuffer():ListBuffer[aNamedCallarg]
141             for(i<-0 until defargs.length)
142             {
143                 if(defargs(i).typestr.value!="Inclusion")
144                 {
145                     mnewargs += namedcallargs(i)
146                 }
147             }
148             Right(aNamedCallargs(mnewargs.toList))
149         }
150         case None=>println("in
151             getCallsAnnotatedTreeInternal");scala.sys.exit()
152         //case _=>Left(NoArgs())
153     }
154 }
155 val ldeff = findinscope(Some(arguments), inSameDefinition,
156     containingDefinition.map(x=>x.leftside), fid)
157 val newargs = annotateCallargs(removeInclusions(args,
158     ldeff.args), arguments, inSameDefinition,
159     containingDefinition)
160 //println("ldeff.cppid.value: "+ldeff.cppid.value)
161 aFunctionCallStatement(fid, ldeff.cppid.value, newargs,
162     ldeff.returntype.value)
163 }
164 }
165
166 def indexlefts(in:List[aExpression]):List[aDefLhs]=
167 {
168     in.foldLeft(List():List[aDefLhs]) ((list,y)=> y match
169     {
170         case aDefinition(leftside, _)=>list :+ leftside;
171         case _=> list
172     }
173 )
174 }
175
176 def findinscope(arguments:Option[List[aDefLhs]],
177     inSameDefinition:List[aDefLhs],
178     enclosingDefinition:Option[aDefLhs], searchFor:String):aDefLhs=
179 {
180     val argsres = arguments match{ case
181         Some(args)=>args.filter(y=>y.id.value==searchFor); case
182         None=>List():List[aDefLhs]}
183     val inscoperes = inSameDefinition.filter(x=>x.id.value==searchFor)
184
185     val enclosingres = enclosingDefinition match
186     {

```

```

179     case None => List()
180     case Some(deff) => if (deff.id.value == searchFor) {List(deff)}
181         else {List()}
182   }
183   val res = argsres ++ inscoperes ++ enclosingres
184
185   if(res.length==0){println("{arguments:
186     "+arguments+"\n\ninSameDefinition:
187     "+inSameDefinition+"\n\nenclosingDefinition:
188     "+enclosingDefinition+"\n\nsearchFor:
189     "+searchFor+"}\n\n\n");scala.sys.exit()}
190   res.head
191 }
192
193 def getAnnotatedTreeInternal(ast:List[Expression],
194   topthreadcalls:List[FunctionCallStatement], hierarchy:String,
195   callingthread:Option[String]):List[aExpression] =
196 {
197   val topactions:List[aExpression] =
198   {
199     if(hierarchy=="")
200     {
201       val mess = topthreadcalls.map(threadcall=>threadcall.args
202         match
203         {
204           case Left(IdentifierStatement(argname)) =>
205           {
206             val deff = ast.filter(x => x match {case
207               Definition(DefLhs(ActionT(), IdT(thisargname), _,
208                 _),_) =>argname==thisargname; case _=> false})
209             getAnnotatedTreeInternal(deff, List(),
210               threadcall.functionidentifier,
211               Some(threadcall.functionidentifier)):
212               List[aExpression]
213           }
214           case Left(_)=> List():List[aExpression]
215           case Right(NamedCallargs(namedargs))=>
216           {
217             val deffs = namedargs.flatMap(namedarg=>
218               namedarg match
219               {
220                 case NamedCallarg(_, IdentifierStatement(argname))=>
221                 {
222                   ast.find(y=>y match{case
223                     Definition(DefLhs(ActionT(),
224                       IdT(thisargname), _, _) ,_) => argname ==
225                     thisargname; case _=> false})
226                 }
227                 case _=>None
228               }
229             )
230             getAnnotatedTreeInternal(deffs,List(),
231               threadcall.functionidentifier,
232               Some(threadcall.functionidentifier)):
233               List[aExpression]
234           }
235         }
236     }
237   }
238 }

```

```

217     }
218   ):List[List[aExpression]]
219
220   mess.foldLeft(List(): List[aExpression])((x,y) => x++y):
221     List[aExpression]
222 }
223 else
224 {
225   List()
226 }
227 }
228 val rest:List[aExpression] = ast.flatMap(x=>x match
229 {
230   case Definition(DefLhs(ThreadT(), id, args, returntype),
231     DefRhs(expressions)) =>
232   {
233     val aexps = getAnnotatedTreeInternal(expressions,
234       topthreadcalls.filter(x => x.functionidentifier ==
235         id.value), id.value, Some(id.value))
236     //println("\n"+args+"\n\n")
237     val newargs =
238       args.map(args=>aArguments(args.args.map(arg=>arg match
239         {
240           case Argument(id, TypeT("Inclusion")) =>
241             aArgument(id, id, TypeT("Inclusion"))
242           case Argument(argid, typee) =>
243             {
244               if (argid.value.startsWith("synchronized"))
245                 {
246                   aArgument(argid, argid, typee)
247                 }
248               else
249                 {
250                   aArgument(argid, IdT(id.value+"$"+argid.value),
251                     typee)
252                 }
253             }
254           }
255       })
256     //println(newargs+"\n\n\n")
257     Some(aDefinition(aDefLhs(ThreadT(), id, id, id.value,
258       newargs, returntype),aDefRhs(aexps)))
259   }
260 }
261
262 case Definition(DefLhs(FunctionT(), id, args, returntype),
263   DefRhs(expressions)) =>
264 {
265   val aexps = getAnnotatedTreeInternal(expressions,
266     topthreadcalls, hierarchy+id.value, callingthread)
267   val newargs = args.map(args=>aArguments(args.args.map(arg
268     => aArgument(arg.id, arg.id, arg.typeestr)))
269   Some(aDefinition(aDefLhs(FunctionT(), id,
270     IdT(id.value+"$"+hierarchy), "shouldn't matter",
271     newargs, returntype), aDefRhs(aexps)))
272 }

```

```

261     case Definition(DefLhs(ProgramT(),_,_,_),_) => None //we
262         don't really care about it...
263     case Definition(DefLhs(ActionT(), id, args, returntype),
264         DefRhs(expressions)) =>
265     {
266         if(hierarchy=="")
267         {
268             None
269         }
270         else
271         {
272             val aexps = getAnnotatedTreeInternal(expressions,
273                 topthreadcalls, hierarchy+id.value, callingthread)
274             val newargs = args.map(args=>aArguments(args.args.map(arg
275                 => aArgument(arg.id, arg.id, arg.typestr)))
276             Some(aDefinition(aDefLhs(ActionT(), id,
277                 IdT(id.value+"$"+hierarchy), callingthread match
278                 {case Some(z)=>z; case None=>"not found"}, newargs,
279                 returntype), aDefRhs(aexps)))
280         }
281     }
282     case FunctionCallStatement(fid,args)=>
283     {
284         def annotateCallarg(callarg:Callarg):aCallarg=
285         {
286             callarg match
287             {
288                 case z:aCallarg => z
289                 case FunctionCallStatement(fid,args)=>
290                 {
291                     val newargs:Either[aCallarg,aNamedCallargs] = args
292                     match
293                     {
294                         case Left(arg)=>Left(annotateCallarg(arg))
295                         case Right(NamedCallargs(arglist)) =>
296                             Right(aNamedCallargs(arglist.map(x =>
297                                 aNamedCallarg(x.id,
298                                 annotateCallarg(x.argument)) )))
299                     }
300                     aFunctionCallStatement(fid,"not filled
301                         out",newargs,"not filled out")
302                 }
303             }
304             Some(annotateCallarg(FunctionCallStatement(fid, args))):
305             Option[aExpression]
306         }
307         case z:IdentifierStatement=>Some(z)
308     }
309     ):List[aExpression]
310     rest++topactions
311 }
312 def getFunctionDeclarations(ast:List[aExpression]):(String,String) =
313 {

```

```

304 def actfunrecursivetranslate(cppid:IdT, callingthread:String,
305   args:Option[aArguments], returntype:TypeT,
306   expressions>List[aExpression]):Option[(String,String)] =
307   {
308     val signature = getFunctionSignature(cppid, args, returntype)
309     val functionstart = signature+"\n{"
310     val functionend = "\n}\n"
311     val generals = expressions.flatMap(
312       y=> y match
313       {
314         case aDefinition(leftside, rightside)=>None
315         case z:aFunctionCallStatement =>
316         {
317           if(z.returntype!="Nothing")
318           {
319             Some("return "+functioncalltranslator(z, callingthread)
320               + "; //returntype: "+z.returntype)
321           }
322           else
323           {
324             Some(functioncalltranslator(z, callingthread) + ";")
325           }
326         }
327         case IdentifierStatement(value) => Some("return "+value+";")
328         case StringStatement(value) => Some("return "+value+";")
329         case IntegerStatement(value) => Some("return
330           "+value.toString+";")
331         case DoubleStatement(value) => Some("return
332           "+value.toString+";")
333         case TrueStatement() => Some("return true;")
334         case FalseStatement() => Some("return false;")
335         //case _=> "not implemented" //println("Error in
336           gettopthreaddeclarations. Not implemented.");
337           scala.sys.exit()
338       }
339     ).foldLeft("")(x,y)=>x+"\n "+y)
340     val underfunctions = getFunctionDeclarations(expressions)
341     val body = functionstart+generals+functionend
342     //Some((signature+";",body))
343     Some((signature+";"+underfunctions._1, body+underfunctions._2))
344   }
345
346 val list = ast.flatMap(node=>node match
347 {
348   case aDefinition(aDefLhs(ThreadT(), id, cppid, _, args, _),
349     aDefRhs(expressions)) =>
350   {
351     val attributeNoreturn = if(
352       System.getProperty("os.name").startsWith("Windows") )
353       {"__declspec(noreturn)"} else{"[[noreturn]]"}
354       //Microsoft Visual C++ does not support C++11 attribute
355       syntax
356     val signature = attributeNoreturn+" static void
357       "+cppid.value+"()"
358     val functionstart = signature+"\n{"
359     val functionend = "\n}\n"

```



```

348     val (tailrecursestart, tailrecurseend) = ("while(true)\n{",
349         "\n}")
350
351     def changeNamesToCppOnes(in:aCallarg,
352         threadargs:Option[aArguments]):aCallarg = in match
353     {
354         case call:aFunctionCallStatement=>
355         {
356             val newargs:Either[aCallarg,aNamedCallargs] =
357                 call.args match
358             {
359                 case Left(callarg) =>
360                     Left(changeNamesToCppOnes(callarg,
361                         threadargs))
362                 case Right(aNamedCallargs(namedcallargs)) =>
363                     Right(aNamedCallargs(namedcallargs.map(
364                         namedcallarg =>
365                         aNamedCallarg(namedcallarg.id,
366                             changeNamesToCppOnes(
367                                 namedcallarg.argument, threadargs )))) )
368             }
369             aFunctionCallStatement(call.functionidentifier,
370                 call.cppfunctionidentifier, newargs,
371                 call.returntype)
372         }
373         case IdentifierStatement(value)=>
374         {
375             threadargs match
376             {
377                 case None => IdentifierStatement(value)
378                 case Some(aArguments(arglist)) =>
379                 {
380                     val arg = arglist.find(arg=>arg.id.value ==
381                         value) match{case Some(x)=>x;case None =>
382                         println("error in
383                             functioncallargmodifier");
384                             scala.sys.exit()}
385                     IdentifierStatement(arg.cppid.value)
386                 }
387             }
388         }
389         case _=>in
390     }
391
392     val generals = expressions.flatMap(
393     y=> y match
394     {
395         case aDefinition(leftside, rightside)=>None
396         case aFunctionCallStatement(id.value,_, callargs,_) =>
397         {
398             val updates = args match
399             {
400                 case None => ""
401                 case Some(aArguments(List(aArgument(argid,
402                     cppargid, _)))) =>
403                 {
404                     callargs match

```

```

388     {
389         case Left(callarg) =>
390         {
391             val newvalue = callargTranslator(callarg,
392                 id.value)
393             if (argid.value.startsWith("synchronized"))
394             {
395                 if( argid.value!=newvalue){"\nwe haven't
396                     figured out the correct way to handle
397                     this yet"}
398                 else{""}
399             }
400             else{"\n"+cppargid.value+" =
401                 "+callargTranslator(
402                     changeNamesToCppOnes(callarg, args),
403                     id.value )+"\n"}
404         }
405         case Right(_)=>"error in generating updates1"
406     }
407 }
408 }
409 case Some(aArguments(defargslist)) =>
410 {
411     callargs match
412     {
413         case Right(namedcallargs) =>
414         {
415             namedcallargs.value.foldLeft("\n")((l,r)=>
416             {
417                 val newvalue =
418                     callargTranslator(r.argument,
419                         id.value)
420                 if( r.id.value.startsWith( "synchronized"
421                     ) )
422                 {
423                     if (r.id.value.startsWith(
424                         "synchronized" ) && r.id.value !=
425                         newvalue) {l + "\nwe haven't
426                             figured out the correct way to
427                             handle this yet"}
428                     else{l}
429                 }
430                 else
431                 {
432                     val defarg = defargslist.find(defarg =>
433                         defarg.id.value == r.id.value)
434                     match{case Some(x) => x; case None
435                         => println("error in generating
436                             updates3"); scala.sys.exit()}
437                     l+defarg.cppid.value +" = "+
438                         callargTranslator(
439                             changeNamesToCppOnes( r.argument,
440                                 args ) , id.value) + "\n"
441                 }
442             }
443         }
444     }
445 }

```

```

425         case Left(_)=>"error in generating updates2"
426     }
427 }
428 }
429     Some("waitForRendezvous(\""+ cppid.value+"\");"
        +updates+ "\n continue;")
430 }
431 case z:aFunctionCallStatement =>
432 {
433     val modified = changeNamesToCppOnes(z, args) match
434     {
435         case a:aFunctionCallStatement => a
436         case _=> println("error when modifying function
            call");scala.sys.exit()
437     }
438     Some(functioncalltranslator(modified, id.value) + ";")
439 }
440 //case z:aFunctionCallStatement =>
441     Some(functioncalltranslator(z, id.value) + ";")
442 //case _=> "not implemented" //println("Error in
    gettopthreaddeclarations. Not implemented.");
443     scala.sys.exit()
444 }
445 ).foldLeft(")((x,y)=>x+"\n "+y)
446 val underfunctions = getFunctionDeclarations(expressions)
447 val body = functionstart + tailrecursestart + generals +
    tailrecurseend + functionend
448 Some((signature+";"+underfunctions._1,
    body+underfunctions._2))
449 }
450 case z:aFunctionCallStatement=>None
451 case z:IdentifierStatement=>None
452 case aDefinition(aDefLhs(ActionT(), id, cppid, callingthread,
    args, returntype), aDefRhs(expressions)) =>
    actfunrecursivetranslate(cppid, callingthread, args,
    returntype, expressions)
453 case aDefinition(aDefLhs(FunctionT(), id, cppid,
    callingthread, args, returntype),aDefRhs(expressions)) =>
    actfunrecursivetranslate(cppid, callingthread, args,
    returntype, expressions)
454 }
455 ):List[(String,String)]
456 list.foldLeft(("",""))((x,y)=>(x._1+"\n"+y._1,x._2+"\n"+y._2))
457 }
458
459
460
461 def getFunctionSignature(cppid:IdT, optargs:Option[aArguments],
    returntype:TypeT):String =
462 {
463     def argtranslator(arg:aArgument):String=
464     {
465         typetranslator(arg.typestr)+" "+arg.id.value
466     }
467     val argsString = optargs match

```

```

468 {
469     case None=>"
470     case Some(aArguments(List(arg))=>
471     {
472         if(arg.typestr.value!="Inclusion")
473         {
474             argtranslator(arg)
475         }
476         else{""}
477     }
478     case Some(aArguments(args))=>argtranslator(args.head) +
         args.tail.foldLeft(")((x,y)=>
479         if(y.typestr.value!="Inclusion"){x+", "+argtranslator(y)}
         else{x}
480     )
481 }
482 }
483
484 typetranslator(returntype)+" "+cppid.value+"("+argsString+")"
485 }
486
487 def typetranslator(in:TypeT):String =
488 {
489     in.value match
490     {
491         case "Integer"=>"int"
492         case "Double"=>"double"
493         case "String"=>"std::string"
494         case "Nothing"=>"void"
495         case "Inclusion"=>"shouldn't be here"
496         case "Boolean"=>"bool"
497         case _=>"not implemented"
498     }
499 }
500
501 def callargTranslator(callarg:aCallarg,
502     callingthread:String):String =
503 {
504     callarg match
505     {
506         case StringStatement(value)=>value
507         case IntegerStatement(value)=>value.toString
508         case DoubleStatement(value)=>value.toString
509         case TrueStatement()=>"true"
510         case FalseStatement()=>"false"
511         case IdentifierStatement(value)=>value
512         case call:aFunctionCallStatement =>
             functioncalltranslator(call: aFunctionCallStatement,
                 callingthread:String)
513         case NoArgs()=>" "
514     }
515 }
516
517 def functioncalltranslator(call:aFunctionCallStatement,
518     callingthread:String):String =
519 {
520     //println("in functioncalltranslator. call is "+call)

```

```

519 //if(call.functionidentifier=="plus"){println("found")}
520 //println("\n\n"+call)
521 call match
522 {
523   case aFunctionCallStatement("actionPrint",_,_
      Left(StringStatement(value)),_) => "print" + callingthread
      + ".push_back(" + value + ")"
524   case aFunctionCallStatement("actionPrint",_,_
      Left(IdentifierStatement(value)),_) => "print" +
      callingthread + ".push_back(std::to_string(" + value + "))"
525   case aFunctionCallStatement("actionPrint",_,_
      Left(x:aFunctionCallStatement),_) => "print" +
      callingthread + ".push_back(" +
      functioncalltranslator(x,callingthread) + ")"
526   case aFunctionCallStatement("toString",_,_
      Left(x:aFunctionCallStatement),_) => "std::to_string(" +
      functioncalltranslator(x,callingthread) + ")"
527   case aFunctionCallStatement("toString",_,_
      Left(IdentifierStatement(value)),_) => "std::to_string(" +
      value + ")"
528   case aFunctionCallStatement("toString",_,_
      Left(IntegerStatement(value)),_) => "std::to_string(" +
      value.toString + ")"
529   case aFunctionCallStatement("toString",_,_
      Left(DoubleStatement(value)),_) => "std::to_string(" +
      value.toString + ")"
530   case aFunctionCallStatement("toString",_,_
      Left(TrueStatement()),_) => "true"
531   case aFunctionCallStatement("toString",_,_
      Left(FalseStatement()),_) => "false"
532   case aFunctionCallStatement("equal", _,
      Right(aNamedCallargs(List(aNamedCallarg(IdT("left"),
      IntegerStatement(left)), aNamedCallarg(IdT("right"),
      IdentifierStatement(right))))) , _) => left.toString + " ==
      "+ right.toString
533   case aFunctionCallStatement("equal", _,
      Right(aNamedCallargs(List(aNamedCallarg(IdT("left"),
      IdentifierStatement(left)), aNamedCallarg(IdT("right"),
      IntegerStatement(right))))) ,_) => left.toString + " == "+
      right.toString
534   case aFunctionCallStatement("equal", _,
      Right(aNamedCallargs(List(aNamedCallarg(IdT("left"),
      StringStatement(left)), aNamedCallarg(IdT("right"),
      IdentifierStatement(right))))) , _) => left.toString + " ==
      "+ right.toString
535   case aFunctionCallStatement("equal", _,
      Right(aNamedCallargs(List(aNamedCallarg(IdT("left"),
      IdentifierStatement(left)),
      aNamedCallarg(IdT("right"),StringStatement(right))))) ,_) =>
      left.toString+" == "+right.toString
536   case aFunctionCallStatement("equal", _,
      Right(aNamedCallargs(List(aNamedCallarg(IdT("left"),
      IdentifierStatement(left)), aNamedCallarg(IdT("right"),
      IdentifierStatement(right))))) ,_) => left.toString+" ==
      "+right.toString
537

```

```

538     case aFunctionCallStatement("equal", _,
      Right(aNamedCallargs(List(aNamedCallarg(IdT("left"),
        IntegerStatement(left)), aNamedCallarg(IdT("right"), x:
          aFunctionCallStatement))))), _) => left.toString + " ==
539     "+functioncalltranslator(x, callingthread)
      case aFunctionCallStatement("equal", _,
        Right(aNamedCallargs(List(aNamedCallarg(IdT("left"), x:
          aFunctionCallStatement), aNamedCallarg(IdT("right"),
            IntegerStatement(right))))), _) =>
540     functioncalltranslator(x, callingthread) + " == " +
      right.toString
      case aFunctionCallStatement("equal", _,
        Right(aNamedCallargs(List(aNamedCallarg(IdT("left"),
          StringStatement(left)), aNamedCallarg(IdT("right"),
            x:aFunctionCallStatement))))), _) => left.toString + " ==
541     "+functioncalltranslator(x, callingthread)
      case aFunctionCallStatement("equal", _,
        Right(aNamedCallargs(List(aNamedCallarg(IdT("left"),
          x:aFunctionCallStatement), aNamedCallarg(IdT("right"),
            StringStatement(right))))), _) => functioncalltranslator(x,
542     callingthread) + " == " + right.toString
      case aFunctionCallStatement("equal", _,
        Right(aNamedCallargs(List(aNamedCallarg(IdT("left"),
          x:aFunctionCallStatement), aNamedCallarg(IdT("right"),
            IdentifierStatement(right))))), _) =>
543     functioncalltranslator(x, callingthread) + " == " +
      right.toString
      case aFunctionCallStatement("equal", _,
        Right(aNamedCallargs(List(aNamedCallarg(IdT("left"),
          IdentifierStatement(left)), aNamedCallarg(IdT("right"),
            x:aFunctionCallStatement))))), _) => left.toString + " ==
544     "+functioncalltranslator(x, callingthread)
      case aFunctionCallStatement("equal", _,
        Right(aNamedCallargs(List(aNamedCallarg(IdT("left"),
          x:aFunctionCallStatement), aNamedCallarg(IdT("right"),
            y:aFunctionCallStatement))))), _) =>
545     functioncalltranslator(x, callingthread) + " == " +
546     functioncalltranslator(y, callingthread)
      case aFunctionCallStatement("lessThan", _,
        Right(aNamedCallargs(List(aNamedCallarg(IdT("left"),
          IntegerStatement(left)), aNamedCallarg(IdT("right"),
            IntegerStatement(right))))), _) => left.toString + " <
547     "+right.toString
548     //TODO: Make better solution for commutative functions
549     //TODO: add more types that the comparison functions can accept
      case aFunctionCallStatement("actionMutate", _,
        Right(aNamedCallargs(List(aNamedCallarg(IdT("newValue"),
          IdentifierStatement(newval)),
          aNamedCallarg(IdT("variable"),
            IdentifierStatement(vari))))), _) => vari + " = " + newval
550     case aFunctionCallStatement("actionMutate", _,
        Right(aNamedCallargs(List(aNamedCallarg(IdT("newValue"),
          x:aFunctionCallStatement), aNamedCallarg(IdT("variable"),
            IdentifierStatement(vari))))), _) =>
551     {

```

```

552     "write" + vari.capitalize + " = " + functioncalltranslator(x,
553         callingthread)
554 }
555 case aFunctionCallStatement("plus",_,_,_) =>
556     basicmathcalltranslator(call, callingthread)
557 case aFunctionCallStatement("minus",_,_,_) =>
558     basicmathcalltranslator(call, callingthread)
559 case aFunctionCallStatement("multiply",_,_,_) =>
560     basicmathcalltranslator(call, callingthread)
561 case aFunctionCallStatement("divide",_,_,_) =>
562     basicmathcalltranslator(call, callingthread)
563 case aFunctionCallStatement("if",_,
564     Right(aNamedCallargs(List(aNamedCallarg(IdT("condition"),
565         condstat), aNamedCallarg(IdT("else"), elsestat),
566         aNamedCallarg(IdT("then"), thenstat))), _) =>
567 {
568     def translator(in:aStatement):String=
569     {
570         in match
571         {
572             case TrueStatement()=>"true"
573             case FalseStatement()=>"false"
574             case StringStatement(value)=>value
575             case IntegerStatement(value)=>value.toString
576             case DoubleStatement(value)=>value.toString
577             case IdentifierStatement(value)=>value //Correct
578                 behaviour? .....
579             case z:aFunctionCallStatement=>functioncalltranslator(z,
580                 callingthread)
581         }
582     }
583     condstat match
584     {
585         case TrueStatement()=>translator(thenstat)
586         case FalseStatement()=>translator(elsestat)
587         case _=>
588         {
589             translator(condstat)+" ? "+translator(thenstat)+" :
590             "+translator(elsestat)
591         }
592     }
593 }
594 case aFunctionCallStatement(funcid, cppfuncid, args, _) =>
595 {
596     val argstr = args match
597     {
598         case Left(callarg)=>callargTranslator(callarg,
599             callingthread)
600         case Right(aNamedCallargs(args)) =>
601         {
602             val first = callargTranslator(args.head.argument,
603                 callingthread)
604             val subsequent = args.foldLeft(")((x,y)=>x+",
605                 "+callargTranslator(y.argument, callingthread))
606             first+subsequent
607         }
608     }
609 }

```

```

595     cppfuncid+"("+argstr+")"
596   }
597   case _=> "not implemented"
598 }
599 }
600
601 def basicmathcalltranslator(call:aFunctionCallStatement,
602   callingthread:String):String=
603 {
604   val operator = if(call.functionidentifier=="plus"){ " + "}else
605     if(call.functionidentifier=="minus"){ " - "}else
606     if(call.functionidentifier=="multiply"){ " * "}else
607     if(call.functionidentifier=="minus"){ " / "}
608   call match
609   {
610     case aFunctionCallStatement(_,_,
611       Right(aNamedCallargs(callargs)),_) =>
612     {
613       val argstr = callargs.map(arg=>
614         {
615           arg match
616           {
617             case aNamedCallarg(_, IdentifierStatement(value)) =>
618               value
619             case aNamedCallarg(_, IntegerStatement(value)) =>
620               value.toString
621             case aNamedCallarg(_, DoubleStatement(value)) =>
622               value.toString
623             case aNamedCallarg(_, f:aFunctionCallStatement) =>
624               functioncalltranslator(f, callingthread)
625           }
626         }
627       ):List[String]
628       "(" + argstr(0) + operator + argstr(1) + ")"
629     }
630   }
631 }
632
633 def gettopthreadstatements(ast:List[Definition]):
634   List[FunctionCallStatement] =
635 {
636   ast.find(x => (x.leftside.description match {case ProgramT() =>
637     true; case _=> false})) match
638   {
639     case None => println("Error in getthreads. Should be caught by
640       checker."); scala.sys.exit()
641     case Some(res) =>
642     {
643       res.rightside.expressions.flatMap(x => x match
644       {
645         case x:FunctionCallStatement => if
646           (x.functionidentifier.startsWith("thread")) {Some(x)}
647         else {None}
648         case _ => None
649       })
650     }
651   }
652 }

```



```

638     }
639   }
640 }
641
642 def getprintlistdeclarations(topthreads:
643   List[FunctionCallStatement]): String=
644 {
645   val topthreadnames = topthreadnames.map(x=>x.functionidentifier)
646   var out = ""
647   for(i<-toppthreadnames)
648   {
649     out += "\nstatic std::list<std::string> print"+i+";"
650   }
651   out
652 }
653
654 def getStaticThreadArgs(atree:List[aExpression]):String =
655 {
656   atree.flatMap(exp=>exp match
657   {
658     case aDefinition(aDefLhs(ThreadT(), _, _, _,
659       Some(aArguments(args)), _) => Some(args.flatMap(arg =>
660       if (arg.typestr.value == "Inclusion" ||
661         arg.id.value.startsWith("synchronized"))
662       {
663         None
664       }
665       else
666       {
667         Some("static "+typetranslator(arg.typestr)+
668           "+arg.cppid.value+";\n")
669       }
670     ).fold("")(l,r)=>l+r)
671     case _=> None
672   }
673   ).fold("\n")(l,r)=>l+r)
674 }
675
676 def getmain(topthreads:List[FunctionCallStatement],
677   atree:List[aExpression]):String =
678 {
679   val threaddefs:List[aDefLhs] = atree.flatMap(exp=>exp match
680   {
681     case aDefinition(a @ aDefLhs(ThreadT(),_,_,_,_,_) =>
682       Some(a)
683     case _=>None
684   }
685   )
686
687   val threadargsSet:String = topthreadnames.map(toppthreadcall =>
688   {
689     val threaddefl = threaddefs.find(threaddefl =>
690       threaddefl.id.value == topthreadcall.functionidentifier)
691     match {case Some(x)=>x; case None=>println("error in
692       getmain");scala.sys.exit()}
693   }
694   )

```

```

686 topthreadcall.args match
687 {
688   case Left(callarg) =>
689   {
690     threaddefl.args match
691     {
692       case None=>List("")
693       case Some(aArguments(List(defarg)))=>
694       {
695         if (defarg.typestr.value == "Inclusion" ||
696             defarg.id.value.startsWith("synchronized"))
697         {
698           List("")
699         }
700         else
701         {
702           val modcallarg:aCallarg = callarg match
703           {
704             case a:aCallarg => a
705             case _=>println("error in getmain2(should be
706                             forbidden)");scala.sys.exit() //function
707                             calls in assignment in program statement
708                             doesn't make much sense and should be
709                             forbidden
710           }
711           List(defarg.cppid.value+" =
712                 "+callargTranslator(modcallarg:aCallarg,
713                                     "shouldn't be here")+";")
714         }
715       }
716     }
717   }
718   case Right(NamedCallargs(namedarglist)) =>
719   {
720     val defarglist = threaddefl.args match
721     {
722       case None => println("error in
723                             getmain4");scala.sys.exit()
724       case Some(aArguments(defarglist))=>defarglist
725     }
726     namedarglist.foldLeft(List():List[String])((list,
727         namedarg)=>
728     {
729       val modcallarg:aCallarg = namedarg.argument match
730       {
731         case a:aCallarg => a
732         case _=>println("error in getmain5(should be
733                             forbidden)");scala.sys.exit() //function calls
734                             in assignment in program statement doesn't make
735                             much sense and should be forbidden
736       }
737       val defarg = defarglist.find(defarg =>
738         defarg.id.value==namedarg.id.value) match {case
739         Some(x)=>x; case None=>println("error in
740         getmain6");scala.sys.exit()}

```

```

728         if (defarg.typestr.value == "Inclusion" ||
729             defarg.id.value.startsWith("synchronized"))
730         {
731             list
732         }
733         else
734         {
735             list :+ (defarg.cppid.value+" =
736                    "+callargTranslator(modcallarg:aCallarg,
737                    "shouldn't be here")+";")
738         }
739     }
740 }
741 ).fold(List(): List[String]) ((l1list,r1list) => l1list ++
742                                r1list).foldLeft("\n") ((str, sublist) => if (sublist!="")
743                                {str+"\n"+sublist} else{str})
744
745 var threadsStart = ""
746
747 for(i<-topthreads)
748 {
749     threadsStart = threadsStart + "\n" + "std::thread t" +
750         i.functionidentifier + " (" + i.functionidentifier + ");"
751 }
752
753 "int main()\n{\nrendezvousCounter.store(0);" + threadargsSet +
754     threadsStart + "\nwhile(true)\n {\n
755     std::this_thread::sleep_for(std::chrono::seconds(1)); \n}" +
756     "\n}"
757
758 def getsynchronizerfunction(synchvariables:List[Definition],
759                             toptthreads:List[FunctionCallStatement]):String=
760 {
761     var synchvariablestrings = ""
762
763     for(i<-synchvariables)
764     {
765         val name = i.leftside.id.value
766         synchvariablestrings += name + " = write" + name.capitalize +
767             ";\n"
768     }
769
770     var printstatements = ""
771     for(i<-topthreads)
772     {
773         val currentprintqueueName = "print" + i.functionidentifier
774         printstatements += "while(!"+currentprintqueueName+".empty())
775             {\nstd::cout << "+currentprintqueueName + ".front();
776             \n"+currentprintqueueName+".pop_front(); \n}\n"
777     }
778
779     ("static void waitForRendezvous(std::string name)
780 {

```

```

772     std::unique_lock<std::mutex> lk(rendezvousSyncMutex);
773     ++rendezvousCounter;
774     if (rendezvousCounter.load() < NUMTOPTHREADS)
775     {
776         cv.wait(lk);
777     }
778     else if (rendezvousCounter.load() == NUMTOPTHREADS)
779     {
780         ""
781         + printstatements + synchvariablestrings + ""
782         {
783             rendezvousCounter.store(0);
784             cv.notify_all();
785         }
786     }
787     else
788     {
789         std::cout << "error in wait for " << name << ". Rendezvouscounter
              out of bounds. RendezvousCounter = " <<
              rendezvousCounter.load() << "\n";
790         exit(0);
791     }
792 }""")
793 }
794
795 def getGlobalSynchVariableDeclarations(synchvariables:
    List[Definition]): String=
796 {
797     var synchdeclares = ""
798     for(i<-synchvariables)
799     {
800         val fumurtttype = i.leftside.returntype.value
801         val initialValue = i.rightside.expressions(0) match
802         {
803             case FunctionCallStatement(functionidentifier, args) => args
              match
804             {
805                 case Right(namedcallargs) =>
                    namedcallargs.value(0).argument match
806                 {
807                     case IntegerStatement(value) => value
808                     //case DoubleStatement(value) => value
809                     case _=> println("Error in
                        getGlobalSynchVariableDeclarations. Should be caught
                        by checker."); scala.sys.exit()
810                 }
811                 case _=> println("Error in
                        getGlobalSynchVariableDeclarations. Should be caught by
                        checker."); scala.sys.exit()
812             }
813             case _=> println(i.rightside.expressions(0).toString);
                println("Error in getGlobalSynchVariableDeclarations.
                    Should be caught by checker."); scala.sys.exit()
814         }
815         if (fumurtttype == "Integer")
816         {

```

```

817         synchdeclares += "\nstatic int " + i.leftside.id.value + " =
            " + initialValue + ";"
818         synchdeclares += "\nstatic int write" +
            i.leftside.id.value.capitalize + " = " + initialValue +
            ";"
819     }
820 }
821 synchdeclares
822 }
823
824 def getsynchronizedvariables(ast: List[Definition]):
    List[Definition] =
825 {
826     ast.find(x => (x.leftside.description match {case ProgramT() =>
            true; case _=> false})) match
827     {
828         case None => println("Error in getthreads. Should be caught by
            checker."); scala.sys.exit()
829         case Some(res) =>
830         {
831             res.rightside.expressions.flatMap(x => x match
832             {
833                 case x:Definition => x.leftside.description match {case
                    SynchronizedVariableT() => Some(x); case _=> None}
834                 case _=> None
835             })
836         }
837     }
838 }
839
840 }

```

C.8 Error.scala

```

1 package fumurtCompiler
2
3 import scala.util.parsing.input.Position
4 //import scala.util.parsing.input.NoPosition
5
6 case object Global extends Position
7 {
8     def column:Int = 0
9     def line:Int = 0
10    protected def lineContents:String = "global position"
11 }
12 case class Source(val line:Int, val column:Int, val
    lineContents:String) extends Position
13
14 case class FumurtError(val position:Position, val message:String)
15 {
16     override def toString:String=
17     {
18         position.toString + ": " + message + "\n" + position.longString +
            "\n"

```

19
20

}
}