NTNU – Trondheim
Norwegian University of
Science and Technology

A programming language with deterministic threading

Tormod Gjeitnes Hellen

Spring 2015

MASTER THESIS
Department of Engineering Cybernetics
Norwegian University of Science and Technology

Supervisor : Sverre Hendseth

Abstract

It is determined that [somethings]

Contents

Chapter 1

Introduction

First and foremost, you should write about the most interesting or important parts of your project. Devote most space and time to this. For example:

What design choices did you have along the way, and why did you make the choices you made? What was the most difficult part of the project? Why was it difficult? How did you overcome the difficulties? Did you discover anything novel? What did you learn?

Set the scene and problem statement/specification. Provide the motivation for reading this report. Introduce the structure of report (what you will cover in which chapters).

This report describes a master thesis done as a part of the MSc Engineering Cybernetics course at NTNU. It was written in the spring semester of 2015. It assumes that readers have prior experience working with a programming language and knows some common nomenclature.

It is commonly understood that writing software is hard and that writing mutlithreaded 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 judgement and trade-offs and the latter is a highly challenging exercise in software engineering.

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 (parallel but not concurrent code 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 Background chapter contains information needed to understand the rest of the report. [more here]

The report layout adheres to a standard set by University College London[3], modified in consultation with superviser.

Chapter 2

Background

You should provide enough background to the reader for them to understand what the project is all about. For example:

What the reader needs to know in order to understand the rest of the report. Examiners like to know that you have done some background research and that you know what else has been done in the field (where relevant). Try to include some references. Related work (if you know of any) What problem are you solving? Why are you solving it? How does this relate to other work in this area? What work does it build on?

For 'research-style' projects - ones in which a computational technique (for example neural networks, genetic algorithms, finite element analysis, ray tracing) is used to explore or extend the properties of a mathematical model, or to make predictions of some kind - it may be a good idea to split this chapter into two shorter ones, one covering the computational technique itself and one the area of application.

The Examiners are just as interested in the process you went through in performing your project work as the results you finally produced. So, make sure your reports concentrate on why you made the particular choices and decisions that you did. We are looking for reasoned arguments and for critical assessment. This is especially so where design, implementation and engineering decisions have been made not just on technical merit but under pressure of non-functional requirements and external influences.

2.1 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 [5]
- 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 [17]. To ameliorate this problem, a variety of machanisms 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 analysability. 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 multi-threaded programming[19]. 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[citation needed]. 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 deviced for dealing with the two concurrency problems in a systematic manner, to the author's knowledge:

- Actors [12]
- CSP [13]
- Transactional memory[11]
- Synchronous programming[4]

In the end a decision was made in favor of using the synchronous programming paradigm. There are tradeoffs associated with chosing 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 synchronicity abstraction requires all operations to use the same amount of time.

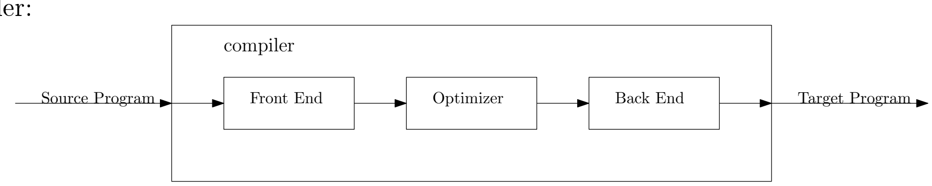
Synchronous programming therefore has substantial problems, yet for single-machine systems it presents a way to achieve multi-threaded performance and achitecture but with single-threaded predictability and therefore debugability. While the other abstractions place the responsibility for correct concurrent behaviour on the programmer, synchronous programming takes care 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. Distributed systems have to correctly handle hardware failures, so loss of information is an acceptible failure mode for actors. However, this makes actors unsuitable for real-time systems as recovering from data loss and unpredictable memory usage are unacceptable tradeoffs. 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.

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 [8], illustrates the structure of a typical compiler:



Consider the steps:

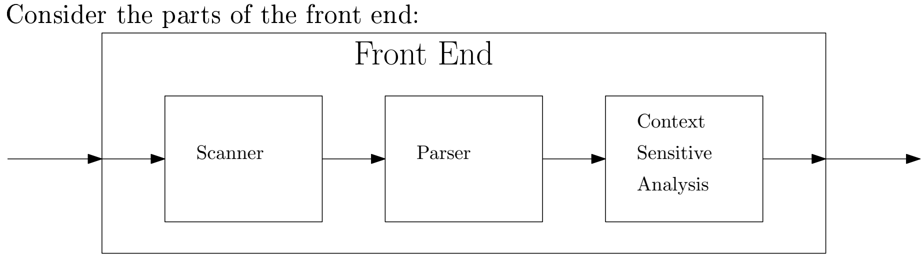
1. The front end accepts source text and transforms it into an intermediate representation that is easier to work with. It is generally independent of the target format.

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 compiler described in 6 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 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 in 6 it is done here. In the case where a language has type inference, this step may emit a modified intermediate representation.

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 ouput 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.

2.3.1 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".

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

```
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 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:

```
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 the curly brackets signify that their contents can be repeated one or more time. 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:

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

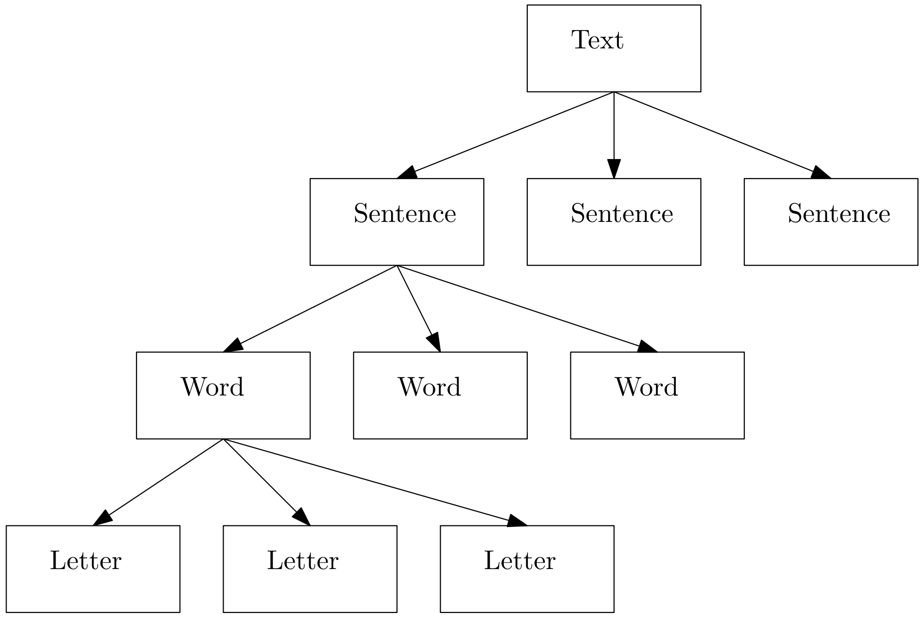
Parentheses allows grouping of sequences. Here, it allows us to alternate between sequences of symbols rather than just single symbols. Finally:

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

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

2.3.1.1 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 abreviated 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:

```
class Text(val sentences:List[Sentence])
class Sentence(val words:List[Word])
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[**?**]. 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 6.2).

A Note on Conflicting terminology: Unfortunately there is a case of conflicting terminology concerning the term "parser". The parser is refered 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 [1].

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 dissallow backtracking.
- * applies the parser to the left as many times as it is succesful, moving on at failure
- + applies the parser to the left as many times as it is succesful, 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 succesful 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 The Compiler Implementation Language

In order to understand the code in the compiler, which is included in appendix C

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 only uses 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.
- There are sequences, like Array or List

- lines need not be terminated with ";" (but it is optional)

```
object HelloWorld
{
  def main(args:Array[String]):Unit =
  {
    println("Hello, world!")
  }
}
```

2.5.3 Creating and Using Objects

- All values are objects, even native types
- Functions are objects, but methods are not
  - 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.
- Var lets you create mutable references to objects
- Val lets you create immutable references to objects

```
scala> def int1 = 3
int1: Int

scala> val int2 = 2
int2: Int = 2

scala> var int3 = 7
int3: Int = 7

scala>//reassignment to a def is illegal

scala> int1 = int1+1
<console>:8: error: value int1_= is not a member of object $iw
       int1 = int1+1
       ^

scala>//so is reassignment to val

scala> int2 = int2+1
<console>:8: error: reassignment to val
       int2 = int2+1
            ^

scala>//reassignment to var is completely ok

scala> int3 = int3+1
int3: Int = 8

```

```
scala> int1+int2+int3
res0: Int = 13

scala>//all values are objects

scala> int1.+(int2.+(int3))
res1: Int = 13

scala>//even functions

scala> val square = ((x:Int) => x*x)
square: Int => Int = <function1>

scala> square(3)
res2: Int = 9

scala> square.toString
res3: String = <function1>

scala>//but methods are not

scala> def cube(x:Int) = x*x*x
cube: (x: Int)Int

scala> cube(3)
res4: Int = 27

scala> cube.toString
<console>:9: error: missing arguments for method cube;
follow this method with '_' if you want to treat it as a partially applied function
              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 it, its values or both.
    * Pattern matching looks like this:

```
val x:String = input match
{
  case TypeA("specific string") => "specific string"
  case TypeB(anystring, otherstring) => anystring + " " + otherstring
```

```
    case TypeB(_,otherstring) => "only care about "+otherstring
    case TypeB(_,_) => "only care about type"
    case reference:TypeA => "the object looks like this: "+reference.toString
    case reference @ TypeA(str) => "both a reference and the constructor parameter"
    //case reference:TypeA(str) => "this is unfortunately a syntax error"
}
```

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

```
scala>//classes in scala function mutch like classes in Java

scala> class A(int:Int, str:String)
defined class A

scala> val a = A(3,"a string")
<console>:7: error: not found: value A
       val a = A(3,"a string")
               ^

scala> val a = new A(3,"a string")
a: A = A@66ae2a84

scala>//case classes, on the other hand, have more functionality. Their constructors are called like normal functions

scala> case class B(str:String, int:Int)
defined class B

scala> val b = B("other string", 5)
b: B = B(other string,5)

scala>//and one can pattern match on them

scala> case class C(double:Double, int:Int)
defined class C

scala> val c = C(3.0, 3)
c: C = C(3.0,3)

scala> def matchfunc(in:Any):Unit = in match
     | {
     |    case B(string,integer) => println(string + integer.toString)
     |    case x:C => println(x.double.toString+x.int.toString)
     |    case _ => println("unknown type")
     | }
matchfunc: (in: Any)Unit

scala> matchfunc(b)
other string5
```

```
scala> matchfunc(c)
3.03
```

2.5.5 Inheritance

- A trait is an interface, a class with only abstract methods, 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

```
scala> trait Super
defined trait Super

scala> trait Side
defined trait Side

scala> trait Side2
defined trait Side2

scala> case class Sub(int:Int) extends Super with Side with Side2
defined class Sub
```

Inheritance is used very sparingly in this report.

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 reprecented 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
  - 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.

```
scala> var int = 0
int: Int = 0

scala> while(int<10){println(int); int=int+1}
0
1
2
3
4
5
6
7
8
9

scala> for(x <- 0 until 10){println(x)}
0
1
2
3
4
5
6
7
8
9

scala> val l = List(0,1,2,3,4,5,6,7,8,9)
l: List[Int] = List(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)

scala> for(x <- l){println(x)}
0
1
2
3
4
5
6
7
8
9
```

For fold, foldLeft, foldRight, map and flatMap examples, see 3.3

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 has a fixed type. So "(3, "str", 5.0)" is of type (Int, String, Double). specific places in the set are accessed using "set._n", where n is the 1-indexed index.

```
scala>//Option:

scala> def maybeSquare(in:Option[Int]):Option[Int] = in.map(x =>
    x*x)
maybeSquare: (in: Option[Int])Option[Int]

scala> maybeSquare(Some(3))
res0: Option[Int] = Some(9)

scala> maybeSquare(None)
res1: Option[Int] = None

scala>//Either:

scala> def squareOrCube(in:Either[Int,Int]) = in match
     | {
     |   case Left(x) => x*x
     |   case Right(x) => x*x*x
     | }
squareOrCube: (in: Either[Int,Int])Int

scala> squareOrCube(Left(3))
res2: Int = 9

scala> squareOrCube(Right(3))
res3: Int = 27

scala>//set:

scala> def change(in:(Int, String, Double)):(Int, String, Double)
    = (in._1*in._1, in._2+"ing", in._3)
change: (in: (Int, String, Double))(Int, String, Double)

scala> change((3, "str", 5.0))
res4: (Int, String, Double) = (9,string,5.0)
```

2.6 Deterministic Multithreading

All material here is based on [15] unless otherwise stated.

Deterministic multithreading is an active area of research. Two components are necessary for determinism[15]:

- A deterministic logical clock, which orders synchronization operations deterministically

- A deterministic memory consistency model, which ensures unsynchronized load operations have deterministic results

2.6.1 Deterministic Logical Clock

There are two main approaches to this:

- Round-robin scheduling
- Instruction-count based scheduling

Round robin scheduling means that for each lock, all threads using that lock needs to pause whenever the lock is acquired

2.6.2 Deterministic Memory Consistency Model

2.7 Prior Art

2.7.1 Esterel

2.7.2 CONSEQUENCE

Chapter 3

Knowledge Dividends

3.1 Regular Expressions

The parsers in the scanner operate at the string-level and are based on regular expressions, though the author had no knowledge of how they worked before writing this thesis.

3.2 C++11

3.3 Functional Programming

The compiler includes only two loops, instead the code relies primarily on functions such as:

- fold: supplied with a function which produces a single value from two input values, all of the same type, fold repeatedly uses this to produce a single value from a list. Can be executed in parallel.

```
scala> (0 to 2).fold(0)((left,right) => left+right)
res1: Int = 3

scala> (0 to 2).par.fold(0)((left,right) => left+right)
res2: Int = 3
```

- 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.

```
scala> (0 to 9).foldLeft("numbers")((string,number) =>
    string+number.toString)
res1: String = numbers0123456789

scala> (0 to 9).foldRight("numbers")((number,string) =>
    number.toString+string)
```

```
res2: String = 0123456789numbers
```

- map

```
scala> (0 to 9).map(x=>x*x)
res1: scala.collection.immutable.IndexedSeq[Int] = Vector(0,
    1, 4, 9, 16, 25, 36, 49, 64, 81)

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

- flatMap

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

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

Together:

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

3.4 Parser Combinators

Before starting to write the thesis, the mere existence of parser combinators were outside the scope of the author's knowledge. Both the scanner and parser in the compiler described herein is built with parser combinators.

Chapter 4

Specification

Elaboration of the problem

Initially, the goal of this thesis was to create a fundamentally new approach to managing concurrency, wherein the programmer would manually schedule the execution of tasks at compile time. Tasks would be allowed to write to special variables which would be used in lieu of final ones if the task could not finish in the alloted time frame. This effort was abandoned because of the burden it would impose on the programmer, the perceived difficulty of implementation and the unsatisfactory failure modes. Instead, it was decided that an approach belonging to the tradition of synchronous programming would be preferable. Given the importance of a familiar superficialities for language adoption[16], it was decided that the language should have a familiar C/Algol-style syntax, rather than invent or adopt something less common.

4.1 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.

4.2 Runtime Execution Model

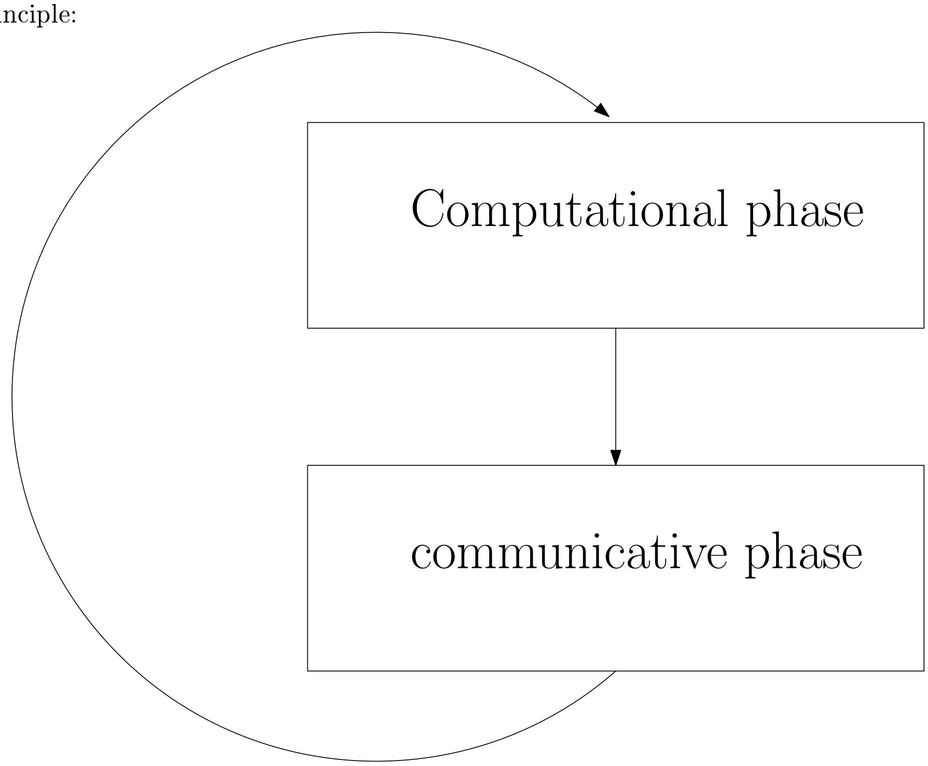
The goal of the programming language is to make a multithreaded program behave as predictably as were it singlethreaded and, more generally, to help create reliable applications. A corollary of this is that only changes of state that are visible to a single thread can happen concurrently. All IO and inter-thread communication are required happen in a statically determined sequence. There are several ways to do this. CONSEQUENCE[15] and similar systems built using an instruction-based logical clock[18] provide superior performance to round-robin systems. However, these are for the time being an active area of research, require a deviation from standard C++ and IO sequence depends on compilation target, requiring programmer intervention where this is undesirable.

In the interest of future relevance, it was decided to implement multithreaded behavior in a standard-only way. One way to do this is to have the program have two alternating phases:

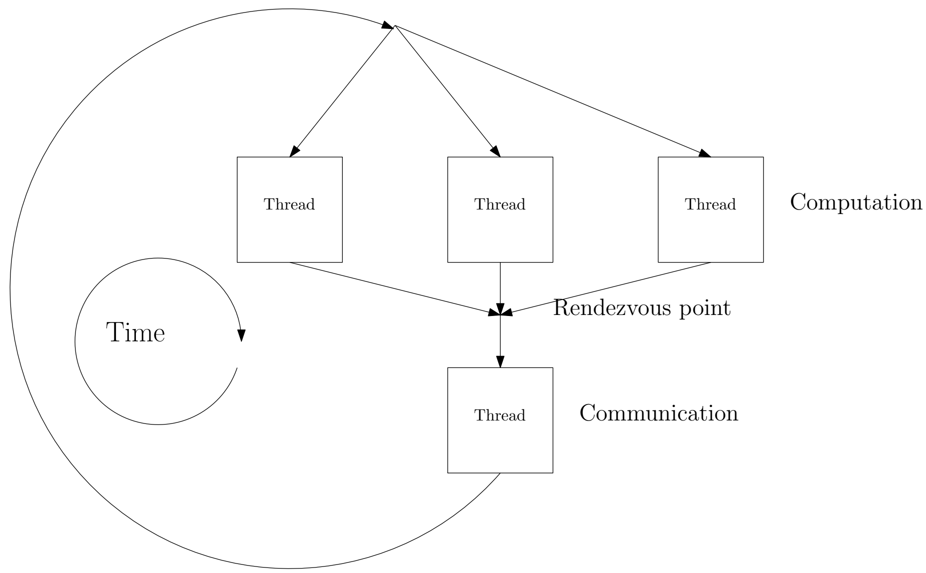
- Computational phase: In which computations local to a thread are performed.
- Communicative phase: In which IO is effected and shared variables are updated, all in a single-threaded manner.

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. In the terms from 2.6, all memory stores are synchronized using a round-robin deterministic clock and global lock. This is somewhat reminiscent of a low-tech Dthreads[14].

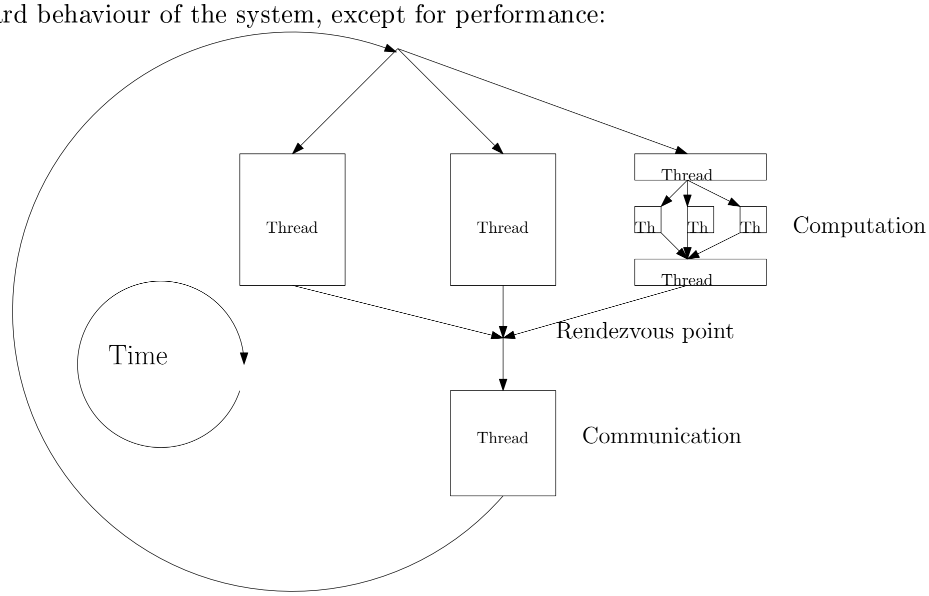
Using this scheme, the application appears to be single threaded both to itself and to the rest of the world, all the while enabling separation of concerns and better utilization of multi-core systems. The following figure illustrates the principle:



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 behaviour 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[9], is easy to use and widely applicative to many problems.[6] Indeed, map-and-fold is intensely used in the Fumurt compiler. 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.

4.3 Inter-thread communication

4.4 Syntax

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

- 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.

4.4.1 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:

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

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

4.4.2 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:

```
action actionPrintFoo:Nothing =
{
  actionPrint("  FOO   ")
}
```

Similarly thread names begin with “thread” and synchronized variable names begin with “synchronized”.

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

```
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 value.

  – 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.

4.5 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:

```
action actionA:Nothing =
{
  b()
}
action actionB:Nothing =
{
  c()
}
action actionC:Nothing =
{
  actionPrint("string")
}
```

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

```
action actionA(b:Inclusion, c:Inclusion):Nothing =
{
  b(c=c)
}
action actionB(c:Inclusion):Nothing =
{
```

```
  c()
}
action actionC:Nothing =
{
  actionPrint("string")
}
```

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.

4.6 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. Their arguments are nameless, which kind of sabotages the point of having named arguments for functions a little
3. How to define operator precedence? For math operators there's convention, but otherwise this may be confusing for users of operators

A prime example of unhelpful operator behaviour is found in 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 case-by-case basis using parentheses. Ambiguous use of operators are not allowed.

4.7 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.

4.7.1 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[17]. Loops that are as safe as tail recursion while being almost as friendly as common loops are possible:

```
value y:Int = 5

value x:Int = loop(y=y,x=y)
{
  if(
  condition=(y>0),
  then=
  {
  x = x*y
  y = y-1
  continue
  },
  else=break)
}
```

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 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.

4.8 Types

4.8.1 Classes

In trying to be familiar, it is desireable 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:

```
class IntAndString(int:Integer, string:String) =
{
  function combine:String = {concatenate(left=toString(int),
      right=string)}
}

value x = IntAndString(int=3, string="something")
```

```
actionPrint(x.combine)
actionPrint("==")
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.

4.8.2 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.

```
interface IntAndString(int:Integer, string:String)
//or
class IntAndString(int:Integer, string:String)

class IntAndStringAndBool(int:Integer, string:String,
    bool:Boolean) implements IntAndString
```

4.8.3 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.

4.9 Program Declaration

The program declaration is meant to give a high level overview of the behaviour 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.

4.10 Built-in Functions

Fumurt provides the following built-in functions:

- toString(x), defined for all types
-

Chapter 5

Analysis and Design

If your project involves designing a system, give a good high-level overview of your design.

In many projects, the initial design and the final design differ somewhat. If the differences are interesting, write about them, and why the changes were made.

If your design was not implemented fully, describe which parts you did implement, and which you didn't. If the reason you didn't implement everything is interesting (eg it turned out to be difficult for unexpected reasons), write about it.

5.1 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[10], and C is often used. Though many modern languages would be suitable for this, a wishlist 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.

5.2 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:

- A 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.

5.3 Synchronization Mechanisms in The Intermediate Language

Our execution model formulated in 4.2 needs 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.

6. A main function is defined, inside of which:

   (a) rendezvousCounter is set to 0, 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(

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 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. Consider the salient details:

```
static void waitForRendezvous(std::string name)
{
  std::unique_lock<std::mutex>
      lk(rendezvousSyncMutex);
  ++rendezvousCounter;
  if(rendezvousCounter.load() < NUMTOPTHREADS)
  {
    cv.wait(lk);
  }
  else if (rendezvousCounter.load() == NUMTOPTHREADS)
  {
    while(!printthreadPrintHello.empty())
    {
      std::cout << printthreadPrintHello.front();
      printthreadPrintHello.pop_front();
    }
        /*similarly for other thread print lists*/
    synchronizedNumber = writeSynchronizedNumber;
    rendezvousCounter.store(0);
    cv.notify_all();
  }
  /*abnormal situation diagnostics mechanism here*/
}
```

5.4 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, the only rule that required annotation was the rule for determining the C++ names of function. There are three aspects to the naming:

1. Actions and functions that are in other functions need to get new names and the hierarchy needs to be flattened

2. Actions need to be demultiplexed, as their C++ code 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

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

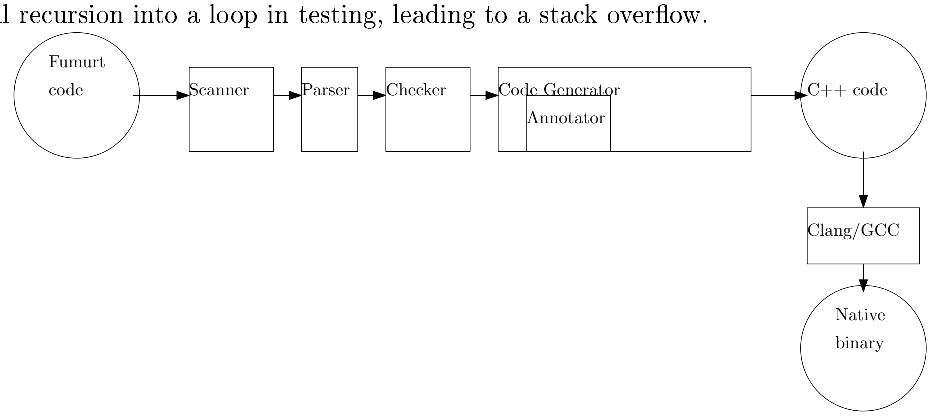
Chapter 6

Implementation

Give code details (not a complete listing, but descriptions of key parts). Discuss the most important/interesting aspects. It probably won't be possible to discuss everything - give a rationale for what you do discuss.

6.1 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 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 a stack overflow.



Consider the steps:

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.

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 from the parser is given to the code generator, which produces C++ code conforming to the C++11 standard.

5. The Clang C family compiler[7] is used to compile the C++ code to native binaries.

6.2 Scanner

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:

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

The parsers are then combined into the final scanner using the alternate operator"[?].

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

```
abstract class Token() extends Positional
abstract class DefDescriptionT() extends Token
abstract class BasicValueT() extends Token
abstract class SyntaxT() extends Token

case class TrueT() extends BasicValueT {override def toString =
    "true"}
```

Positional[2] 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 sparser to serve as scanner at compile time

- x*Parser: Parser[XT]* Parses that particular type of token, for example *newlineParser: Parser[NewlineT]*

6.3 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.

6.3.1 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 (ISO/IEC 14977) for the grammar, as implemented:

```
prog = paddedDef, {paddedDef}, EoF;
paddedDef = {"\n"}, def, {"\n"};
def = deflhs, "=", {"\n"}, defrhs;
deflhs = defdescription, id, args, ":", type;
args = ("(", id, ":", type, {subsequentArg}) | "";
subsequentArg = ",", id, ":", type;
defrhs = "{", {"\n"}, expression, {("\n", {"\n"}, expression)},
    {"\n"}, "}";
expression = def | statement;
statement = functionCall | basicStatement | identifierStatement;
callargs = "(", (namedcallargs|callarg), ")";
callarg = statement | "";
namedcallargs = namedcallarg, subsequentnamedcallarg,
    {subsequentnamedcallarg};
subsequentnamedcallarg = ",", namedcallarg;
namedcallarg = id, "=", callarg;
functionCall = id, callargs;
identifierStatement = id;
defdescription = "program" | "action" | "thread" | "function" |
    "value";
basicStatement = boolean | string | integer | float;
float = integer, ".", digit, {digit};
integer = "0" | (digit excluding zero, {digit});
digit excluding zero = "1" | "2" | "3" | "4" | "5" | "6" | "7" |
    "8" | "9" ;
digit = "0" | digit excluding zero ;
upper case = "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" ;
lower case = "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" ;
id = lower case, {(upper case | lower case)}
type = upper case, {(upper case | lower case)}
```

6.3.2 Code

This is where the grammar is encoded into the program:

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

```
def paddedDefParser:Parser[Definition] = { newlineParser.* ~>
    defParser <~ newlineParser.* }
/*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:

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

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:

```
class Expression() extends Positional
trait Callarg extends Positional
trait Statement extends Expression
trait BasicValueStatement extends Statement with Callarg with
    aCallarg with aStatement

case class Definition(val leftside:DefLhs, val rightside:DefRhs)
    extends Expression
case class DefLhs(val description:DefDescriptionT, val id:IdT, val
    args:Option[Arguments], val returntype:TypeT)
case class Arguments(val args:List[Argument])
case class Argument(val id:IdT, val typestr:TypeT)
case class DefRhs(val expressions:List[Expression] )
case class Empty();
case class DefDescription(val value:Token)
case class NamedCallarg(id:IdT, argument:Callarg) //extends Callarg
case class NamedCallargs(val value:List[NamedCallarg])
case class NoArgs() extends Callarg with aCallarg

case class StringStatement(val value:String) extends
    BasicValueStatement
case class IntegerStatement(val value:Int) extends
    BasicValueStatement
case class DoubleStatement(val value:Double) extends
    BasicValueStatement
case class TrueStatement() extends BasicValueStatement
case class FalseStatement() extends BasicValueStatement
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 head of the parsers, from which the parser combinator library will generate the final parser
- x*Parser: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 using *defParser:Parser[Definition]* to parse the definition part.

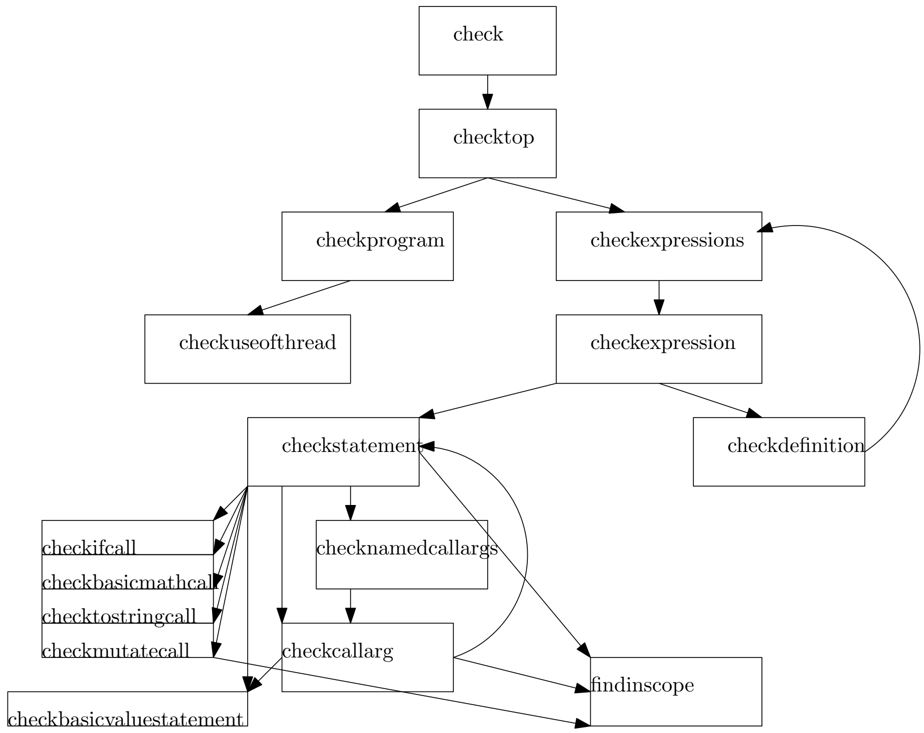
Class List

- *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
- Different classes forming parts of the AST, for example *class Definition(val leftside:DefLhs, val rightside:DefRhs).*

6.4 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(in:List[Definition]):Option[List[FumurtError]]* is the interface to the rest of the program. Takes in a AST and returns a list of errors, if there are any.

- *checktop(in:List[Definition], basicFunctions:List[DefLhs]): List[FumurtError]* 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(program:Definition, topleveldefs:List[DefLhs], basicFunctions:List[DefLhs]): List[FumurtError]* checks the program statement. Uses checkuseofthread and checks whether there are any calls to non-threads or definition of non-synchronized variables.

  - *checkuseofthread(program:Definition, thread:DefLhs):List[FumurtError]* checks that the thread given is actually called in the program statement. Declaring a thread and failing to call it is an error.

- *checkexpressions(tree:List[Expression], containingdefinition:Option[Definition], containingdefinitionarguments:Option[List[DefLhs]], basicFunctions:List[DefLhs]):List[FumurtError]* 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(tocheck:Expression, containingdefinition:Option[Definition], arguments:Option[List[DefLhs]], basicFunctions:List[DefLhs], inSameDefinition:List[DefLhs]):List[FumurtError]* checks an individual expression. Uses *checkstatement* and *checkdefinition*

- *checkstatement(tocheck:Statement, containingdefinition:DefLhs, arguments:Option[List[DefLhs]], basicFunctions:List[DefLhs], inSameDefinition:List[DefLhs], expectedreturn:TypeT): List[FumurtError]* 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(ifcall:FunctionCallStatement, expectedtype:TypeT, containingdefinition:DefLhs, arguments:Option[List[DefLhs]], basicFunctions:List[DefLhs], inSameDefinition:List[DefLhs]):List[FumurtError]* 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, of course.

- *checkmutatecall(call:FunctionCallStatement, expectedtype:TypeT, containingdefinition:DefLhs, arguments:Option[List[DefLhs]], basicFunctions:List[DefLhs], inSameDefinition:List[DefLhs]):List[FumurtError]* checks that the variable is a synchronized variable and otherwise has the same type as the new value

- *checkbasicmathcall(call:FunctionCallStatement, expectedtype:TypeT, containingdefinition:DefLhs, arguments:Option[List[DefLhs]], basicFunctions:List[DefLhs], inSameDefinition:List[DefLhs]):List[FumurtError]* checks the four basic math operators, with special attention to the return type when double and int are mixed

- *checktostringcall(call:FunctionCallStatement, expectedtype:TypeT, containingdefinition:DefLhs, arguments:Option[List[DefLhs]], basicFunctions:List[DefLhs], inSameDefinition:List[DefLhs]):List[FumurtError]* checks that there is only one argument and that the expected type is String

- *checknamedcallargs(calledfunction:DefLhs, namedcallargs:List[NamedCallarg], containingdefinition:DefLhs, arguments:Option[List[DefLhs]], basicFunctions:List[DefLhs], inSameDefinition:List[DefLhs]):List[FumurtError]* 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(expectedtype:TypeT, arg:Callarg, containingdefinition:DefLhs, arguments:Option[List[DefLhs]], basicFunctions:List[DefLhs], inSameDefinition:List[DefLhs]):List[FumurtError]* checks a call argument. Makes sure the type is correct. Uses checkbasicvaluestatement and checkstatement.

- *checkbasicvaluestatement(expectedtype:TypeT, basicstatement:BasicValueStatement, role:String):List[FumurtError]* checks that the type is correct.
- *checkdefinition(tocheck:Definition, containingdefinition:Option[DefLhs], arguments:Option[List[DefLhs]], basicFunctions:List[DefLhs]): List[FumurtError]*
- *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(arguments:Option[List[DefLhs]], inSameDefinition:List[DefLhs], basicfunctions:List[DefLhs], enclosingDefinition:Option[DefLhs], searchFor:String):Either[String, DefLhs]* finds a left side of the definition in the current scope with the same name as that which is searched for.

6.5 Code generator

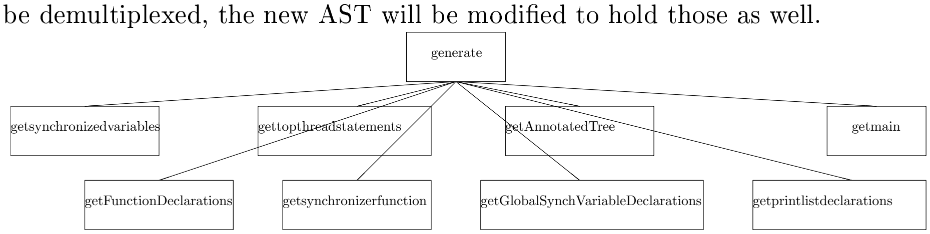
Function List

- *generate(ast:List[Definition]):String* generates the final C++ code from the Fumurt AST
- *getAnnotatedTree(ast:List[Expression], topthreadcalls:List[FunctionCallStatement]):List[aExpression]* Returns an annotated version of the supplied AST. This version has the final C++ names for functions and function calls
- *getCallsAnnotatedTreeInternal(ast:List[aExpression], arguments:List[aDefLhs], containingDefinition:Option[aDefinition]):List[aExpression]* returns an annotated version of the AST with final C++ names for function calls. Requires that function names have been annotated first
- *annotateFunctionCall(functioncall:aFunctionCallStatement, arguments:List[aDefLhs], inSameDefinition:List[aDefLhs], containingDefinition:Option[aDefinition]):aFunctionCallStatement* annotates a single function call
  - annotateCallargs(args:Either[aCallarg,aNamedCallargs], arguments:List[aDefLhs], inSameDefinition:List[aDefLhs], containingDefinition:Option[aDefinition]):Either[aCallarg,aNamedC annotates that function calls call arguments. Since call arguments can be function calls, this is often recursive.
  - *removeInclusions(args:Either[aCallarg,aNamedCallargs], ldeffargs:Option[Arguments]):Either[aCall* removes inclusion arguments from functions, since these have no purpose in C++
- *indexlefts(in:List[aExpression]):List[aDefLhs]* indexes DefLhs's like in the checker, but with the annotated types.

- *findinscope(arguments:Option[List[aDefLhs]], inSameDefinition:List[aDefLhs], enclosingDefinition:Option[aDefLhs], searchFor:String):aDefLhs* same as the version in the checker, but with annotated types.
- *getAnnotatedTreeInternal(ast:List[Expression], topthreadcalls:List[FunctionCallStatement], hierarchy:String, callingthread:Option[String]):List[aExpression]* returns an AST with with final C++ names for functions
- *getFunctionDeclarations(ast:List[aExpression]):(String,String)* gets the functions, in C++, from the AST
  - *actfunrecursivetranslate(cppid:IdT, callingthread:String, args:Option[Arguments], returntype:TypeT, expressions:List[aExpression]):Option[(String,String)]* gets function body and signature of a function corresponding to the arguments as well as all functions defined in the body of the definition.
- *getFunctionSignature(cppid:IdT, optargs:Option[Arguments], returntype:TypeT):String* constructs a C++ function signature from the arguments
  - *argtranslator(arg:Argument):String* translates an argument as used in defining a function
- *typetranslator(in:TypeT):String* translates (basic) Fumurt types to their C++ equivalents
- *callargTranslator(callarg:aCallarg, callingthread:String):String* translates a call argument to C++ equivalent
- *functioncalltranslator(call:aFunctionCallStatement, callingthread:String):String* translates function call
- *basicmathcalltranslator(call:aFunctionCallStatement, callingthread:String):String*
- *gettopthreadstatements(ast:List[Definition]):List[FunctionCallStatement]* gets the C++ statements spawning the threads.
- *getprintlistdeclarations(topthreads:List[FunctionCallStatement]):String* gets the printList declarations. These are lists in which strings to be printed are kept.
- *getmain(topthreads:List[FunctionCallStatement]):String* gets the main function. This only spawns the threads and then goes to sleep
- *getsynchronizerfunction(synchvariables:List[Definition], topthreads:List[FunctionCallStatement]):String* gets the mostly static and hand-written function that performs all actions during the communication phase
- *getGlobalSynchVariableDeclarations(synchvariables:List[Definition]):String* gets the C++ declarations of the synchronized variables

- *getsynchronizedvariables(ast:List[Definition]):List[Definition]* gets the definitions of the synchronized variables, so that they can later be used in *getGlobalSynchVariableDeclarations*

Classes The generator holds 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).* In cases where functions need to be demultiplexed, the new AST will be modified to hold those as well.



6.5.1 Annotator

Chapter 7

Testing

Test plan – how the program/system was verified. Put the actual test results in the Appendix. This section is useful if your project is more on the software engineering side than research focused.

7.1 Hello World

A simple repeating Hello World is written like this:

```
program helloworld:Nothing =
{
  threadPrintHelloWorld()
}

thread threadPrintHelloWorld:Nothing =
{
  actionPrint("Hello World\n")
  threadPrintHelloWorld()
}
```

Which prints Hello World forever:

```
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
/*and so on*/
```

7.2 Full program test

The following Fumurt code:

```
program p:Nothing =
{
  synchronized variable synchronizedNumber:Integer = {synchronized(variable=0, writer=threadPrintHello)}
  threadPrintHello(synchronizedNumber)
  threadPrintWorld(synchronizedNumber)
  threadPrintLol(actionPrintFoo=actionPrintFoo, integerIdentity=integerIdentity)
}

thread threadPrintWorld(synchronizedNumber:Integer):Nothing =
{
  actionPrint("world ")
  actionPrint(toString(synchronizedNumber))
  threadPrintWorld(synchronizedNumber)
}

thread threadPrintHello(synchronizedNumber:Integer):Nothing =
{
  actionPrint(toString(synchronizedNumber))
  actionPrint(" Hello ")
  actionMutate(variable=synchronizedNumber, newValue=plus(left=synchronizedNumber, right=1))
  threadPrintHello(synchronizedNumber)
}

thread threadPrintLol(actionPrintFoo:Inclusion, integerIdentity:Inclusion):Nothing =
{
  action actionPrintLol:Nothing =
  {
    actionPrint("  LOL   ")
  }

  actionPrintLol()
  actionPrintFoo(integerIdentity)
  threadPrintLol(actionPrintFoo=actionPrintFoo, integerIdentity=integerIdentity)
}

action actionPrintFoo(integerIdentity:Inclusion):Nothing =
{
  action actionPrintFooo:Nothing =
  {
    actionPrint("  FOOO  ")
  }
  actionPrint("  FOO   ")
  actionPrintFooo()
  actionPrint(toString(integerIdentity(5)))
  actionPrint("  ")
  actionPrint(if(condition=true, then=toString(6), else=toString(3)))
  actionPrint("\n")
```

```
}

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

Gets compiled to the following C++11 code:

```
#include <iostream>
#include <thread>
#include <string>
#include <atomic>
#include <condition_variable>
#include <list>
#include <chrono>


#define NUMTOPTHREADS 3

[[noreturn]] static void threadPrintWorld();
[[noreturn]] static void threadPrintHello();
[[noreturn]] static void threadPrintLol();
void actionPrintLol$threadPrintLol();
int integerIdentity$(int x);
void actionPrintFoo$threadPrintLol();
void actionPrintFooo$threadPrintLolactionPrintFoo();

static int synchronizedNumber = 0;
static int writeSynchronizedNumber = 0;
static std::list<std::string> printthreadPrintHello;
static std::list<std::string> printthreadPrintWorld;
static std::list<std::string> printthreadPrintLol;
static std::atomic<int> rendezvousCounter;
static std::mutex rendezvousSyncMutex;
static std::condition_variable cv;static void
    waitForRendezvous(std::string name)
{
  std::unique_lock<std::mutex> lk(rendezvousSyncMutex);
  ++rendezvousCounter;
  if(rendezvousCounter.load() < NUMTOPTHREADS)
  {
    cv.wait(lk);
  }
  else if (rendezvousCounter.load() == NUMTOPTHREADS)
  {
    while(!printthreadPrintHello.empty()){
std::cout << printthreadPrintHello.front();
printthreadPrintHello.pop_front();
}
while(!printthreadPrintWorld.empty()){
std::cout << printthreadPrintWorld.front();
printthreadPrintWorld.pop_front();
}
while(!printthreadPrintLol.empty()){
std::cout << printthreadPrintLol.front();
printthreadPrintLol.pop_front();
}
synchronizedNumber = writeSynchronizedNumber;

    {
```

```
      rendezvousCounter.store(0);
      cv.notify_all();
    }
  }
  else
  {
    std::cout << "error in wait for " << name << ".
        Rendezvouscounter out of bounds. RedezvousCounter = " <<
        rendezvousCounter.load() << "\n";
    exit(0);
  }
}



[[noreturn]] static void threadPrintWorld()
{while(true)
{
  printthreadPrintWorld.push_back("world ");
  printthreadPrintWorld.push_back(std::to_string(synchronizedNumber));
  waitForRendezvous("threadPrintWorld");
  continue;
}
}

[[noreturn]] static void threadPrintHello()
{while(true)
{
  printthreadPrintHello.push_back(std::to_string(synchronizedNumber));
  printthreadPrintHello.push_back(" Hello ");
  writeSynchronizedNumber = (synchronizedNumber + 1);
  waitForRendezvous("threadPrintHello");
  continue;
}
}

[[noreturn]] static void threadPrintLol()
{while(true)
{
  actionPrintLol$threadPrintLol();
  actionPrintFoo$threadPrintLol();
  waitForRendezvous("threadPrintLol");
  continue;
}
}

void actionPrintLol$threadPrintLol()
{
  printthreadPrintLol.push_back("  LOL   ");
}

int integerIdentity$(int x)
{
  return x;
}

void actionPrintFoo$threadPrintLol()
```

```
{
  printthreadPrintLol.push_back("   FOO   ");
  actionPrintFooo$threadPrintLolactionPrintFoo();
  printthreadPrintLol.push_back(std::to_string(integerIdentity$(5)));
  printthreadPrintLol.push_back("  ");
  printthreadPrintLol.push_back(std::to_string(6));
  printthreadPrintLol.push_back("\n");
}

void actionPrintFooo$threadPrintLolactionPrintFoo()
{
  printthreadPrintLol.push_back("   FOOO  ");
}


int main()
{
rendezvousCounter.store(0);
std::thread tthreadPrintHello (threadPrintHello);
std::thread tthreadPrintWorld (threadPrintWorld);
std::thread tthreadPrintLol (threadPrintLol);
while(true)
{
std::this_thread::sleep_for(std::chrono::seconds(1));
}
}
```

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

```
0 Hello world 0  LOL     FOO     FOOO  5  6
1 Hello world 1  LOL     FOO     FOOO  5  6
2 Hello world 2  LOL     FOO     FOOO  5  6
3 Hello world 3  LOL     FOO     FOOO  5  6
4 Hello world 4  LOL     FOO     FOOO  5  6
5 Hello world 5  LOL     FOO     FOOO  5  6
6 Hello world 6  LOL     FOO     FOOO  5  6
7 Hello world 7  LOL     FOO     FOOO  5  6
8 Hello world 8  LOL     FOO     FOOO  5  6
9 Hello world 9  LOL     FOO     FOOO  5  6
10 Hello world 10  LOL     FOO     FOOO  5  6
11 Hello world 11  LOL     FOO     FOOO  5  6
12 Hello world 12  LOL     FOO     FOOO  5  6
13 Hello world 13  LOL     FOO     FOOO  5  6
14 Hello world 14  LOL     FOO     FOOO  5  6
15 Hello world 15  LOL     FOO     FOOO  5  6
16 Hello world 16  LOL     FOO     FOOO  5  6
17 Hello world 17  LOL     FOO     FOOO  5  6
18 Hello world 18  LOL     FOO     FOOO  5  6
19 Hello world 19  LOL     FOO     FOOO  5  6
20 Hello world 20  LOL     FOO     FOOO  5  6
21 Hello world 21  LOL     FOO     FOOO  5  6
22 Hello world 22  LOL     FOO     FOOO  5  6
23 Hello world 23  LOL     FOO     FOOO  5  6
24 Hello world 24  LOL     FOO     FOOO  5  6
25 Hello world 25  LOL     FOO     FOOO  5  6
26 Hello world 26  LOL     FOO     FOOO  5  6
27 Hello world 27  LOL     FOO     FOOO  5  6
28 Hello world 28  LOL     FOO     FOOO  5  6
```

```
29 Hello world 29   LOL      FOO      FOOO   5   6
30 Hello world 30   LOL      FOO      FOOO   5   6
/*and so on...*/
```

Observe that all output is deterministic.

7.3 Error messages

Error messages are useful to detect errors in the program at compile time. Changing the source in 7.2 to the following erroneous program allow us to test them:

```
program p:Nothing =
{
  synchronized variable synchronizedNumber:Integer =
      {synchronized(variable=0, writer=threadPrintHello)}
  threadPrintHello(synchronizedNumber)
  threadPrintWorld(synchronizedNumber)
  threadPrintLol(actionPrintFoo=actionPrintFoo,
      integerIdentity=integerIdentity)
}

thread threadPrintWorld(synchronizedNumber:Integer):Nothing =
{
  actionPrint("world ")
  actionPrint(toString(synchronizedNumber))
  threadPrintWorld(synchronizedNumber)
}

thread threadPrintHello(synchronizedNumber:Integer):Nothing =
{
  actionPrint(toString(synchronizedNumber))
  actionPrint(" Hello ")
  actionMutate(variable=synchronizedNumber,
      newValue=plus(left=synchronizedNumber, right=1))
  threadPrintHello(synchronizedNumber)
}

thread threadPrintLol(actionPrintFoo:Inclusion,
    integerIdentity:Inclusion):Nothing =
{
  action actionPrintLol:Nothing =
  {
    actionPrint("   LOL    ")
  }

  actionPrintLol()
  actionPrintFoo(integerIdentity)
  threadPrintLol(actionPrintFoo=actionPrintFoo,
      integerIdentity=integerIdentity)
}

function printFoo(integerIdentity:Inclusion):Nothing =
{
  action actionPrintFooo:Nothing =
```

```
  {
    actionPrint("  FOOO  ")
  }
  actionPrint("  FOO   ")
  actionPrintFooo()
  actionPrint(toString(integerIdentity(5.0)))
  actionPrint("  ")
  actionPrint(if(condition=true, then=toString(6), else=toString(3)))
  actionPrint("\n")
}

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

The following type errors are produced:

```
32.3: actionPrintFoo not found
  actionPrintFoo(integerIdentity)

  ^

33.33: actionPrintFoo not found
  threadPrintLol(actionPrintFoo=actionPrintFoo, integerIdentity=integerIdentity)

                                ^

38.3: actions cannot be defined in  functions
  action actionPrintFooo:Nothing =

  ^

44.40: Call argument type should be Integer. Call argument type was Double
  actionPrint(toString(integerIdentity(5.0)))

                                        ^

Four errors found
```

7.4 Not Implemented

- Loops
- User-defined types
- Boolean functions and logic
- exit function

Chapter 8

Results

This covers different areas to the 'Testing' chapter, and is appropriate for 'research style' projects. For such projects this chapter should detail the types of experiments/simulations that were carried out with the code written. Why were certain experiments carried out but not others? What were the important parameters in the simulation and how did they affect the results? If there are very many graphs and tables associated with this chapter they may be put in the Appendix, but it is generally better to keep these close to the text they illustrate, as this is easier for the reader.

Chapter 9

Conclusion, Evaluation and Further Work

What have you achieved? Give a critical appraisal (evaluation) of your own work - how could the work be taken further (perhaps by another student next year)?

9.1 Suggestions for Future Work

9.1.1 Further Grammar Development

The “defRhs” in the current grammar is essentially a multiline statement. It would be beneficial if the line between statement and defRhs could be eliminated. For instance, multiline statements could replace function calls in if-calls or a single line statement without brackets could serve as right hand side in a definition.

Adding operators seem beneficial, as do

9.1.2 More IO

At the time of writing, the only type of IO that Fumurt has is printing to terminal. It goes without saying that this is inadequate for serious usage.

9.1.3 Boolean Functions

9.1.4 Custom Types, Generics

9.1.5 Evaluate the Possibility of Using Instruction-Based Logical Clock

Systems such as CONSEQUENCE[15] may make it possible to obtain greater performance in cases where scheduling requirements can be relaxed.

Bibliography

[1] 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.

[2] 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.

[3] 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.

[4] Berry, G., and Gonthier, G. The esterel synchronous programming language: Design, semantics, implementation. *Science of computer programming 19*, 2 (1992), 87–152.

[5] 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.

[6] 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.

[7] Clang, C. language family frontend for llvm, 2005.

[8] Cooper, K., and Torczon, L. *Engineering a compiler*. Elsevier, 2011.

[9] Dean, J., and Ghemawat, S. Mapreduce: simplified data processing on large clusters. *Communications of the ACM 51*, 1 (2008), 107–113.

[10] 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.

[11] Herlihy, M., and Moss, J. E. B. *Transactional memory: Architectural support for lock-free data structures*, vol. 21. ACM, 1993.

[12] 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.

[13] Hoare, C. A. R. Communicating sequential processes. *Communications of the ACM 21*, 8 (1978), 666–677.

[14] 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.

[15] 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.

[16] Meyerovich, L. A., and Rabkin, A. S. Empirical analysis of programming language adoption. In *ACM SIGPLAN Notices* (2013), vol. 48, ACM, pp. 1–18.

[17] 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.

[18] Olszewski, M., Ansel, J., and Amarasinghe, S. Kendo: efficient deterministic multithreading in software. *ACM Sigplan Notices 44*, 3 (2009), 97–108.

[19] Sutter, H. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's journal 30*, 3 (2005), 202–210.

Appendix A

System manual

This should include all the technical details (where is the code? what do you type to compile it? etc) that would enable a student to continue your project next year, to be able to amend your code and extend it.

Appendix B

User manual

This should give enough information for someone to use what you have designed and implemented.

Appendix C

Code listing

Your code should be well commented. In order not to use up too many pages of your maximum 120 on code, you may like to use the 'a2ps' Unix facility, which allows you to put two pages of code onto one side of paper - see the Unix 'man' pages for details. If you have a great deal of code, and including all of it would take you over the page limit, you can make the rest available on a floppy disk or CD-ROM. You will need to bring in two copies of any disks or CDs you include when you hand in your project report, one to go with each copy of your project.