CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Region-based Memory Management and Actor Model Concurrency

An initial study of how the combination performs

Master's thesis in Computer science and engineering

Robert Krook

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2020

Master's thesis 2020

Region-based Memory Management and Actor Model Concurrency

An initial study of how the combination performs

Robert Krook

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2020

Region-based Memory Management and Actor Model Concurrency
An initial study of how the combination performs
Robert Krook

Supervisor: John Hughes, Department
Examiner: Mary Sheeran, Department

Master's Thesis 2020
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2020

Region-based Memory Management and Actor Model Concurrency
An initial study of how the combination performs
Robert Krook
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Modern computer systems and the requirements we place upon them are vastly different from those of early systems. With the emergence of *Internet of Things* (IoT) devices, the number of devices with hard, real-time deadlines have increased greatly. The presence of a garbage collector does not resonate well with such systems, as garbage collectors typically use a *stop-the-world* approach. The problem is amplified further in languages such as Erlang, where it is commonplace to spawn many processes. In Erlang, each process has its own heap which is individually garbage collected.
A common design of an IoT device is a board with several different sensors and peripherals. The presence of so many garbage collectors should be enough to deter us from using Erlang to program IoT devices, but the idea of writing small, isolated programs to manage each of the sensors is appealing.
Another memory management principle, one which could eliminate the need for a garbage collector, is called Region-based Memory Management. This thesis has investigated how well current Region-based Memory Management techniques work when they are applied to a setting that implements Actor Model concurrency. To investigate this an Actor Model concurrency-library has been implemented in Standard ML and compiled with the MLKit compiler - a compiler which uses Region-Based memory management.
Evaluating the speed and memory performance of the library shows that the combination performs poorly. The Region-inference algorithm employed by the compiler struggles with identifying when data can be deallocated and retains most data throughout the execution of a program. We identify some key problems and propose how they could be solved.
We conclude that current techniques are not well suited in a setting with Actor Model concurrency. We cannot, however, say if the combination of Region-based memory management and the Actor concurrency model works well or not, as further research is required.

Keywords: Region-based memory management, Standard ML, Actor-Model Concurrency, Functional programming

Acknowledgements

I would like to extend my gratitude towards my supervisor, John Hughes, for his guidance and the many insightful discussions we have had. I have learned more than I ever anticipated.
To my family and friends, for all the support and understanding.

Robert Krook, Gothenburg, June 2020

Contents

List of Figures

List of Tables

1
Introduction

Since the arrival of modern computers, memory has up until recently been a scarce resource. As a result of this scarcity, programmers were initially forced to do manual memory management to avoid running out of memory. Manual memory management is a very tedious and error-prone activity. Accidentally deallocating just one value before it is safe to do so can result in an entire program crashing.
In an effort to simplify manual memory management in Lisp, John McCarthy invented the concept of garbage collection[16]. It was first described in a paper dubbed *Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I*. As the section describing the garbage collector was awarded just half a column in the seminal paper, it was demonstrably not known at the time just how important this concept would become.
Today garbage collection is a very popular feature in many programming languages. Being freed from the burden of manual memory management, however, comes at a price. In order for the garbage collector to perform a sweep of the memory associated with a program, that whole program needs to be suspended. Sweeping the memory may take varying amounts of time to do depending on the state of the memory. As the world digitalises further, the requirements we place on machines change as well. Some systems have hard deadlines to meet regarding what must happen when and having a garbage collector take an arbitrary amount of time away at arbitrary points in time from the main program is not acceptable.
The situation is amplified further in Erlang[3]. Erlang is a functional language that has a mature implementation of the actor concurrency model. Every thread that is spawned has its own heap, resulting in a setting where threads are completely separate as there is no shared memory. The heap of a thread, however, has to be carefully managed to avoid running out of memory. In Erlang this is done by having the threads perform their own individual garbage collection.
Now, instead of a situation where there is a single program which might be halted by a single garbage collector, there is in Erlang a garbage collector associated to each thread in the program. Reasoning about the behaviour of a program becomes much more involved, and reasoning about the behaviour of individual processes is easier.
A memory management principle which retains the property of not requiring deallocation annotations from the programmer, while simultaneously not relying on a garbage collector, is called Region-based memory management. Region-based memory management was introduced by Mads Tofte and Jean-Pierre Talpin[20][19]. The idea behind Region-based memory management is that objects are allocated in regions rather than directly on the heap. While individual objects within a region

cannot be deallocated, the region as a whole - and all objects within it - can be deallocated. The programmer does not have to specify allocation and deallocation points as this information is inferred by the compiler. The compiler will infer this information by performing region inference on the source program, annotating it with region constructs
If the region inference is clever enough, values might not have to remain allocated long past their lifetimes. In this case, there is not always a need for a garbage collector, as allocations and deallocations become explicit in the code after region inference. Regions are eventually deallocated, making the memory they previously occupied available for use. One downside of this, however, is that objects might remain alive for longer than required if the deallocation point is far in the future.
This thesis presents the results obtained when investigating how well current Region-based memory management techniques work when they are applied to coroutines as they are described by the actor concurrency model. The setting is one where threads use Region-based memory management to manage their memory instead of relying on a garbage collector. To assess how such a setting performs using current Region-based memory management techniques, a library has been implemented with the functionality specified by the actor concurrency model. The work has been carried out in Standard ML, as there is a complete implementation of Standard ML that uses regions to manage memory. This implementation is called the MLKit[18, 20]. The MLKit is a well-documented compiler which enables profiling of regions at runtime, to assess the amount of memory being consumed by a program.
In contrast to Erlang, which is dynamically typed, Standard ML is statically typed. This causes some trouble when sending messages, as we cannot write a send or receive function that can handle messages of any type. In other statically typed languages, such as Cloud Haskell[8], an alternative interface using *typed channels* is implemented. We, however, have solved this problem by serialising messages, allowing us to transmit messages of almost any type.

2
Background

This chapter will give an explanation of all the concepts involved in this thesis. To explore the Region-based memory management part of the thesis the MLKit[18] compiler for Standard ML is used as it has a mature implementation of Region-based memory management. First, an overview of Standard ML is presented, followed closely by an explanation of Region-based memory management using examples written in Standard ML. Lastly, an overview of actor model concurrency is given.

2.1 Standard ML

This section will give an overview of the Standard ML programming language. First, the way Standard ML passes parameters to functions is explained, followed by some words about how effects are expressed and evaluated in Standard ML. Lastly, a brief but thorough overview of the module system is given.

2.1.1 Parameter passing in Standard ML

Standard ML is a general-purpose functional programming language. It is strongly typed and implements a call-by-value semantics. Call by value semantics specifies that an expression should be evaluated before it is passed as an argument to a function, and is the most common parameter passing style found in other programming languages. In a call by name setting the argument would not be evaluated before application, and every reference to the argument within the function body would evaluate the argument again. To give an example, consider a function which will add its arguments and another that will double its single argument.

```
fun add a b  = a + b
fun double a = a + a
```

In a language with a call by name semantics, evaluation of the expression `double (add 2 2)` could happen like this, depending on the reduction semantics.

```
> double (add 2 2)
> (add 2 2) + (add 2 2)
> (2 + 2) + (add 2 2)
> 4 + (add 2 2)
> 4 + (2 + 2)
> 4 + 4
```

```
> 8
> 8
```

The above example illustrates that the argument, `add 2 2`, is evaluated twice. If the argument is a computation which consumes a lot of resources - e.g consumes a lot of memory or takes a long time to normalise - this is not a favourable situation. If the argument to a function performs a side-effect that side-effect would be performed each time the value is evaluated. On the other hand, if the argument is not used by the function body there is almost no overhead with call by name semantics, as the argument would not be evaluated.
With a call by value semantics, the argument is evaluated just once - before the function is applied to it - and subsequent references to the argument within the function body would read the computed value. If the argument is referenced many times within the function body a lot of work is saved. Contrary to the call by name semantics, however, if the argument is not used in the function body it would have been evaluated unnecessarily.

```
> double (add 2 2)
> double (2 + 2)
> double 4
> 4 + 4
> 8
```

2.1.2 Effect-Handling and Concurrency in Standard ML

Standard ML is not pure, meaning that a program is free to perform any side-effect. Consider the innocent looking program below.

```
fun feed_the_cat () = (
  (* fire missiles *);
  ())
```

The type of the above function is `feed_the_cat : unit -> unit`, which does not indicate that a side-effect will occur if it is applied to `()`, whereas, in reality, it would fire the missiles. The analogy of firing the missiles is meant to indicate that `IO` effects are irrevocable.
In Haskell[15] the opposite is true; an effectful computation would need to annotate its type to indicate that it may perform a side-effect.

```
feed_the_cat () = do
  {- fire the missiles -}
  return ()
```

The type of the above Haskell function is `feed_the_cat :: () -> IO ()`, where it is clear from the result type that the function may perform any `IO` effect at all. Any computation that `feed_the_cat` is a part of needs to annotate its result type to be that of `IO`.

Standard ML is inherently single-threaded. There are no concurrency primitives defined in the formal definition of the language. Any compiler for Standard ML which offers parallelism or concurrency primitives deviates from the language specification[17].

2.1.3 Standard ML's Module System

Standard ML has a complex module system. The programmer can reason about modules in the code and structure code in a hierarchical way. This system is used by defining signatures and implementing structures.
A signature describes the interface of a module. The signature can name abstract types of which the representation is unknown by just observing the signature. Values of such an abstract type can then only be instantiated by calling functions defined by the signature. Consider the signature below.

```
signature PROP =
  type ('p, 'q) con     (* p /\ q *)
  type ('p, 'q) dis     (* p \/ q *)
  type ('p, 'q) implies (* p => q *)
```

The signature defines three abstract types representing logical and, or and implication. We proceed by defining the type signatures for the introduction and elimination rules, indicating how values of such types are created and destroyed.

```
  (* Introduction rules *)
  val con_intro     : 'p * 'q    -> ('p, 'q) con
  val dis_intro1    : 'p         -> ('p, 'q) dis
  val dis_intro2    : 'q         -> ('p, 'q) dis
  val implies_intro : ('p -> 'q) -> ('p, 'q) implies

  (* Elimination rules *)
  val con_elim1 : ('p, 'q) con -> 'p
  val con_elim2 : ('p, 'q) con -> 'q
  val dis_elim  : (('p, 'r) implies, ('q, 'r) implies) con
                  ->
                  ('p, 'q) dis -> 'r
```

We can also define some laws we expect to hold for propositional logic, as is done below. The definition of the signature is terminated with the `end` keyword.

```
  (* (p => q) /\ (p => r)
  *  --------------------
  *      p => q /\ r      *)
  val composition : (('p, 'q) implies, ('p, 'r) implies) con
                    ->
                    ('p, ('q, 'r) con) implies

  (* p \/ (q \/ r)
  * --------------
```

```
  * (p \/ q) \/ r *)
  val association1 : ('p, ('q, 'r) dis) dis
                     ->
                     (('p, 'q) dis, 'r) dis

  (* p /\ (q /\ r)
  * --------------
  * (p /\ q) /\ r *)
  val association2 : ('p, ('q, 'r) con) con
                     ->
                     (('p, 'q) con, 'r) con
end
```

After clearly defining what is expected of an implementation of the signature, it can be implemented as a structure. The structure is named and it is defined to implement the `PROP` signature. It begins with the keyword `struct` and ends with the keyword `end`.

```
structure Prop : PROP =
struct
  type ('p, 'q) con = 'p * 'q

  datatype ('p, 'q) Either = Left of 'p | Right of 'q
  type ('p, 'q) dis = ('p, 'q) Either

  datatype ('p, 'q) Implies = Implies of ('p -> 'q)
  type ('p, 'q) implies = ('p, 'q) Implies
```

The abstract datatypes are implemented as concrete types within the structure. The types `Either`, `Implies` and `'p * 'q` are not visible outside the signature. As such, e.g their constructors cannot be pattern matched on outside the signature, forcing clients to rely on the abstraction rather than the implementation.

```
  (* Introduction rules *)
  fun con_intro pq       = pq
  fun dis_intro1 p       = Left p
  fun dis_intro2 q       = Right q
  fun implies_intro ptoq = Implies ptoq

  (* Elimination rules *)
  fun con_elim1 (p,_) = p
  fun con_elim2 (_,q) = q
  fun dis_elim (Implies ptor, _) (Left  p) = ptor p
    | dis_elim (_, Implies qtor) (Right q) = qtor q

  fun composition (Implies ptoq, Implies ptor) =
      Implies (fn p => (ptoq p, ptor p))
```

```
  fun association1 (Left p)          = Left (Left p)
    | association1 (Right (Left q))  = Left (Right q)
    | association1 (Right (Right r)) = Right r

  fun association2 (p,(q,r)) = ((p,q),r)
```

The introduction rules, elimination rules and the laws are implemented as normal Standard ML functions. Not only have we defined a signature and implemented a structure - we have also done some simple proofs.
A simple example that illustrates the modularity the module system gives us is that of sorting numbers. We define two signatures.

```
signature ORD =
sig
  type typ
  val leq : typ * typ -> bool
end

signature SORTER =
sig
  type typ
  val sort : typ list -> typ list
end
```

The first signature exposes a type and a function that compares two elements of that type, determining which is smaller or greater. The second signature also defines a type and exposes a function that is intended to sort a list. Using Standard ML *functors*, we can define a structure that implements the `SORT` signature regardless of how elements are compared.

```
functor SortFunctor (O : ORD) : SORTER =
struct
  type typ = O.typ

  fun sort xs = (* ... some sorting function ... *)
end
```

Within `SortFunctor` we can access the types, values and functions defined in the `ORD` interface. The implementation of `sort` makes use of `O.leq` to determine in which order elements should be sorted. The code below implements two structures that define different ways to order integers.

```
structure AscInt : ORD = struct
  type typ = int
  fun leq (x,y) = x <= y
end
```

```
structure DescInt : ORD = struct
  type typ = int
  fun leq (x,y) = not (x <= y)
end
```

The code that sorts elements, defined in the functor, can now be reused to sort integers both in ascending order and descending order. In fact, `SortFunctor` can be used to sort lists of any type as long as it is given an implementation of `ORD` that tells it how to order elements of that type.

```
structure AscendingIntSorter  = SortFunctor (AscInt)
structure DescendingIntSorter = SortFunctor (DescInt)
```

2.2 Region-based Memory Management

The seminal work by Tofte and Talpin[20] describes a system that uses inference rules to transform source code to a target language they call *TExp*. The source language in the seminal paper is the polymorphically typed lambda calculus, but in this section, examples are presented using Standard ML code. The implementation in MLKit transforms Standard ML expressions into one of two different forms of region-annotated expressions. The two forms of annotations are illustrated below.

```
e => e at p
e => letregion p in e end
```

The first type transforms a source expression `e` to `e at p`, which means that when `e` is evaluated the result should be stored in the region bound to the region variable `p`. The second expression describes allocation and deallocation of regions. At run time `letregion p in e end` will first allocate a new region and bind it to the region variable `p`. After this, the expression `e` is evaluated and is able to use the freshly allocated region. Once `e` has been fully evaluated, the region bound to `p` is deallocated.
Expressions of the form `letregion p in e end` are the only construct that will create and destroy regions. As `e` after region inference might contain additional `letregion` constructs, the region allocation points are lexically scoped. When a program is evaluated this will create a stack of regions. They are allocated and deallocated in a stack-like manner.

```
letregion p2 in
  let val xs = "hello" at p2
  in letregion p3 in
       let val len = size xs at p3
       in (len + 5) at p1 end
     end
  end
end
```

Before the above expression is evaluated there is a region `p1` already allocated. When the expression above is evaluated the first thing that will happen is that another region `p2` will be allocated. The inner expression will allocate a string `xs` in the freshly allocated region, and then begin evaluating another nested expression. Again, a new region `p3` is allocated and a local value `len` - the size of the string `xs` - is allocated in `p3`. The result of the expression is the length of the string plus `5`, which is put in the oldest region `p1`. Next, `p3` is deallocated followed closely by the deallocation of `p2`. The nice thing about region annotations like this is that as soon as the result `len + 5` has been computed, `len` is no longer needed. Since it is stored in the innermost region it will be deallocated along with that region immediately. Another example to drive the point home is given in the seminal paper[18].

```
(let x = (2,3)
 in (fn y => (#1 x, y))
 )(5)
```

The example applies a function to the constant `5`. The function will return a pair where the first element is the first projection of `x`, and the second element is the value the function was applied to. The second component of `x` is not needed after the closure of the function has been computed. A region annotated program that reflects this property is shown below

```
letregion p4,p5
in letregion p6
   in let x = (2 at p2, 3 at p6) at p4
      in (fn y => (#1 x, y) at p1) at p5
      end
   end
   5 at p3
end
```

The innermost region, which will be deallocated first, contains only one value, the second component of the tuple `x`. When the closure of the function has been computed the innermost region is deallocated. The tuple as a whole is allocated in region `p4`, while the components themselves reside in potentially other regions. This ability to deallocate parts of a data structure while retaining the relevant parts is very convenient. The expression above that creates the function also creates a tuple, but the parts of that tuple which are not required are safely deallocated before the function is applied.
Consider the return value of a function. We have covered that the only way to create and deallocate regions is with the `letregion` construct. Where does a function allocate its result? Any locally allocated regions using `letregion` would be deallocated when the function returns. If the result is put in a global region, all applications of the function must allocate their result in the same region. This region would have to remain alive until it can be safely determined that the function will not be applied again, which can be difficult to do. To mitigate this problem, functions are region polymorphic and accept regions as parameters at run time.

```
fun add [r0] a b = (a + b) at r0
```

The above function will add two numbers and put the result in the region `r0`, which is given as a parameter to the function. What is evident in the code is that the result is put in the region bound to `r0`, but which region this is will not be known until call time.

```
fun example [r0] () = letregion p0,p1 in
                          let val x = add [p0] (5 at p0) (5 at p0)
                              val y = add [p1] (2 at p1) (2 at p1)
                          in (x + y) at r0 end
                      end
```

The code above makes two calls to `add`, but the region in which `add` put its result is different in each call.
An important point to understand is that some functions will produce results in new regions while some will place their result in the same regions as their arguments. An excellent example of this is that of list concatenation. The physical representation of lists in MLKit is that the empty list is a single word, while any other list is a pointer to a tuple of two words. The first component of this tuple is the head of the list while the second component points to the rest of the list, which in turn is either the empty list or another tuple.
All elements of a list must be placed in the same region, and all the tuples - also called auxiliary pairs - must be placed in the same region. The region containing the elements and the region containing the auxiliary pairs do not necessarily have to be the same region. If we consider how the append function is defined, we can see that the produced list will be placed in the same region as the second operand.

```
fun nil     @ ys = ys
  | (x::xs) @ ys = x :: (xs @ ys)
```

As in the base case the result is `ys`, which is located in some region already, the new elements must be placed in the same region. Functions like this one are said to be *Region Endomorphic* functions.

2.2.1 Runtime representation

In the MLKit, a distinction is made between finite and infinite regions. A region is finite if the compiler can statically determine an upper bound on how big the region needs to be. Consider the trivial example below.

```
letregion p in (3 : int) at p end
```

It is clear that the region `p` will only ever contain one value, an `int`. The example below, however, illustrates the opposite.

```
fun lengthdouble [r0] (xs : string) =
letregion p in
  let xxs = xs@xs at p
  in (length xxs) at r0 end
end
```

The local value `xss` is a `string` - the string `xs` concatenated to itself - in the local region `p`. The size of `xs`, and subsequently `xxs`, is not known during region inference. Finite regions are stored in an activation record directly on the stack, whereas infinite regions are slightly more involved. Infinite regions are represented by a tuple of three elements, `(e,fp,a)`, on the stack. `fp` points to the first page in a linked list of region pages, `a` is the *allocation pointer* and `e` is the *end pointer*. The allocation pointer points to the first free word in the region, while the end pointer points to the end of the last region page. When an object `o` is placed in a region, if `a + size of o > e` is true the region is not big enough to hold the object. In this case, a new region page is allocated and appended to the list of region pages identified by `fp`, and the end pointer is updated to point to the end of the region page. At this point, another attempt is made to put the object in the region, and if successful, the allocation pointer is updated to be `a = a + size of o`. The representation of infinite regions is illustrated in the figure below[1].

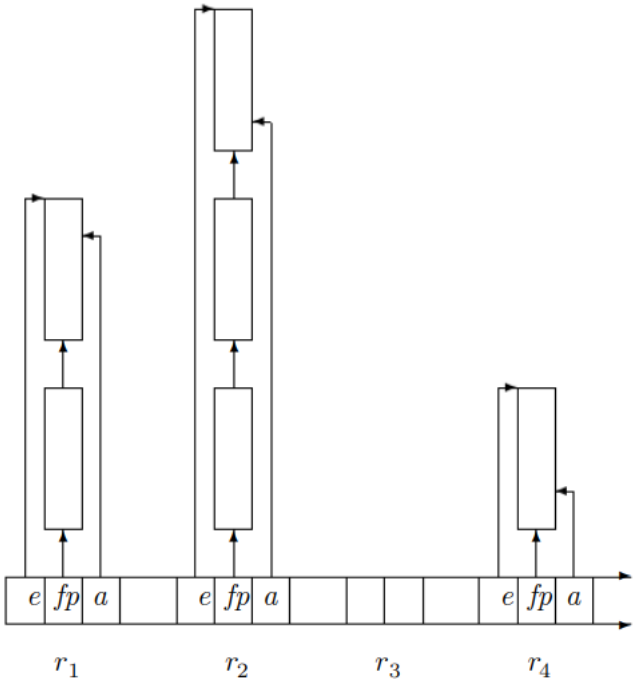

Figure 2.1: Regions r1, r2 and r4 are infinite regions. r3 is placed directly on the stack as it is finite.

Deallocating a region is quite straightforward. If it is a finite region the entire region is located on the stack, and deallocation occurs by moving the stack pointer. If the

[1]This picture has been borrowed from *A Retrospective on Region-Based Memory Management*. Tofte, M., Birkedal, L., Elsman, M. et al. Higher-Order and Symbolic Computation (2004) 17: 245. https://doi.org/10.1023/B:LISP.0000029446.78563.a4

region is infinite, the region pages are appended to a global list of free region pages. When a new region page is requested it is fetched from this list. After this, the triple `(e,fp,a)` is deallocated from the stack by moving the stack pointer.
Occasionally regions can be recycled at runtime rather than deallocated. An analysis performed by the compiler, storage mode analysis, can further annotate each `at ...` annotation to be either `attop` or `atbot`. `attop` is interpreted as described above, while `atbot` tells the compiler that the value can be stored at the beginning of the region, effectively resetting the region. A decision such as this one cannot always be inferred at compile time, hence there exists a third annotation `sat`, *somewhere at*. This annotation says that the decision to store at the top or bottom of a region should be deferred to runtime. At runtime, the two least significant bits of the name of a region hold information of the storage mode. A concrete storage mode is given to all regions at runtime, meaning that a region will never have `sat` as the storage mode at runtime. When evaluating a `sat p`, the storage mode will be fetched from `p` and evaluated accordingly.
The decision cannot always be made as the MLKit supports separate compilation. It cannot be known at compile time what all call sites to a region polymorphic function look like. In this case, `sat` is the chosen storage mode.

2.3 Actor-model Concurrency

If a system is developed in a monolithic way, it quickly becomes hard to maintain and manage the system. It is always good practice to separate areas of concern and write smaller units that are specialised to do just one task. The smaller pieces are then composed and glued together to implement the desired behaviour. Maintaining a system written this way is easier; the component responsible for a bug, if localised, is small and specialised and would hopefully be quicker to debug.
The actor concurrency model agrees with this idea of creating smaller tasks that together perform some computation. In this model, a concurrent computation is described by distributing work among actors and giving them the capability of communicating with each other, which they do by sending and receiving messages. When a message is sent there is no need for a handshake with the recipient. If the mailbox of the recipient exists the message is placed inside it regardless of what the recipient is doing. The sender will make sure that the message is put in a designated area of the recipient's memory, such that when the recipient is ready to receive the message it is already at hand.
Consider the pseudocode below.

```
process1():
  receive
    (add, A, B, Caller) -> send (A+B) Caller
    (mul, A, B, Caller) -> send (A*B) Caller
  end
  process1()
end
```

The actor receives a message, produces some result and then sends it to the actor identified by the `Caller` variable, followed by looping on itself. Note that the actor can choose to receive either a request to add two numbers or to multiply two numbers. A request to divide two numbers, for instance, would not be received by the actor. Such a message would be left in the mailbox, as it is not known at runtime whether the message will be received at a later time or not. It cannot be safely discarded.
In the case that no other actor sends a message to `process1`, `process1` sleeps until there is a message to receive, preserving system resources. Below is an actor that sends a message to `process1`.

```
process2():
  send (add, 6, 2, process2) process1
  receive Answer -> print Answer end
end
```

In the case that `process1` is asleep - waiting for a message - the `send` done by `process2` delivers the message and then wakes up `process1`. After this `process2` sleeps until `process1` delivers the result and wakes it up. When the result has been received and printed, `process2` is finished and will die while `process1` will go to sleep, waiting for the next message.
Actors themselves are independent units of computation, with their own memory and resources. If one actor crashes, the remaining actors stay alive. As they have their own memory, they do not need to bother with protecting shared data using locks and other synchronisation primitives. The only way they may affect each other is by sending and receiving messages. In the example above, `process2` would affect `process1` by sending a message, as it wakes up `process1` when it delivers the message.

2.4 Continuations

Considering an implementation of the scenario described in section 2.3, we quickly realise that we will need some way to model processes such that they can be stopped and resumed at will. When we compute a value we intend to do something with the result, and that is what we call the continuation of a process. Andrew W. Appel has written a book about how to compile with continuations[1]. The book uses ML as a tool to teach the reader how to convert source code into continuation-passing style (CPS). Consider the example below, which is borrowed from the book mentioned above.

```
fun prodprimes n =
  if n = 1
  then 1
  else if isprime n
       then n * prodprimes (n-1)
       else prodprimes (n-1)
```

The code above does some evaluations and has a base case where it returns a 1. In the recursive cases, a check is made on the input, after which e.g recursive calls to `prodprimes` are made. Looking closer we see e.g that the result of `isprime n` is passed to an `if-then-else` expression, after which either of the branches is selected. Now, let's inspect a function which is semantically the same, but which uses continuation-passing style instead.

```
fun prodprimes n c =
  if n = 1
  then c 1
  else let fun k b =
           if b = true
           then let fun j p =
                      let val a = n * p
                      in c a end
                    val m = n - 1
                in prodprimes m j
           else let fun h q = c q
                    val i = n - 1
                in prodprimes i h end
       in isprime n k end
```

The above code *never returns*. In a program that is CPS-converted, every function receives an extra argument, the *continuation*. Where the function would normally return a result, it will in a CPS-converted case apply the continuation to the result instead.
When you would wish to halt the execution of a program in order to let another program have some processor time, you would simply grab the *current continuation* and store it. When it is time to run the process again, the continuation is applied and evaluation is resumed.

3
Library

This chapter will begin by describing the interface of the implemented library, followed by an explanation of the implementation. Finally some examples of programs written using the library are presented.

3.1 Interface

The library is influenced by Erlang, exposing much of the same core functionality. The complete interface is described as a Standard ML signature.

```
signature ActorConc =
sig
  type pid
  type ('a, 'b) either
  type trampoline
```

Three abstract types are defined by the signature. A type of process identifiers, a normal sum type and lastly the type `trampoline`. The last type will be explained in more detail in section 3.2.1. Suffice it to say at this point that this is the type of actors, hereafter referred to as processes.

```
  val spawn : (unit -> trampoline) -> pid
  val embed : (unit -> unit) -> trampoline
```

The function `spawn` applied to a function creates a new process and instructs it to execute that function. After creating the new process, `spawn` returns the freshly minted process identifier.
Only by invoking functions defined by the concurrency interface can a value of type `trampoline` be created. These functions run some computation which has the ability to perform side effects. The function `embed` takes such a computation and embeds it as a process that can be spawned. When this process is allowed to execute, it will run atomically; there will be no context switches interrupting it. This is a design choice that had to be made for *this* implementation of actor model concurrency. In Erlang, for example, the situation is the opposite. Our library cannot grab the continuation of a process, which limits our ability to suspend it. If the programmer is not careful when using `embed`, there is a risk that the atomic computation will starve the system if it never terminates or takes a substantial amount of time to finish.

```
val register   : string -> pid -> unit
val unregister : string        -> unit
val whereis    : string        -> pid option
```

As in Erlang, a process can be associated with a name. This enables a process which does not hold the `pid` of another process to still send messages to that process, given that the name of the process is known. Apart from it also being possible to unregister a process, the process identifier of a named process can be retrieved.

```
exception NameAlreadyRegistered
exception NameNotRegistered
```

Associated with registering processes, two exceptions are defined. The exception `NameAlreadyRegistered` is raised if a call to `register` is made where the first argument is already associated with a `pid`. `NameNotRegistered` is raised if a message is sent to a name that is not associated with a process identifier. In this case, it is impossible to know who the recipient is.

```
val recv      : (string -> 'b) list * ('b -> trampoline)
                ->
                trampoline
val recv_many : int -> (string -> 'b) list * ('b list -> trampoline)
               ->
                trampoline
```

A process can receive messages by using either `recv` or `recv_many`. The function `recv` accepts a single argument which is a tuple. The first component is a list of functions that can deserialise a message and produce some result `'b`. The second component is the continuation, which is applied to the received and deserialised message. A convenience function `recv_many` is exposed, which simplifies receiving many messages before continuing. If the first argument to `recv_many` is a positive integer, that integer specifies how many messages should be received. Without this convenience function, receiving many messages will have to be nested `recv`'s, which produces a lot of parantheses and context switches. `recv_many n` only does a context switch if less than `n` messages are received. `recv_many n` does not call its continuation until `n` messages are received and deserialised.

```
val when : (string -> 'a) -> ('a -> bool) -> (string -> 'b)
           ->
           (string -> 'b)
```

As in Erlang, messages can be selectively received. As will be explained in section 3.2.2, there is more than one way to achieve this behavior. The function `when` accepts a deserialiser, a predicate on deserialised values and the branch that should be guarded by the predicate. The result of calling `when` is a new branch that only receives messages that fulfil the predicate. Messages that don't fulfill the predicate raise the exception `UnpackException`, which is caught and handled internally by the library. A process will not recognise that a message failed to fulfill the predicate, it will continue to wait for a message.

```
val conv : ('a -> 'b) -> 'a pu -> (string -> 'b)
```

To receive messages of any type, messages are serialised. To receive such messages, the branches guarded by a receive are applied to strings. Assuming the message is not serialised, the branch can be of type `'a -> 'b`. Knowing how to deserialise values of type `'a` - which is proved by applying `conv` to a value of type `'a pu` - the branch for deserialised values can be converted to instead be a branch for seralised values. Serialising values, deserialising values and `'a pu` are explained in more detail in section 3.2.2.

```
val picklepid  : pid -> string
val upicklepid : string -> pid
```

Two functions are exposed to serialise and deserialise process identifiers. After serialising a process identifier it can be sent in a message to another process.

```
val pid  : pid    -> (pid, string) either
val name : string -> (pid, string) either
```

To send a message the recipient must be identified. However, the process identifier of a process is not always known. The only way to deliver a message to a process whose process identifier is unknown is if the process has been registered with a name. As there are two ways to identify processes, the two functions `pid` and `name` will indicate if the process is identified by a process identifier or by a name.

```
val send      : (string * (pid, string) either) *
                (unit -> trampoline)
                ->
                trampoline
val send_many : (string * (pid, string) either) list *
                (unit -> trampoline)
                ->
                trampoline
```

To send a message to a recipient, `send` is applied to a tuple where the first component of that tuple is the serialised message and the recipient. The second component of the tuple is the continuation of the process sending the message.

```
val self : unit -> pid
```

The function `self` returns the process identifier of the process that is currently running. It is useful e.g when the process identifier of a parent process should be transmitted to child processes.

```
val endP : trampoline
```

Calling `endP` results in the process evaluating it dying.

```
  val run : unit -> unit
  exception NoRunnableThreads
end
```

A function `run` continuously chooses a processe to run, runs it until a context switch occurs, and then continues with running another process. The exception `NoRunnableThreads` is raised if there are processes left that have not died, but none of which are runnable. This completes our description of the library API.

3.2 Implementation

3.2.1 Trampolines

Trampolines offer a way of executing a program in a discrete number of steps. Initially their use was targeted towards compilers that wanted to do multithreaded computations but did not want to implement continuation-passing style conversion[10]. In this project we have used trampolines towards the same purpose, to achieve some level of multithreading. The discussion that follows tries to educate the reader about trampolines by illustrating how trampolines can improve situations in a language that does not do tail-call optimisation.
To give some intuition for what trampolines are, we consider the factorial function.

```
fun fac 0 = 1
  | fac n = n * fac (n-1)
```

Inspecting the call stack when applying `fac` to `5`, it would look something like below.. Note that the word *ret* represents a stack frame that the result needs to be returned from.

```
fac 5 = ret 5 *
           ret (4 *
               ret (3 *
                   ret (2 *
                       ret (1 * 1)
                       )
                   )
               )
```

The code builds up a call stack where there is work left to do after each call returns, namely multiplying the result of `fac (n-1)` by `n`. Let's rewrite the code in such a way that the result is passed into the calling function, making it tail recursive.

```
fun factr 0 a = a
  | factr n a = factr (n-1) (a * n)
```

In the base case the accumulated result is returned, and in the recursive case there is no work left once the recursive call to `factr` returns. Inspecting the callstack when running this code would show us something that looks like this.

```
factr 5 = ret
             ret (
                 ret (
                     ret (
                        ret (120)
                         )
                     )
                 )
```

There is no work left to do in every stack frame. However, the result still has to be passed up through all stack frames, which is ineffective. Some compilers will optimise this behaviour away by doing *tail-call optimisation*. What this optimisation does is that it recognises when a recursive call is a tail-call. In such a case the compiler generates assembly code that *jumps* to the function rather than *calls* the function. Jumps don't require a stack frame, but function calls do as a stack frame is required to hold e.g the parameters, the return address and frame pointer.
Trampolines are a way of controlling how the stack grows. Evaluating a trampoline will result in either a value or another trampoline. Consider the Standard ML representation.

```
datatype ('a, 'b) Either = Left of 'a | Right of 'b
datatype 'a Trampoline = T of (unit -> ('a, 'a Trampoline) Either)

fun eval (T t) = case t () of
             (Left v)  => v
           | (Right f) => eval f
```

Evaluating a trampoline is done by applying `()` and observing if the result was a value or another trampoline. In the case where the result is another trampoline, `eval` is recursively applied to the new trampoline. Rewriting `factr` to make use of trampolines would look like this.

```
fun facT 0 a = T (fn () => Left a)
  | facT n a = T (fn () => Right (facT (n-1) (a * n)))
```

The base case returns a trampoline that returns the accumulator, `Left a`. The recursive case, however, returns another trampoline.

```
eval (facT 5 1)
> 120
```

When `eval (facT 5 1)` is evaluated there will be one stackframe for `eval` and one for the call to `facT`. Where before the call stack looked like this.

```
factr 5
  factr 4
    factr 3
      factr 2
        factr 1
          factr 0
```

It would now look like this.

```
eval
  facT 5
  facT 4
  facT 3
  facT 2
  facT 1
  facT 0
```

There is one stack frame for `eval` and one for `facT`. Since `facT` does not make the recursive call, however, but rather return it to `eval`, the stack depth for calls to `facT` never exceeds one. In *this particular case*, however, a stack frame is still needed as `eval` is still implemented using recursion. In a realistic example eval would be implemented as a loop, which doesn't use any stack space.

3.2.2 Serialising messages

Unlike in Erlang, where messages of any type can be sent to anyone, in Standard ML we need to take more care when crafting our messages. Every Standard ML expression will have a type inferred for it at compile-time, which raises the question of what type the mailbox should have. It might be tempting to just make it polymorphic in its contents, but that is not a suitable solution. At compile time the type checker would try to unify the polymorphic type and instantiate it with a concrete type. The mailbox would indeed be able to contain integers, booleans or any other type of value. This type would however have to be decided at compile-time, after which the mailbox can only hold values of that specific type. What we desire is a mailbox that can contain messages of different types at the same time.
In the implementation of the library we have achieved this by serialising and deserialising messages, converting them to strings. The type of the contents of a mailbox thus simply becomes `string`. The MLKit comes with a serialiser, one which does not do any checks regarding types. A message can be deserialised to any type, regardless of which type the original value had. While deserialising a string to a `b` if it was originally an `a` does not always succeed, sometimes it does and you would get an unexpected `b`. As an example; if an integer is serialised, the resulting string should only be able to be deserialised back into an integer, not e.g a boolean.
We modified the code to not blindly try to deserialise a value, but rather to inspect if the attempted deserialisation is safe. Upon serialisation the serialised value is prefixed with a string representation of its original type. When an attempt is made to deserialise that string it is first checked that the type it is deserialised to matches that of the prefix found on the string.
The core functionality of the serialiser is expressed by the following definitions.

```
exception UnpackError
type 'a pu
val pickle   : 'a pu -> 'a -> string
val unpickle : 'a pu -> string -> 'a
```

```
val trep     : 'a pu -> string
```

The type `'a pu` can be thought of as a description of how to serialise and deserialise values of type `'a`. Given a `p : 'a pu`, `pickle p` is a serialiser for values of type `'a`, and `unpickle p` is a deserialiser for values of type `'a`. `trep` returns the string the `'a pu` prefixes serialised messages with. Let's consider an example.

```
val intpu       = (* int pu *)
val serialise   = pickle   intpu
val deserialise = unpickle intpu
val msg         = serialise 5
val val         = deserialise msg

val eq          = (val = 5)
val _ = print ((Bool.toString eq) ^ "\n")
> "true"
```

Given a `intpu : int pu`, we can construct a serialiser and deserialiser for values of type `int` by applying `pickle` and `unpickle` to it, as shown above. `eq` will evaluate to `true` if the deserialised value is the same as the initial value, and the call to `print` verifies this.
As mentioned above, values of type `'a pu` can be thought of as a description of how to serialise and deserialise values of type `'a`. If a value is serialised using a specific `'a pu`, the serialised value can only be deserialised using the same `'a pu`. If this is not the case the exception `UnpackError` is raised.
There can be many different `'a pu` for a `'a`, and an interesting side effect of this is that the same value could potentially by serialised by many different `'a pu`'s. Even if there are two perfectly valid but slightly different `'a pu`'s, a message serialised using one of them cannot be deserialised using the other. This lets programmers be very fine grained about how values are serialised and deserialised. It could be argued that not being able to deserialise a value despite having a perfectly capable `a pu` is a bug, but in this project we have used this quirk to our advantage and consider it a feature. Instead of using the function `when` to selectively receive values, different `a pu`'s could be used to achieve a similar effect.
The module comes equipped with serialisers for the base types of Standard ML, and functions to facilitate the creation of serialisers for more involved types such as tuples and lists. Creating a serialiser for pairs of integers and booleans is for example quite simple.

```
val ints   : int  pu         = (* int  pu *)
val bools  : bool pu         = (* bool pu *)
val tuples : (int * bool) pu = pairGen(ints, bools)
```

Creating a `pu` for a custom datatype is a little more involved, but still straightforward. Let's consider the sum type.

```
datatype ('a, 'b) Either = Left of 'a | Right of 'b
```

To create a value of type `('a, 'b) Either` either the constructor `Left` is applied to a value of type `'a`, or the constructor `Right` is applied to a value of type `'b`. To create a `('a, 'b) Either pu` we can apply the function `val dataGen : string * ('a->int) * ('a pu` The first argument is a function that maps the different constructors of the datatype to unique integers in ascending order, starting from zero. In this case there are two constructors, `Left` and `Right`. They are mapped to `0` and `1`, respectively.

```
fun index (Left _)  = 0
  | index (Right _) = 1
```

The second argument for `dataGen` is a list of functions. The functions will create a `('a, 'b) Either pu` each, one for each constructor of the datatype. The idea is that the indexing function, when applied to a value, will return the index in the list where the right `('a, 'b) Either pu` can be found.
To construct an `('a, 'b) Either pu` we use the function below.

```
val con1 : ('a->'b) -> ('b->'a) -> 'a pu
           ->
           'b pu
```

If we have a serialiser for `'a`'s, and a way of converting `'a`'s to and from `'b`'s, we can create a serialiser for `'b`'s.

```
fun eitherPickler pa pb =
let
  fun index (Left _)  = 0
    | index (Right _) = 1
  fun leftP  pu = con1 Left  (fn Left i  => i) pa
  fun rightP pu = con1 Right (fn Right i => i) pb
in dataGen("Either" ^ (trep pu) ^ (trep pb), index, [leftP, rightP])
end
```

Applying `dataGen` to get a `('a, 'b) Either pu` is now possible. The first argument is a string that represents the type being serialised. It is important that this string is uniquely associated with this type, as this is what will hinder the value from being deserialised using another `('a, 'b) Either pu`. If the string was just `"Either"`, a `(int, bool) pu` could be used where a `(bool, int) pu` is expected. To solve this we append the string representations of the two recursive pu's to `"Either"`. Then, also given a `'a pu` and a `'b pu`, the result from `eitherPickler` is a `('a, 'b) Either pu`.
This way of creating `'a pu`'s allows the programmer to be very specific when writing serialisers. A `'a pu` does not have to be defined for all possible values of type `'a`. Consider the alternative `('a, 'b) Either pu` below.

```
fun leftPickler pa =
let
  fun index _ = 0
  fun fun_L pu = con1 Left (fn Left i => i | _ => raise UnpackError) pa
```

```
in dataGen("EitherLeft", index, [fun_L])
end
```

The `('a, 'b) Either pu` above only describes how to serialise and deserialise values created with the `Left` constructor. If an attempt is made to serialise a value constructed with the `Right` constructor, the exception `UnpackError` is raised. If a value `Left val` is serialised with `eitherPickler`, it cannot be deserialised with `leftPickler`, as the prefix of the serialised message does not match the one specified in the creation of `leftPickler`. Consider the example below.

```
val intpu  = (* int  pu *)
val boolpu = (* bool pu *)
val msg    = pickle (eitherPickler intpu boolpu) (Left 5)
```

The value has now been successfully serialised, and could be transmitted using the library. However, if the recipient does not have access to an identical serialiser despite having a serialiser for the same type of values, the deserialisation will fail.

```
val res : (int, bool) Either = unpickle (leftPickler intpu) msg
> uncaught exception UnpackError
```

3.2.3 Mailboxes

The requirements we impose upon a mailbox are few and simple, but nonetheless important. Apart from being able to receive messages, we require that messages be received in the order in which they arrive in the mailbox. If we remove a message from the mailbox, it should be possible to put it back in the same spot. For our purposes this functionality is required when for example a message has been retrieved and deserialised, but it was blocked by a selective receive. In this case we wish to put the message back in the mailbox.
The type of (mutable) mailboxes is a reference to a tuple. The first component of this tuple is a queue of messages, while the second component is a list of messages. To clarify why there are two data structures holding messages we consider how messages are sent and received.
When a message is sent to the mailbox, it is placed at the back of the queue. When a message is received, it is taken from the front of the queue. After a message has been received the library is going to try to deserialise it. In the case that this fails, the message should be left in the mailbox and the next one should be retrieved instead.
When a message is put back in the mailbox after being received, it is put at the front of the list of messages. If it was put back in the queue instead, the queue would need to be traversed to find the next message. Doing things like this makes sure that the next message to receive is always at the front of the queue.
After a message has been successfully deserialised, a call to the function `resetsave` will create a new queue of messages. The front of this queue will be the reversed list of already checked messages and the back will be the old queue. The next time a message is received it is truly the oldest message.

3.2.4 Library implementation

The implementation of the library is realised by defining a structure that implements the signature described in section 3.1.
Actors are implemented in this library by using continuations. Actors, hereafter referred to as processes, keep the remainder of their computation represented as a trampoline. The two remaining components associated with a process are a unique identifier and a mailbox. These three components together make up the process control block of a process, and a process identifier is a reference to one of these triples.

```
type PCB = (int * string mailbox * unit Trampoline)
type pid = PCB ref
```

To serialise a reference it must be possible to serialise the referenced value. The referenced value of a process identifier contains a function, which we have no way of serialising. As a consequence of this we cannot directly serialise process identifiers. Despite this `picklepid` and `upicklepid` allows a user to serialise and deserialise a process identifier. To make this possible we maintain a global map that maps the id of a process to the process identifier of that process. What happens when `picklepid` is called is that only the id is serialised, and when a call to `upicklepid` is made the id is deserialised and used to fetch the process identifier from this map. Apart from this map, the library maintains five other global values that can be manipulated by calling the functions exposed by the library.

```
val ready_queue   : (pid set) ref
val waiting_queue : (pid set) ref
val registry      : (string, pid) map
val current       : pid option ref
val last_id       : int ref
```

The first two are queues containing process identifiers. They maintain the state that defines which processes are ready to run and which are waiting for new messages. The `registry` map is used to associate a name with a process identifier. To keep track of which process is currently running, the process identifier of that process is stored in the mutable variable `current`. `last_id` is used to generate fresh identifiers for spawned processes.

```
  fun spawn (f : unit -> trampoline) : pid =
  let
    val id      = next_id ()
    val mailbox = new     ()
    val pid     = ref (id,mailbox,embedUTT f)
  in
    (init_pid pid;
     insert_ready pid;
     pid) (* return process identifier *)
  end
```

When a process is spawned a fresh id is generated, an empty mailbox created and the process body `f` is embedded as a trampoline. The process identifier is put in the ready queue after which it is returned to the caller.

```
fun run () : unit =
  case pop_ready () of
      SOME pid => (set_current pid;
                   case let val (_,_,Trampoline cont) = !pid
                        in cont () end of
                     Left ()  =>  run ()
                   | Right f' => (update_cont pid f';
                                  run ()))
    | NONE     => case size (!waiting_queue) of
                    0 => ()
                  | _ => raise NoRunnableThreads
```

To begin executing the concurrent computation the function `run` must be applied to unit. This function will try to fetch the next process to run from the ready queue, and if there is one, run its continuation. If the result is another computation, the process identifier is updated to point to this new continuation before a recursive call to `run` is made. As `run` is a tail-recursive function, without tail-call optimisation every recursive call is going to create a new stack frame. This would severely impact memory performance.

```
fun recv_many n handler_and_cont : trampoline =
let
  (* receive and attempt to deserialise a
   * message until either one succeeds or
   * there are no more messages *)
  fun fetch_message handlers = ...

  (* reset save pointer in mailbox *)
  fun resetsave () = ...

  fun recv' 0 (handlers, cont) res = (resetsave (); cont (rev res))
    | recv' n (handlers, cont) res =
  case fetch_message handlers of
      SOME handled => (recv' (n-1) (handlers, cont) (handled::res))
    | NONE         => (insert_waiting (get_current ());
                       Trampoline (fn () =>
                       Right (recv' n (handlers, cont) res)))
in recv' n handler_and_cont [] end
```

To receive messages two important auxiliary functions are used.
`fetch_message` will return the first message from the mailbox that could be deserialised using one of the supplied deserialisers. If no message could be deserialised `NONE` will be returned. Otherwise the result is `SOME` message.

`resetsave` is going to fetch the mailbox of the currently running process and reset its save pointer. This function is called as soon as the function has received all `n` messages.
If `n` messages could be received the continuation is applied to a reversed list of received messages. Otherwise the process is placed in the waiting queue and the work that remains, receiving the rest of the messages, is made to be the new trampoline of the process.

```
fun when deserialise predicate branch =
    fn str => if   predicate (deserialise str)
              then branch str
              else raise UnpackException
```

`when` is implemented by returning a function that accepts a string. If the predicate holds on the result of deserialising the string, the branch is applied to the string. If on the other hand the predicate does not hold, an exception is raised.

```
  fun send_many ([], cont) = (insert_ready (get_current ()); embedUTT cont)
    | send_many (((msg, recipient)::xs), cont) =
    if
       (* is the process alive? *)
       let val pid = deref recipient
       in member ((!ready_queue), pid)
          orelse
          member ((!waiting_queue), pid)
       end
    then (let val pid = deref recipient in
            let val (_,mailbox,_) = !pid in
              (deliver (mailbox, msg);
               insert_ready pid)
             end
          end;
          send_many (xs, cont))
    else  send_many (xs, cont)
```

Sending a message is done with either `send` or `send_many`. `send` is implemented by invoking `send_many` with a singleton list of messages. Before the message is transmitted it is checked that the recipient is still alive. If that is the case, the message is sent to the mailbox of the recipient and the recipient is placed in the ready queue. Otherwise the recipient is dead. In this case the message is dropped and `send_many` moves on to the next message.

4
Results

This chapter will begin by describing the different benchmarks used to evaluate the performance of the library, followed by the actual performance measured. The tests were run on a Lenovo Thinkpad 13 with an Intel Core i3-6100U processor (3MB cache, 2.30 GHz).

4.1 Benchmarks

The benchmark programs have been implemented both using the library described in this report and Erlang.[1]Erlang was chosen as the baseline as it has a mature implementation of the actor concurrency model. The benchmarks will be run and both speed and memory usage will be measured.
The *Bitonic Sorting Algorithm* is a sorting algorithm that does exactly $n * log^2(n)$ comparisons, where n is the size of the collection being sorted. The algorithm assumes that the length of the input is a power of 2.
Skynet is a program that aims to measure the cost of spawning actors. Initially, one actor is spawned, which will spawn three children, who in turn will spawn three children each of their own, and so on. When the level of recursion reaches a predefined depth the children at the bottom, the leaves will send a one to their parents, who will receive three messages, sum the results and send that to their own parent. In the end, the actor that was spawned first is going to receive three messages, the sum of which is equal to the number of leaves.
The purpose of the *Message Bombing* benchmark is to evaluate the overhead of carrying around unreceived messages. This benchmark consists of two programs, one in which a process will be sent 200 messages it will not receive, followed by being sent 200 messages it does receive. The second program will not send the initial 200 messages, but rather just the 200 that will be received. Measuring the difference in time and memory usage between the two should give some estimate of the cost of having the messages take up space in the mailbox.

4.2 Speed

When measuring speed the Standard ML benchmarks were compiled by invoking the *mlkit* executable with the $-no\_gc$ flag. When measuring speed the memory profiling is completely disabled.

[1]The code for some of the benchmarks can be found in the Appendix.

The time of the Standard ML benchmarks were measured using the Unix command *time*. The Erlang benchmarks were measured using the *tc*/3 function from the *timer* module. The reported execution times are the avarages of running each benchmark one hundred times.

Benchmark	Standard ML	Erlang
Skynet	177.3 ms	2.7 ms
Bitonic mergesort	353.6 ms	1.6 ms
Message bombing 1	7.5 ms	0.5 ms
Message bombing 2	4.1 ms	0.17 ms

Table 4.1: The table above presents the execution times for the different benchmarks in milliseconds. The entry Message bombing 1 reports the time measured while executing the version of the test that first sends 200 messages that won't be received, while the entry Message bombing 2 reports the time measured while executing the version that only sends messages which will be received.

4.3 Memory

The MLKit compiler comes prepared with a profiler to measure memory usage. The profiler can be instructed to stop and profile every region at specified intervals. The measurements can be rendered as a graph, allowing for visual inspection to detect memory leaks. Below are the profiles for the benchmark programs.

The MLKit has the option of enabling a garbage collector to garbage collect the regions individually. Memory was measured with the garbage collector disabled and then again with it enabled. Seeing the results of enabling the garbage collector gives some indication of how well the library frees up unused memory.

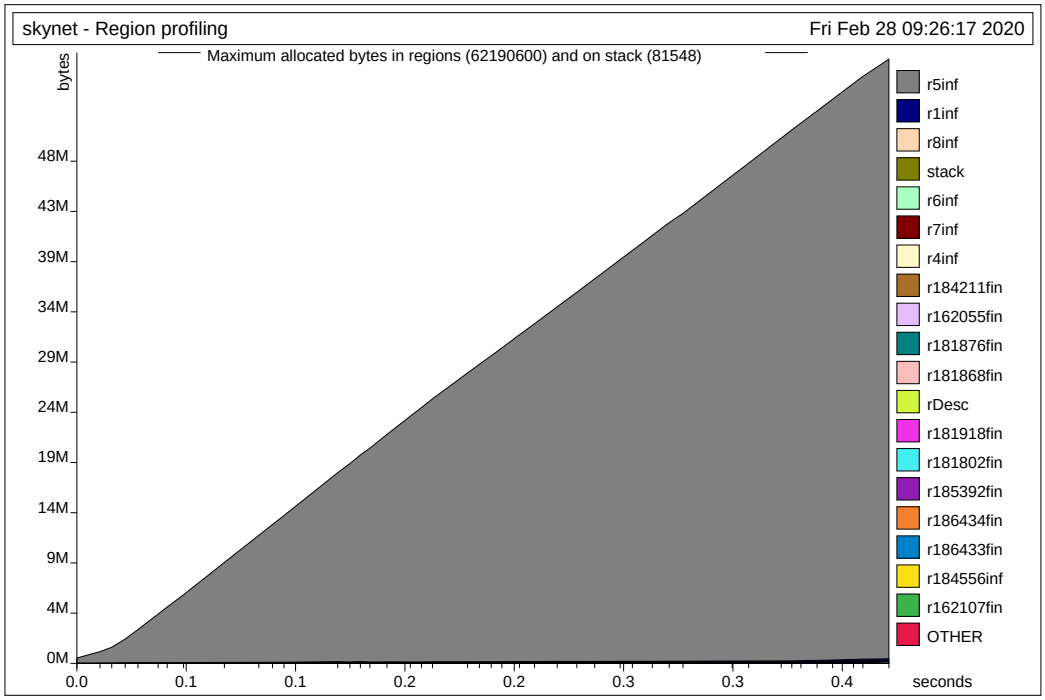

Figure 4.1: Memory performance of running the Skynet test without garbage collection.

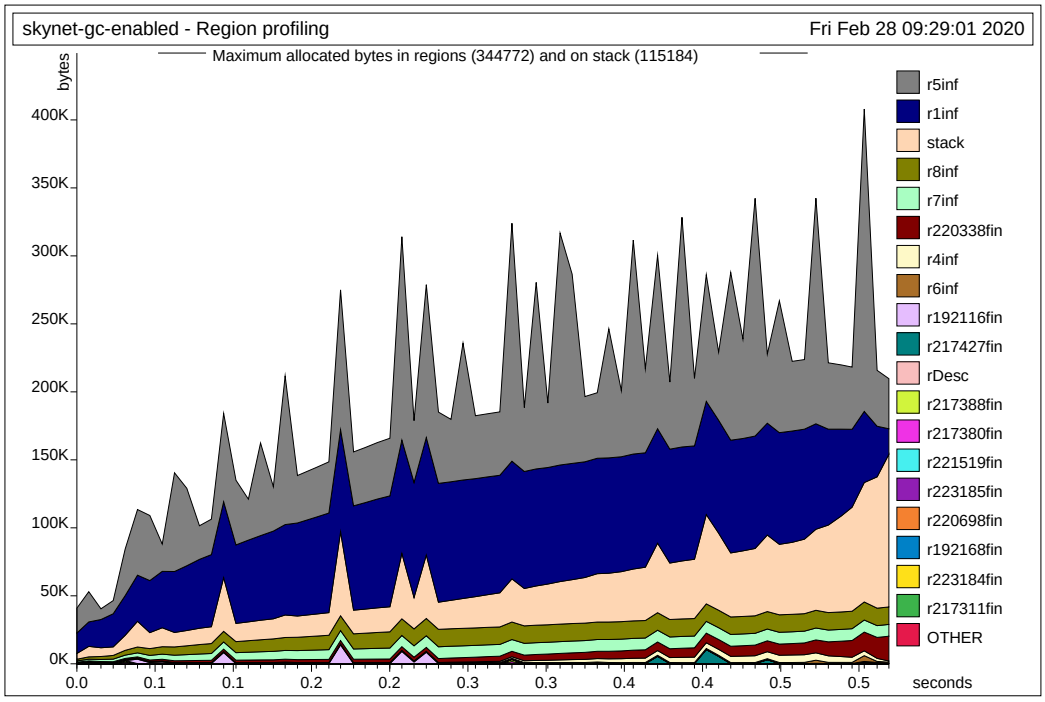

Figure 4.2: Memory performance of running the Skynet test with garbage collection. It is evident that a lot of dead values can be reclaimed by the garbage collector.

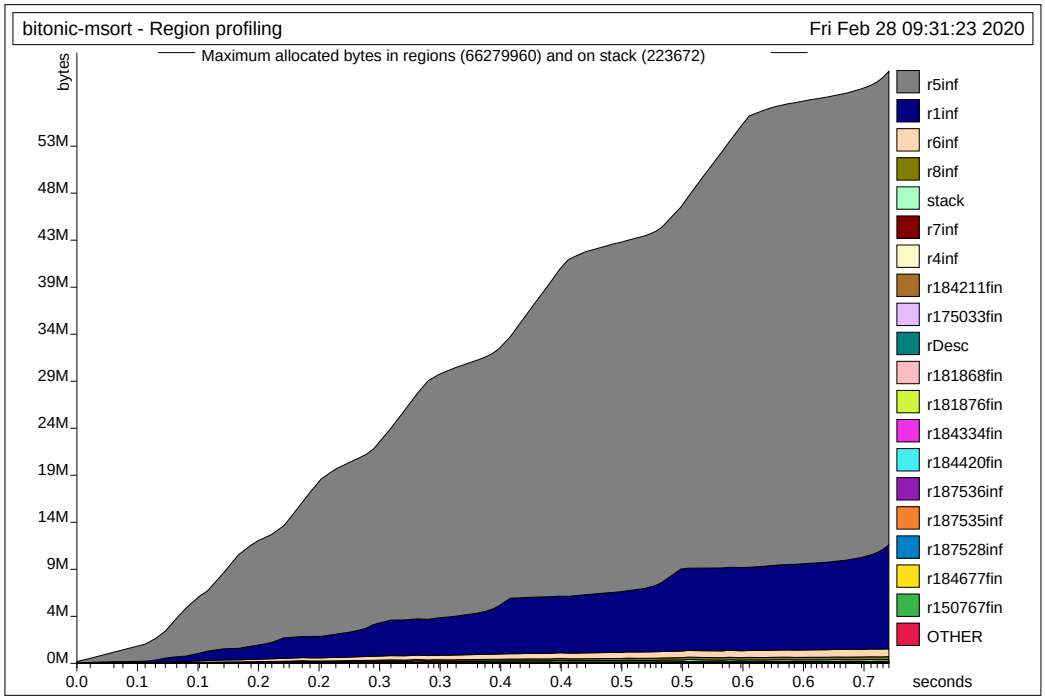

Figure 4.3: Memory performance of running the Bitonic mergesort test without garbage collection.

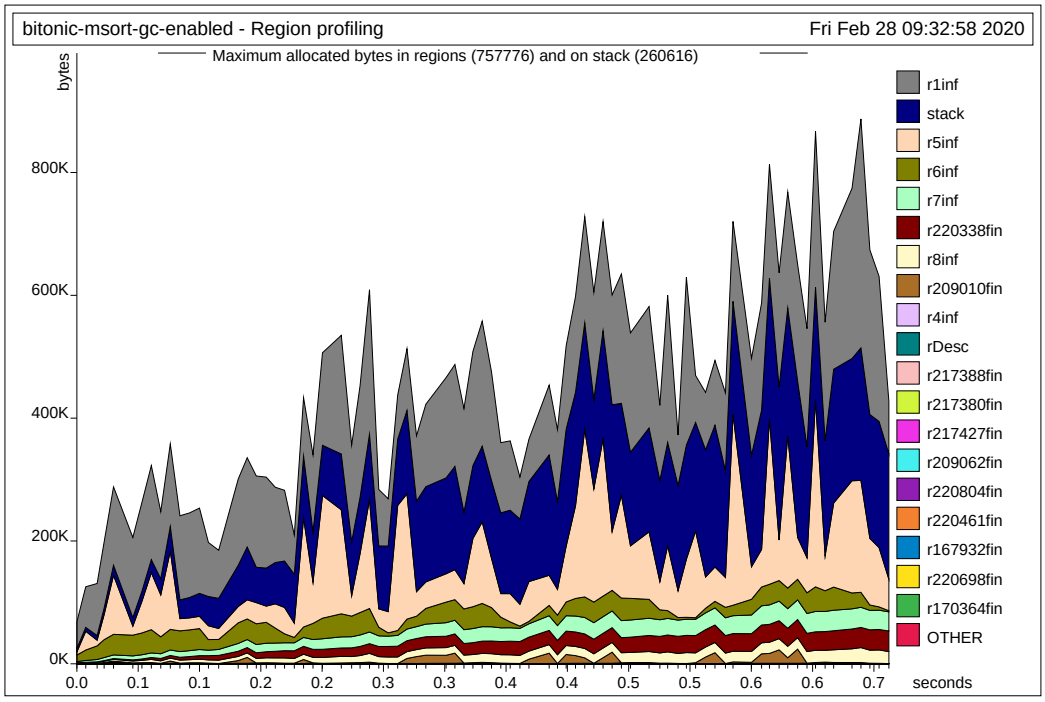

Figure 4.4: Memory performance of running the Bitonic mergesort test with garbage collection. Also here, a lot of dead values can be reclaimed by the garbage collector.

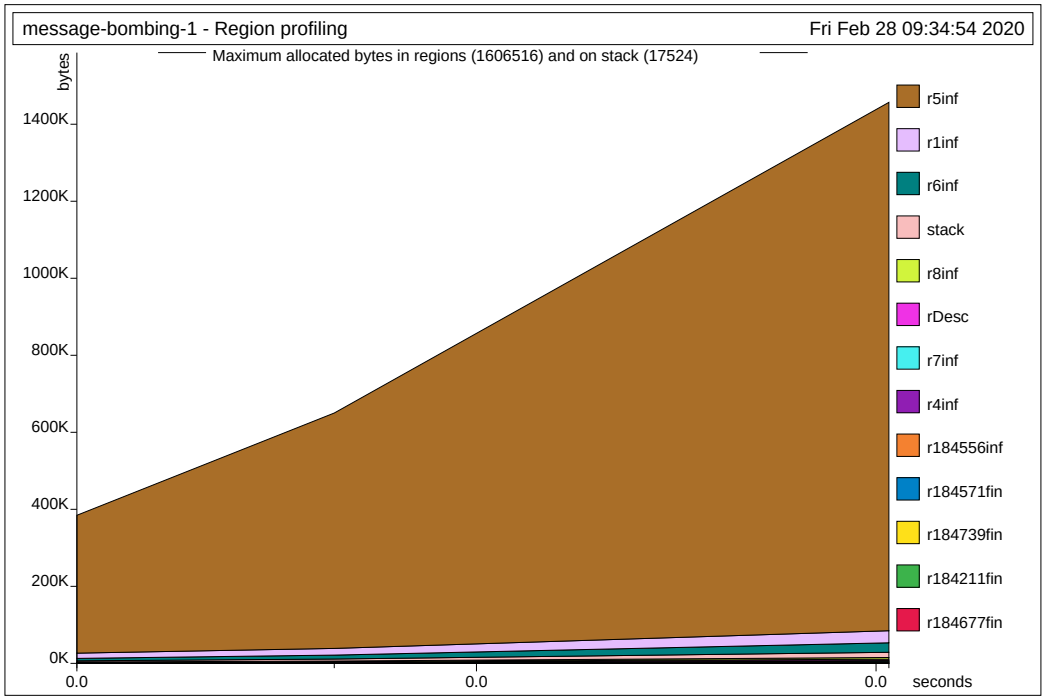

Figure 4.5: Memory performance of running the Message bombing test without garbage collection. This is the version of the test that first sends 200 messages that are not received.

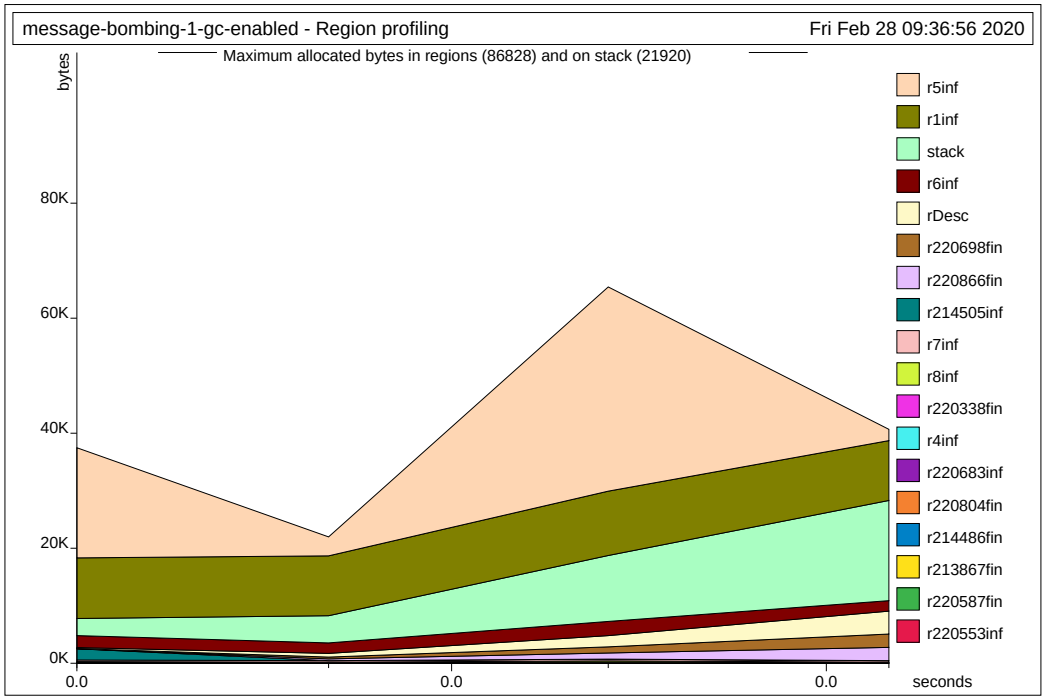

Figure 4.6: Memory performance of running the Message bombing test with garbage collection. This is the version of the test that first sends 200 messages that are not received. The garbage collector can reclaim a lot of dead values in this test. Memory usage is more than 18x lower.

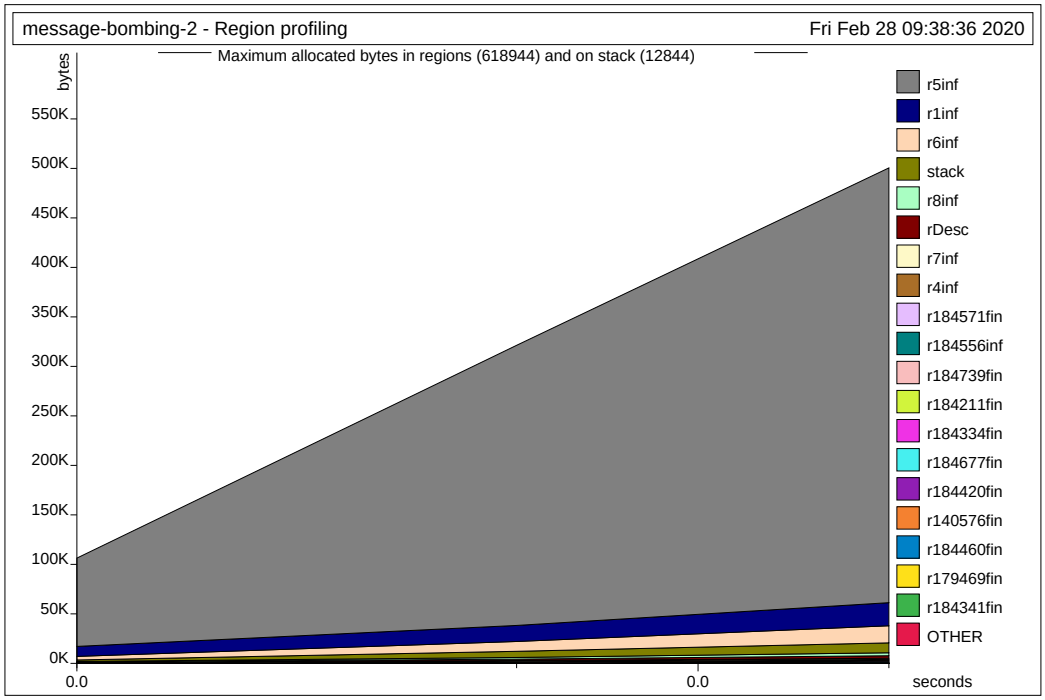

Figure 4.7: Memory performance of the Message bombing test without garbage collection. In this version only received messages are sent. Slightly more than 0.5 MB is needed.

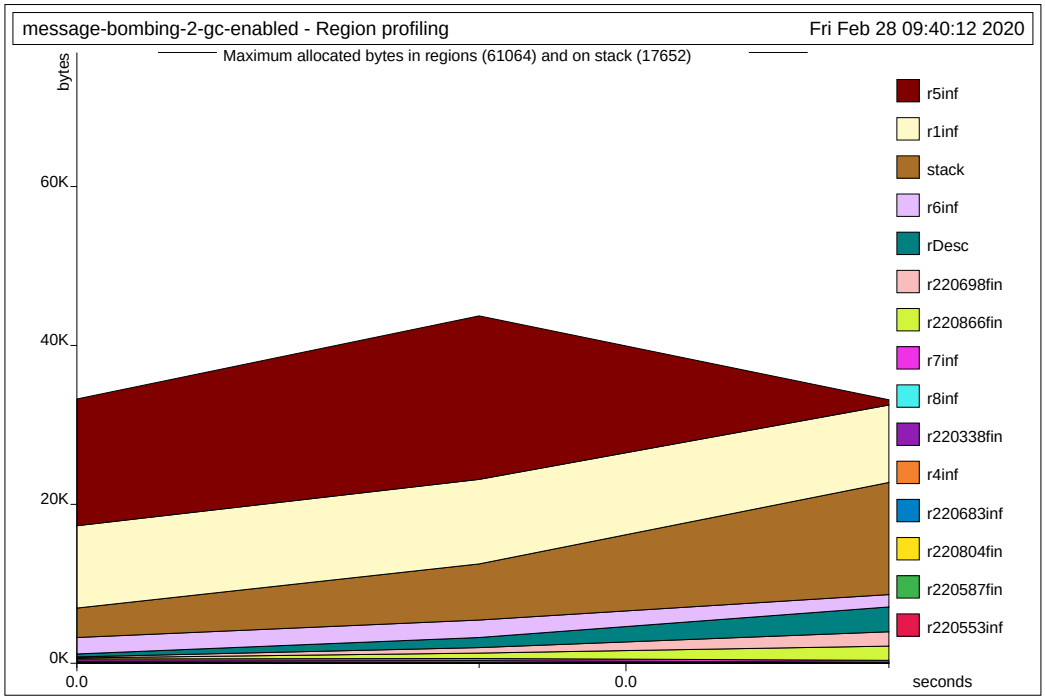

Figure 4.8: Memory performance of the Message bombing test with garbage collection. In this version only received messages are sent. It is interesting to see here that stack usage goes up, while the infinite regions are downsized alot. Peak memory consumption in this case is ten times lower.

Benchmark	Standard ML	Erlang
Skynet no gc	62272148	n/a
Skynet gc	459956	2701946
Bitonic mergesort no gc	66503632	n/a
Bitonic mergesort gc	1018392	1291234
Message bombing 1 no gc	1624040	n/a
Message bombing 1 gc	108748	416075
Message bombing 2 no gc	631788	n/a
Message bombing 2 gc	78716	424078

Table 4.2: The table above summarises the memory performance of the library implementation in Standard ML and that of Erlang. The reported numbers are number of allocated bytes. It is evident that a lot of memory can be reclaimed by a garbage collector in the Standard ML version, for all benchmarks. With the garbage collector enabled memory usage is less than that of Erlang - but the memory reported for the Erlang version also includes the Erlang VM.

5

Analysis

The results indicate that while execution speed is very slow, memory usage is extremely high. A lot of data is placed in early regions and thus survives for a very long time. This chapter will try to shed some light on why this is the case.

5.1 References

The first problem we identified is the use of references. To understand why references pose a problem for the work in this thesis, we need to consider the region annotated type schemes for the operations that can be done on references. Functions in a type scheme are annotated with their effects. An effect is a finite set of atomic effects, which are either of the form `get p` or `put p`. `get p` specifies that a value is read from the region `p`, and the dual `put p` specifies that a value is written to the region `p`.

$$
\begin{array}{ll}
\texttt{ref} & \forall\alpha\rho_1\rho_2\epsilon.(\alpha,\rho_1) \xrightarrow{\epsilon.\{\mathbf{put}(\rho_2)\}} ((\alpha,\rho_1)\texttt{ref},\rho_2) \\
\texttt{!} & \forall\alpha\rho_1\rho_2\epsilon.((\alpha,\rho_1)\texttt{ref},\rho_2) \xrightarrow{\epsilon.\{\mathbf{get}(\rho_2)\}} (\alpha,\rho_1) \\
\texttt{:=} & \forall\alpha\rho_1\rho_2\epsilon.[((\alpha,\rho_1)\texttt{ref},\rho_2),(\alpha,\rho_1)] \xrightarrow{\epsilon.\{\mathbf{get}(\rho_2)\}} \texttt{unit}
\end{array}
$$

An important point is that the reference itself and the value it references does not have to reside in the same region. A reference is a word in a region that contains the address of a value in another region.
When a reference is created, the function `ref` is applied to a value of type α in region p_1, and the result is a reference in a region p_2. Note that in the type scheme, the type of the result from `ref` clearly indicates that the reference, located in region p_2, references a value in another region.
The type scheme for the de-referencing operator is not that involved; instead, consider the assignment operator. The reference being updated references a value of type α in region p_1, and the new value is also a value of type α in region p_1. What this means is that every value that is written to a reference has to reside in the same region.
If the values are unboxed this is not an issue, as the address stored by the reference will be the actual value. If the value is boxed, however, this could lead to severe space leaks. The region p_1 cannot be deallocated as long as a reference pointing to values inside it is alive.
The library implemented in this thesis makes use of references for a couple of things. The global values maintained by the library, such as the process registry, have to be

mutable. As mutable data structures are implemented using references, this means that every version of the data structure has to reside in the same region. Considering the process registry, the registry has to be alive for the complete duration of the program, as it is difficult to identify the point after which it is no longer needed. The result of this is that a lot of data is going to remain alive until the program in its entirety has finished. In fact, every time a mutable data structure is updated the old copy will remain allocated until the program terminates.

5.2 Global values are put in global regions

As we explained above, the way that regions are allocated using `letregion` will at runtime build a stack of regions. Regions that are allocated early in a program might be alive for a very long time. Even functions have to be put in a region, and this region cannot be deallocated until the program as a whole has finished.
The library presented in this thesis uses continuations to implement coroutines. When writing a program the continuations are just ordinary functions. They are passed to the library and possibly wrapped in a trampoline constructor. *All* trampolines, after passing through the region inference algorithm, are translated to `Trampoline attop r2 ...`. In every place where the creation of a trampoline is accompanied by the creation of an anonymous function, which is almost always the case, the trampoline is translated to `Trampoline attop r2 (fn attop r1 ...`.
Pretty much every time that a context switch happens, and a new trampoline is created, there is data placed at the top of the regions `r1` and `r2`. As `r1` and `r2` will be the two oldest regions during execution, that data will be the most long-lived in the program. This is very unfortunate as every time a context switch happens and a trampoline is executed, that trampoline will return a *new* trampoline if the process has work left to do. The trampoline that was just executed is no longer needed once it has returned a new trampoline, and could, in theory, be deallocated, but this fact is not recognised by the region inference algorithm.
As we saw in the previous section, references can be dangerous when it comes to space leaks. Mutable values are implemented using references, to ensure that any variable that references a value will see changes to that value made by someone holding another reference to that value. Two of the global values kept by the library are the ready queue and the waiting queue. These are references to queues, and thus all values ever written to the reference - all versions of the queue - must be stored in the same region. Inspecting the creation point of e.g the ready queue, after region inference has transformed it, gives the following expression.

```
val ready_queue =
   ref attop r7
      (nil attop r2,
       fn attop r1 v154 =>
          let val v155 = #0 v154; val v156 = #1 v154
          in v155 = attop r2 v156
          end
      ) attop r5;
```

The queue becomes a reference in region r7, which points to a tuple - the internal representation of the queue, in this case, is a tuple consisting of a list and a predicate on the contents - in region r5. Here we can draw the conclusion that every time the reference is updated with a new queue, the new tuple will be put in region r5, adding to the buildup of data inside it. To further understand why this behaviour would cause such a huge buildup of dead values, we can inspect the methods that produce new queues - after region annotation.

```
fun insert attop r1 [r714:1, r706:0, r703:0,
                     r702:INF, r705:INF, r700:0]
                    (v320, v618) =
  let val v321 = #0 v320
  in  (case v321 of
         nil => let val v322 = #1 v320
        in (:: attop r2 (v618, nil attop r2) attop r705,
            v322) attop r714
                end

      |  _ =>   let val v334 = #1 v320
                in  letregion r718:1
                    in (case letregion r716:1
                             in find[atbot r718,attop r705,atbot r716,attop r700]
                                 <fn atbot r716 x =>
                                    v334 (v618, x) attop r702, v321>
                             end
            of SOME => (v321, v334) attop r714
                        |  _ =>    (letregion r729:1
                    in @[atbot r729,attop r705,
                                    attop r705,attop r700]
                       (v321,
                        :: attop r2 (v618, nil attop r2) attop r705)
                                     atbot r729
                    end,
                   v334) attop r714
                           )
                    end
                end
      )
  end ;
```

The red sections indicate where the result of a call to `insert` is created, and in which regions it is put. The result will again be a tuple, of which the first component is a new list. We see that the `::` constructor, which contains two words pointing to the head and the tail of a list, is placed in region `r2` using the `attop` storage mode. The list itself is put in the region denoted by the region parameter `r705`. The entire tuple produced by the function is placed in the region denoted by the region parameter `r714`, again using `attop` to indicate the storage mode.
So there is clearly some data being put at the top of `r2`, but which regions are `r705` and `r714` bound to? They are the first and fifth region parameter in `insert`.

Inspecting the call site we find the following region annotated call to look like this.

```
val v808 = insert[attop r5,attop r5,attop r1,
                  attop r5,attop r5,attop r7]
           <!ready_queue, pid>
```

It turns out that `r705` and `r714` are actually both bound to the region `r5`, again using `attop`. So in the end, during a context switch, just updating the ready and waiting queues are going to contribute to the build-up of data in region `r2` and `r5`. If the queues are long, this build-up could be quite substantial.
Similar problems arise for mailboxes as well, where a lot of the underlying data structures are placed in regions `r2`, `r5` and `r7`. Furthermore, mailboxes are mutable data structures, and thus make use of references. Every version of a mailbox remains alive at least as long as the process is alive. If the region inference algorithm cannot determine when a process dies, as is the case for the library described in this thesis, the data and associated references must remain alive for the entire duration of the program.
As there is a global value `current` which is a reference to the currently running process, all process identifiers have to be placed in the same region. This value naturally has to remain alive for the complete duration of a program, forcing process identifiers to stay alive.

5.3 Awkward Syntax

There are a couple of aspects of the library described here that makes writing client code awkward. Some of the quirks are because Standard ML is a statically typed language, while some are because Standard ML has no support for monads. Additionally, a lack of support for continuations is also a problem.

5.3.1 No Support for Continuations

As there is no support available to handle continuations exposed to the programmer, continuations have to be made explicit in the code. This creates a lot of parentheses and intermediary values. The logic of a process would preferably be written as an ordinary Standard ML function, but without being able to grab the continuation of a process this is not possible. Receiving two messages would ideally look something like this.

```
val intp : int pu = (* ... *)

fun process () =
  let val a = recv [unpickle intp]
      val b = recv [unpickle intp]
  in send (a+b, name "recipient") end
```

After receving the message `a` it is evident that the way to continue is with receiving the message `b`. As `recv` cannot grab the current continuation, however, the functionality must be implemented as follows.

```
fun process () =
  recv ([unpickle intp],         fn a  =>
  recv ([unpickle intp],         fn b  =>
  send ((a+b, name "recipient"), fn () =>
  endP)))
```

Here the continuations are expressed as two functions, one that receives `a` as an argument and one that receives `b`. At a context switch these must be allocated and wrapped up in a trampoline, which causes memory buildup. With support for continuations this problem could be solved by having e.g `recv` grab the current continuation if there was no message to receive, while this work now is left for the programmer.
In conclusion, support for continuations would eliminate the syntactic problem, but not the problem of memory buildup. The library has no way of interrupting a thread, grabbing its continuation and then letting another thread run. If this was the case the library could schedule threads, minimising the number of context switches while still remaining fair. As this is not the case, the context switches would still have to be done at points where a thread interacts with the library - e.g via sending messages.

5.3.2 Statically Typed Mailboxes

In a dynamically typed language such as Erlang, sending and receiving messages requires much less boilerplate code. Messages of arbitrary type can be sent without having to explicitly serialise them. In Erlang, a message is sent by evaluating `Expr1 ! Expr2`, where `Expr2` is the message being sent and `Expr1` identifies the receiver. When a process has been registered under a name, messages can be sent to that process simply by writing that name - without quotes. The name has to be an *atom*, which is a named constant. Consider the example written in Erlang below.

```
process() ->
  Xs = [1,2,3,4,5,6,7,8,9,10],
  calculator ! {add, 5, 5},
  calculator ! {sum, xs}.
```

Both applications of `!` send a message to the same recipient, but the messages are of different types. In the first application, the message is a tuple of size 3, where the first component is an atom and the other two are integers. In the second application, however, the message is a tuple of size 2, where the second component is a list of integers. While these are both valid expressions in Erlang, there is no way to assign a static type to the `!` operator.
In Standard ML we *must* give a type to an operator of this kind, as Standard ML is a statically typed language. The solution implemented in the library described in this thesis is to serialise messages, turning them all into values of type `string`. The price to pay for this solution is that messages must be serialised and deserialised explicitly. Ideally, this would be done under the hood by the compiler. Imagining that the atoms in the Erlang example would be turned into Standard ML strings, the above example could be implemented like this.

```
val trippu  : (string * int * int) pu = (* ... *)
val pairpu  : (string * int list)  pu = (* ... *)

fun process () =
let val xs = [1,2,3,4,5,6,7,8,9,10]
in
  send ((pickle trippu ("add", 5, 5), name "calculator"), fn () =>
  send ((pickle pairpu ("sum",xs),    name "calculator"), fn () =>
  endP))
end
```

The code for creating the `pu`'s is omitted, but it adds a few lines to the example as well. Similarly when messages are received, before anything sensible can be done to received messages, they must be deserialised. Rather than listing the receive clauses one after the other as in Erlang, in this implementation, the receive clauses must be explicitly converted to receive strings rather than the message type. While the overhead is manageable it is nonetheless overhead which we would preferably do without.
A smaller but still important detail is that in this implementation we match on messages by type rather than by value as in Erlang. In Erlang you can evaluate `receive {"two", "strings"} -> ok end`, which only receives a message if it is a tuple of size 2 where the components are the strings *two* and *strings*. In Standard ML, unless guarded by using `when` to selectively receive only the specific tuple in question, the branch will have to be prepared to handle tuples containing any two strings.

5.4 Processes Share Memory

At the heart of the project lies the actor concurrency model. As stated in section 2.3, a major benefit of this model is that the units of computation, the actors, do not share any memory.
The library described in this thesis does not accurately follow this description. The region inference algorithm treats the source code as a single program and annotates it as such. The fact that we are separating a program into several distinct logical units is not recognised by the algorithm. As a consequence of this, there is still a single stack of regions.
The fact that the regions are not separated across multiple stacks creates problems. One major problem is that a lot of data becomes very long-lived. Assuming some early regions belong to a process which is already dead, these regions cannot be deallocated until any region allocated after them has been deallocated. This is very redundant and could be a critical flaw in a memory-constrained system.
Even if the processes in the system described here cannot interact with each others memory, the single stack goes against the principle of keeping separate memory for each process. If the case was the opposite, that they properly received their own stack of regions, careful treatment of references is required. References can be serialised and sent in messages. If the reference references a value in a processes

memory, another process with that reference can affect the memory of that process, which again goes against the actor concurrency model.

5.5 Tail recursion

Iteration in functional languages is implemented using recursion. If the recursion depth is large, the price of allocating all those frame pointers can be quite large as well, as discussed in section 3.2.1. What is not discussed in that section is what to do about the arguments of the recursive functions. Assuming that a tail call is made with new arguments, what should be done about the arguments to the previous call? They will never be accessed by the function again, and could, in theory, be deallocated.
The storage mode analysis performed by the compiler and briefly mentioned in section 2.2.1 tries to recognise when this is the case. If possible it will put the new arguments in the same region using `atbot` to indicate that the region should be reset first. This would avoid the otherwise inevitable usage of region space that is proportional to the number of recursive calls.
Unfortunately, there is little room for tail call optimisation in this implementation. While the `run` function has a very distinct iterative pattern to what it does, it does not receive or produce any values. It performs side effects on global values.
Removing the side effects and making everything pure could perhaps make memory usage better - at the price of even more awkward syntax. Every call that would cause a side effect would need to provide its continuation. E.g spawning new functions would need not only the work of the spawned process but also the remaining work for the current process. A possible way to realise this solution could be to have the result of running a process be a function that will modify some environment appropriately, where that environment contains what is currently implemented as global values. The pseudocode below illustrates this point. This version of send returns a function that modifies an environment. The run function runs a process and applies the result of running that process to the environment.

```
fun send (msg,recip) cont =
  fn env => (* fetch recipient from env and send msg
              * update continuation of current process with cont *)

fun run env =
  let val modify =
    let val process = (* fetch process from env *) in
  (* run process *) end
  in run (modify env) end
```

This solution has not been explored.

6

Proposed Compiler Modifications and Future Work

In this chapter, we suggest modifications to the compiler that will hopefully address some of the issues pointed out in chapter 5. An outline of another implementation based on recent work on the MLKit is also presented.

6.1 Internal support for continuations

If the MLKit exposed a way to grab and execute continuations all the details of grabbing and applying continuations could be done in the library implementation, freeing the developer from the hassle that comes with it. A common interface to such functionality is via the two functions `callcc` and `throw`.

```
signature CONT =
  sig
    type 'a cont
    val callcc : ('a cont -> 'a) -> 'a
    val throw  :  'a cont -> 'a  -> 'b
    ...
  end
```

`callcc` stands for *call with current continuation*. The argument to `callcc` is a function that accepts the current continuation as its argument. If the function body evaluates some expression and produces a value, that value is returned and applied to the current continuation. If the continuation is invoked, however, it is as if the call to `callcc` returned the value the continuation was applied to.

```
val _ = print (callcc (
    fn continuation => "I am some string!" ^ "\n"))
>I am some string
```

In the above example, the continuation object within the call to `callcc` contains the continuation, which is a function that will print the result of the call to `callcc`. The continuation is not invoked inside the call to `callcc`, so the result of the entire call is the concatenated string `"I am some string!\n"`

```
val _ = print (callcc (
    fn continuation => "I am some string!" ^ (throw k "invoke" ^ "\n")))
>invoke
```

This example, however, invokes the continuation inside the call to `callcc`. It is applied to the string `"invoke"`, which is printed by the continuation.
Other Standard ML compilers already offer this functionality of `callcc` and `throw`, such as SMLofNJ[2][1]. Those compilers don't use Region-based memory management, however, which does not motivate using such a compiler for this thesis. Modifying MLKit to enable the use of `callcc` and `throw` could turn out to be a not so trivial affair. Consideration needs to be given to how these functions should fit in with the region aspect of things. When a continuation is invoked, regions will probably have to be popped off the stack to bring it to the state it was in at the time when `callcc` was called. The call to `callcc` might have, before the continuation was invoked, placed objects in regions that were alive before the call was made. Restoring the state of the memory will be tricky as these individual objects that should not be alive cannot be individually deallocated.

```
fun example (xs : 'a list) (ys : 'a list) =
  callcc (fn continuation =>
  let val xxs = xs@ys in throw continuation [] end)
```

Assume the two arguments to `example` reside in different regions. The call to `callcc` will produce a new list `xss` by using `@` to concatenate the lists. As `@` repeatedly prepends the elements of `xs` to `ys`, the resulting list `xss` has to be placed in the same region as `ys`. This is because `@` is *region endomorphic*. After this list has been created the continuation is invoked with the empty list, forcing `example` to return `[]`.
When invoking a continuation the stack should look like it did at the time when the continuation was created. In the example above, while the stack might be identical, the meaning of the region descriptors on the stack might not. The heap allocated region pages are modified to contain different values, which in this example is observed by `ys`'s region containing more values.
There are a few different ways to implement continuations. The case that is of interest here is when the runtime makes use of the system stack - which is the case in MLKit. Essentially the stack needs to be copied to the heap when the continuation is created, and then restored when the continuation is invoked. How this should be done when a lot of the stack entries are pointers to the heap could be non trivial.
Questions such as this one needs to be addressed before such a modification could be done to the compiler.

6.2 Mailbox Region

Messages in the library described in this thesis are not deallocated once received. This is very wasteful as we know that as soon as a message has been retrieved from

[1] https://www.smlnj.org/doc/SMLofNJ/pages/cont.html

the mailbox we will never attempt to retrieve it again. The entire region cannot be deallocated, however, as the mailbox itself and other messages within it might still be live.

The region inference algorithm already distinguishes different types of regions. Apart from regions being either finite or infinite, they are further classified with a region type. The possible region types are *real*,*string*,*top*,*word* and *bot*. They indicate what type of values can reside within the region. E.g string and real indicate regions which hold only strings or reals, while top is the type of regions that can contain any value.

It should be possible to add yet another type - a region type for the mailbox. The proposed region would be an infinite one, making use of region pages. Where infinite regions were previously defined to be triples, this new region type would be a quadruple `(e,fp,rp,a)`, adding a *read pointer*. The read pointer points to the first word which has not been read yet.

Mailboxes use a *first in first out (FIFO)* order of reads and writes. Some processes write in a specific order, and the recipient process reads in the same order. When an object is read from the region, that object begins at the read pointer. After reading an object the read pointer should be updated to be `rp + sizeof object`. If `rp > fp + sizeof reagion-page` then the next message begins in the second region page, and the first page will never be read from again. At this point, it should be safe to deallocate `fp`. To maintain the size of the mailbox a fresh region page should be appended to the end of the list of region pages. In reality `fp` can probably be reset and appended to the end of the list of region pages, and the end pointer updated to point to the end of the new page instead.

One issue with a mailbox such as this one is that we can move away from the FIFO behaviour of the mailbox via selective receive. Instead of always retreiving a message from the first region page, we could choose not to receive the oldest one and receive another one instead. Simply deallocating the first page when the read pointer points to the second page is not possible, as there might be unreceived messages left. More careful thought needs to be given to the design of a mailbox region.

Disregarding the last point; While this idea would work well at deallocating messages that are no longer needed, it imposes some restrictions on the mailboxes. They now have a fixed size defined by the number of region pages pointed to by the region descriptor. We argue, however, that this problem would eventually have to be addressed anyway; what to do about the size of mailboxes.

If they are completely unbounded in size they could potentially consume all available resources until the system runs out of memory. Traditionally when this happens a garbage collector would initiate and try to reclaim memory and possibly, as in Erlang, request a larger memory area. Doing nothing does not sound like an ideal setting as part of the motivation for using regions is that allocations and *deallocations* are explicit in the annotated code, having some guarantees that all memory will eventually be reclaimed.

When the mailbox is full a choice has to be made of what to do. One alternative is to drop messages, either the new messages or the oldest messages. Another alternative is to let the programmer write some garbage collector-like behaviour for the mailbox. Yet another alternative, perhaps the simplest one, is to let the mailbox fetch more region pages as they are needed. The mailbox would then become an ordinary

infinite region, with the exception that region pages can be deallocated from the front. Perhaps this alone provides good enough performance.

6.3 Thread Capabilities Implemented by Martin Elsman

Near the end of this project Martin Elsman - one of the main implementors of MLKit - pushed additional features to the code repository for MLKit[2]. These new features include the ability to spawn threads and join on threads. The function that spawns a thread will only return after the spawned thread has terminated, at which point the regions asssociated with the spawned thread can be safely deallocated.[7] As the modifications were mainly applied to the runtime system, it is not necessary to dive deep into the compiler to add additional functionality.
The modifications were implemented using posix threads (pthreads). When a function is spawned to run in parallel with another, a pthread is created to do the work. The parent process in Standard ML receives a handle it can use to join on the thread, receiving the value produced by the spawned function.
As these features became available late in the project, there was no time left to add proper message passing capabilities to compare the performance of modifying the runtime system to the library described in this thesis. Despite this, a very small proof of concept was implemented to get a feel for what the syntax would look like. A complete implementation where the runtime system is modified rather than implementing the library in pure Standard ML most likely mitigates a few of the problems described in section 5.
The most immediate improvement is that the need for explicit continuations in client code is not as great. When sending and receiving messages the programmer is relieved from the burden of writing explicit continuations. In contrast, spawning a new process currently requires an explicit continuation for the parent process. Further investigation is required concerning the possibility of removing this necessity.
As more of the functionality is implemented in the runtime system, it is not up to the region inference algorithm to handle as much data associated with the implementation. Context switches are handled by the pthreads library, meaning that there is no need to create trampolines.
Retrieving a message is conditioned on the fact that a message is in the mailbox. If this is not the case a thread can wait until the condition becomes true. When another thread delivers a message it will signal that the condition is now true, and any thread that is waiting for a message is woken up. Consider the pseudocode below.

```
Condition cond;

recv():
  if message exists
```

[2]The changes can be found on the branch "spawn" in the GitHub repository: https://github.com/melsman/mlkit/tree/spawn

```
  then return message
  else wait(cond)

send(Thread t, Msg msg)
  deliver msg to t
  signal(cond)
```

The two threads are here using the same condition variable to synchronise. In a real implementation, every mailbox would be associated with a unique condition variable. If there were only one such variable all threads would wake up every time the condition is signalled to be true.
The small proof of concept that has been implemented enables a user to spawn functions, send strings and receive strings. The mailbox has size only for one message, and a message can only be sent if it is empty. In the current implementation, when a thread is created, a struct of type `ThreadInfo` is allocated and initialised. The modification applied to this struct to implement very simple message passing is as described below.

```
typedef struct {
  ...
  pthread_mutex_t condition_mutex;
  pthread_cond_t condition;
  ...
  char message[MSG_SIZE];
} ThreadInfo;
```

The mailbox is represented as a `char` array, capable of holding a single message in string format. The function below is implemented when sending a message.

```
void send(ThreadInfo* ti, int msg) {
  pthread_mutex_lock(&(ti->condition_mutex));

  if(ti->message[0])
    pthread_cond_wait(&(ti->condition), &(ti->condition_mutex));

  convertStringToC((StringDesc*)msg, ti->message, MSG_SIZE, 0);
  pthread_cond_signal(&(ti->condition));

  pthread_mutex_unlock(&(ti->condition_mutex));
}
```

First, a check is performed to see if the first byte in the mailbox is anything other than zero. If it is zero this would indicate that the mailbox is empty, and we could proceed with delivering the message. In the other case, it is not empty, and we will have to wait on the condition associated with the receiving thread before we can continue with delivering the message. Lastly, in the case that the recipient is waiting for a message, the condition is signalled.
Receiving a message is not too different.

```
int recv(Region stringRho, int exn) {
  ThreadInfo* ti = (ThreadInfo*)pthread_getspecific(threadinfo_key);
  pthread_mutex_lock(&(ti->condition_mutex));

  if(!(ti->message[0]))
    pthread_cond_wait(&(ti->condition), &(ti->condition_mutex));

  int res = (int) convertStringToML(stringRho, ti->message);
  ti->message[0] = 0;
  pthread_cond_signal(&(ti->condition));

  pthread_mutex_unlock(&(ti->condition_mutex));
  return res;
}
```

The `ThreadInfo` struct associated with the currently running thread is fetched, followed by a check to see if there is a message to receive or not. If there is none, the thread will wait until someone - a thread sending a message - signals the condition. When there is a message to receive, it is written to the region bound to the variable `stringRho`. After this, a zero byte is written to the first position in the mailbox to indicate that it is empty.
A simple example of using this library is given below.

```
val _ = spawn
        (fn () => let
                    val m1 = recv ()
                    val m2 = recv ()
                    val m3 = recv ()
                  in print (m1 ^ "\n" ^ m2 ^ "\n" ^ m3 ^ "\n") end)

        (fn t => (send ("First message",t);
                  send ("Second message",t);
                  send ("Last message",t)))
>First message
>Second message
>Last message
```

In this case the messages are ordinary strings, but they might as well be serialised data. Sending and receiving looks cleaner as there is no need to pass in the continuation. Messages are received in the proper order as well, which might not be too surprising as the mailbox in this case only had room for one message.

7
Related Work

The following section describes related work in areas concerning the thesis. Related work can be categorised as either being related to concurrent functional programming or region based memory management.

7.1 MLKit

The MLKit[18] is the fruit of pioneering work in the area of region based memory management. It is a compiler for standard Meta Language (Standard ML) that employs the use of regions, and is the first attempt at implementing regions for a complete general purpose language. It applies a rigorous region inference analysis on the source code and then annotates the code with primitives whose semantics describe allocation and deallocation of regions of memory. If the analysis can determine the size of an object statically, the object is placed on the stack. If an objects size cannot be determined statically, however, the choice is made to put it in a region instead. Furthermore the compiler is accompanied by a soundness proof that guarantees that no dangling references are dereferenced.
The code generated by the algorithm will at runtime organise the memory as a *stack* of regions. This comes with some limitations, an important one being that for programs where memory is live throughout the lifetime of a program, memory usage is poor. What is meant by live is illustrated in Figure 7.1. To exemplify this we consider a program with a never ending *loop*, such as server software. The scope

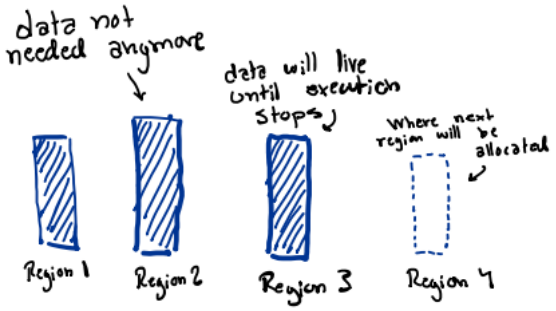

Figure 7.1: Since the regions used by MLKit is organised as a stack, region 2 cannot be deallocated before region 3 have been deallocated. If region 3 is allocated in a never ending loop, any region allocated before it can never be deallocated.

of the loop will constitute the topmost region on the stack of regions, and thus any

memory allocated before entering the loop will never be deallocated.
Using the MLKit compiler is simultaneously challengening and convenient. It is convenient as the source code requires no annotations about where and how to use regions, it is all inferred by the region inference algorithm. Any developer able to write ML code can use the compiler with no additional investment. It is challenging in the sense that some knowledge of how the region inference algorithm works is required in order to write programs that utilise regions fully. These programs are referred to as region optimised programs. To teach developers how to create region optimised programs, a manual describing the process has been published[18].
In 2002 Niels Hallenberg, for his masters thesis, extended the MLKit runtime with the option to use a garbage collector[12]. The implementation is a variant of Cheneys copying garbage collector that is applied to each region locally. The choice of algorithm used by the garbage collector has an interesting effect on the region analysis. The analysis must be weakened to disallow dangling pointers of any sort. With just region analysis the algorithm would allow dangling pointers if it could be statically determined that these pointers would never be dereferenced. This weakening must happen as the garbage collector will follow all pointers when copying memory, and would therefore dereference dangling pointers.
Combining region inference with the garbage collector significantly reduces the number of times the collector is activated, but memory usage is still increased in the general case. Programs that are not optimised for regions run better under the combination, while the opposite holds for region optimised programs.

7.2 Cyclone

Cyclone[11, 13, 14] is an attempt at creating a type-safe version of C. Many techniques are involved, of which one is region based memory management. To illustrate how regions are used to guarantee type-safe programs, consider the example below[1]which will produce a warning if it is compiled with `gcc -Wall`.

```
char *itoa(int i) {
  char buf[20];
  sprintf(buf,"%d",i);
  return buf;
}
```

`gcc` is going to warn the programmer that the address of a local variable is being returned. By just modifying the example slightly, however, the warning will not be produced but the bug is still present.

```
char *itoa(int i) {
  char buf[20];
  char* z;
```

[1]Example is taken from Jim, T., Morrisett, J.G., Grossman, D., Hicks, M.W., Cheney, J. and Wang, Y., 2002, June. Cyclone: A Safe Dialect of C. In USENIX Annual Technical Conference, General Track (pp. 275-288).

```
  sprintf(buf,"%d",i);
  z = buf;
  return z;
}
```

Where `gcc` would consider this a legal program without warnings, Cyclone uses region analysis to determine that this is not a type-safe program. It does this by considering every block to be a region, and infers that both `buf` and `z` exist in the same region and will be deallocated when the method returns.
The region analysis employed by Cyclone places every object in a region. The usual pointer type τ`*` is replaced with τ`*p`, where `p` is the region the referenced object resides in. Arrays of unknown size gets the more specialised type τ`?p`. With this extra information carried by the type of pointers, a subtyping relationship is established between pointer types. As regions form a *Last-in First-out* stack in Cyclone as well, a region τ`*`$\mathtt{p}_1$ is a subtype of a region τ`*`$\mathtt{p}_2$ if $\mathtt{p}_1$ *outlives* $\mathtt{p}_2$
Similarly as in the MLKit, Cyclone initially encountered the same restrictions imposed by the LIFO stack of regions. Unique pointers were implemented to address this, which are pointers to which only one reference is allowed to exist. They are allocated with *malloc* and deallocated with *free*, and since only one reference is allowed to exist, the call to *free* is always safe. To give the programmer *some* flexibility when using these pointers, Cyclone supports *borrowed pointers*, which are copies of unique pointers that cannot escape or be deallocated.

7.3 Region based memory management for Java

The idea of implementing regions for a language such as Java has enticed a lot of people. By design, the language makes heavy use of references and the heap, which is made possible by the use of a garbage collector.
The programming paradigm employed by Java is that of *Object Orientation*. Common features of object oriented languages are objects, classes, inheritance, subtyping and method overloading, to name a few. Naturally the prospect of writing a compiler for a complete specification of Java is a daunting task. With this in mind, attempts have been made at evaluating how the region idiom performs on subsets of the Java specification.
For their thesis, Christiansen and Velschow[6] defined a subset of Java, named RegJava. Apart from the core specification being a subset of Java, it was extended to include annotations describing the allocation, deallocation and updating of regions. Based on the formalisation of a static and dymamic semantics, a soundness proof is presented. They report their system to perform roughly the same as garbage collected systems, both concerning memory usage and execution speed. They go on to emphasise that their work is meant to lay the foundation for future work. A stronger inference analysis is required, some proofs need a little more care and there are parts of the Java specification not yet implemented in their system.
A formalisation for the complete specification of Java was presented by Cherem and Rugina[5]. Apart from applying regions to the entire Java language, a difference to the work by Christiansen and Velschow[6] is that regions are not lexically scoped.

Yet another distinguishing feature is that the system allows dangling pointers so long as they are not dereferenced. This means that regions can be deallocated when they are no longer needed and not just when they are no longer referenced.
The region inference algorithm employed builds a points-to graph between regions. Following this, the algorithm determines which regions are live at each point in the program by keeping track of live variables and consulting the points-to graph. Lastly, the source code is annotated with region allocation and deallocation points. These are compiled to new bytecode instructions which the Kaffe VM[21] has been extended to properly handle.
The authors refer to their results as encouraging. The specific memory behaviour observed depended heavily on the application in question. Again, where memory was not live throughout execution, very good results were reported. For some benchmarks significant memory savings were reported, and for those where no gains were reported it seemed like the garbage collector also struggled to save memory.

7.4 Cloud Haskell

Cloud Haskell[8] is a project that successfully brought the concurrency style of Erlang to Haskell. The primitives are exposed to the programmer as a Domain Specific Language (DSL). An important difference is that Cloud Haskell matches messages by message type and not by message value as Erlang does.
When a message is received by a Cloud Haskell process, a function handle is applied to it. As with every function, if the function does not exhaustively match all potential values, an exception will be thrown if it is applied to a value it has no clause for. An example situation is where a value of type *Maybe Int* is expected, but the function receiving the message only has a clause for *Just i*. If a value *Nothing* arrives, the type matches and the function will be applied, followed by an exception being thrown. In Erlang, the value *Nothing* would not match the expected value *Just i*, and the message would be left in the mailbox.
In addition to being able to send any message that is serialisable, Cloud Haskell offers the use of typed channels. To use typed channels, a send port and a receive port are created. These ports only allow transmitting messages of a specific type. The send port can be serialised and transmitted to other processes, but the receive port cannot. This is a deliberate design choice that is based on the intended execution environment. The intended use of channels is to communicate over a network. It is not clear what should happen to messages that are sent when a receive port is in transit, and so this is disallowed.

7.5 Manticore

Manticore[9] is a high-level functional programming language with a syntax that is heavily inspired by Standard ML. The concurrency model employed by Manticore uses both coarse-grain parallelism and fine-grain parallelism.
Coarse-grain parallelism is offered by the ability to spawn threads. Instead of having a shared memory area for all threads, threads send and receive messages over syn-

chronous channels. The synchronous channels are typed and allow only transmitting data of a particular type. An important note to emphasise here is that the channels are synchronous; a thread will block until communication is over. Not only will a receiving thread block until another thread is sending a message, a sending thread will block until there is a thread receiving the message.
Fine-grain parallelism is implicit in the computations performed by a program. Inspired by NESL[4], Manticore offers the use of array comprehensions and parallel tuple expressions. In cases where such expressions are nested, the compiler will flatten the expression and perform the individual computations in parallel. The computations are done in parallel since such data objects are *data-parallel*. Data-parallelism is when a piece of data says something about how computations on it can be parallelised. A very intuitive example being that of mapping a function over an array. Each sub-computation for any cell can be done in parallel with computing the sub-computations for the other cells.
Despite the arrays employed by Manticore being data parallel, parallel arrays still have sequential semantics. This means that the compiler has to verify that the sub-computations in a data parallel computation do not send or receive any messages. If a sub-computation sends or receives a message, that computation might have a side effect that changes the result of another computation, which acccording to the semantics should finish first. Additionally, if an exception is raised by a sub-computation, the actual reporting of the exception must be delayed until all prior sub-computations have finished.
At runtime the system heap is divided into several local chunks and a global heap area. Apart from being able to use the global heap, each thread has a local chunk of memory it can use. When a threads local chunk grows too small, the thread can garbage collect that chunk alone without blocking the global heap. All threads will only synchronise when the global heap needs to be extended with more memory.

8

Conclusions

While the main contributions of this thesis are analytical, we have also contributed concretely with a library implementation of the actor concurrency model. At the same time, we have brought the untyped message-passing model of Erlang to the very typed setting of Standard ML. The style in which concurrent processes are written becomes very verbose as a consequence. Having to convert messages of any type to all be of the same type adds boilerplate code to the client code.

The library has been instrumental in analysing how current techniques of Region-based memory management work in an actor model setting. The results indicate that the region inference algorithm is not well suited to handle coroutines. This is not surprising as the algorithm is defined to consider the source code as a single program.

As the allocation points of regions are lexically scoped and form a stack of regions at runtime, the memory of different threads is not separate. Without having multiple stacks of regions memory performance will not be optimal, as the deallocation of regions will be dependent on which is the topmost region. The different threads should only be affected by other threads via messages; if a thread could deallocate its topmost region it should be able to do so regardless of what the other threads are doing.

Not being able to deallocate data that will never be referenced again, as the region the data resides in is still live, turns out to be a major problem in this thesis. The act of receiving messages is a well-defined thing, where the mailbox hands over a message and definitely will never need to reference it again afterwards. Similarly, during a context switch, a new continuation is created, making the previous continuation obsolete. Despite knowing this the old continuation is never deallocated and will contribute to the build-up of memory. A proper implementation would make sure to disregard the rule that forbids the deallocation of individual objects. When a message is received the memory it occupied in the mailbox should be freed, and when a new continuation is created it should replace an old one.

While the results indicate that current techniques are not well suited for this type of concurrency, they do not confirm that Region-based memory management as a management principle is unsuitable. There are quite a few avenues left to explore before it can be determined if that is the case or not. What is most interesting to investigate next is to explore the capabilities of the experimental work that has been pushed to the code repository by Martin Elsman. An implementation that modifies the runtime system would mitigate some of the problems present in the library implementation described in this thesis - most notably the issues concerning explicit continuations and the use of global variables. The global variables that

could be eliminated are e.g the ready queue, waiting queue and the current process, as this information would then be managed by the pthreads library.
What is not optimal in such an implementation is that then a program would have a lot of important information about threads and mailboxes that resides outside of regions. Ideally, thread handles and their mailboxes would be region allocated as well, ensuring that all memory is accounted for.

Bibliography

[1] Andrew W Appel. *Compiling with continuations*. Cambridge University Press, 2007.

[2] Andrew W Appel and David B MacQueen. Standard ml of new jersey. In *International Symposium on Programming Language Implementation and Logic Programming*, pages 1–13. Springer, 1991.

[3] Joe Armstrong. erlang. *Communications of the ACM*, 53(9):68–75, 2010.

[4] Guy E. Blelloch. NESL: A nested data-parallel language (version 2.6). Technical Report CMU-CS-93-129, School of Computer Science, Carnegie Mellon University, apr 1993. Accessed June 2020, http://www.cs.cmu.edu/afs/cs.cmu.edu/project/scandal/public/papers/CMU-CS-93-129.html.

[5] Sigmund Cherem and Radu Rugina. Region analysis and transformation for java programs. In *Proceedings of the 4th international symposium on Memory management*, pages 85–96. ACM, 2004.

[6] Morten V Christiansen and Per Velschow. Region-based memory management in java. Master's thesis, DIKU, University of Copenhagen, 1998.

[7] Martin Elsman. Private communication.

[8] Jeff Epstein, Andrew P. Black, and Simon Peyton-Jones. Towards haskell in the cloud. *SIGPLAN Not.*, 46(12):118–129, September 2011.

[9] Matthew Fluet, Mike Rainey, John Reppy, Adam Shaw, and Yingqi Xiao. Manticore: A heterogeneous parallel language. In *Proceedings of the 2007 workshop on Declarative aspects of multicore programming*, pages 37–44. ACM, 2007.

[10] Steven E Ganz, Daniel P Friedman, and Mitchell Wand. Trampolined style. In *Proceedings of the fourth ACM SIGPLAN international conference on Functional programming*, pages 18–27, 1999.

[11] Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-based memory management in cyclone. In *ACM Sigplan Notices*, volume 37, pages 282–293. ACM, 2002.

[12] Niels Hallenberg, Martin Elsman, and Mads Tofte. Combining region inference and garbage collection. *SIGPLAN Not.*, 37(5):141–152, May 2002.

[13] Michael Hicks, Greg Morrisett, Dan Grossman, and Trevor Jim. Experience with safe manual memory-management in cyclone. In *Proceedings of the 4th international symposium on Memory management*, pages 73–84. ACM, 2004.

[14] Trevor Jim, J Gregory Morrisett, Dan Grossman, Michael W Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of c. In *USENIX Annual Technical Conference, General Track*, pages 275–288, 2002.

[15] Simon Peyton Jones (editor). *Haskell 98 language and libraries: the revised report*. Cambridge University Press, 2003.

[16] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part 1. *Commun. ACM*, 3(4):184–195, April 1960.

[17] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The definition of standard ML: revised*. MIT press, 1997.

[18] Mads Tofte, Lars Birkedal, Martin Elsman, Niels Hallenberg, Tommy Højfeld Olesen, Peter Sestoft, and Peter Bertelsen. Programming with Regions in the ML Kit. *DIKU Rapport*, 97:12, 1997. Accessed June 2020, https://di.ku.dk/forskning/Publikationer/tekniske_rapporter/1997/97-12.pdf.

[19] Mads Tofte and Jean-Pierre Talpin. Implementation of the typed call-by-value λ-calculus using a stack of regions. In *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 188–201, 1994.

[20] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, 132(2):109 – 176, 1997.

[21] Tim Wilkinson. Kaffe - a free java virtual machine. http://www.kaffe.org, 1996. Accessed: August 26, 2019.

A

Appendix 1

The entire code base can be found at the GitHub repository: https://github.com/Rewbert/master-project. Below follow the code for the benchmark programs.

A.1 skynet

The Standard ML version.

```
structure A : ActorConc = A
structure P : PICKLE    = Repickle

fun process 0 pid () = A.send ((P.pickle P.int 1, A.pid pid), fn () => A.endP)
  | process r pid () =
  let
    val self = A.self ()
  in (A.spawn (process (r-1) self);
      A.spawn (process (r-1) self);
      A.spawn (process (r-1) self);

      A.recv_many 3 ([P.unpickle P.int], fn res =>
      A.send ((let val sum = foldl (op +) 0 res in
                P.pickle P.int sum end, A.pid pid), fn () =>
      A.endP)))
  end

fun printer () : A.trampoline = (
  A.recv ([P.unpickle P.int], fn i => (
  print ((Int.toString i) ^ "\n");
  A.endP)))

fun main () : unit =
let
  val pid = A.spawn printer
  val _   = A.spawn (process 6 pid)
in (
  A.run ())
end
```

```
val _ = main ()
```

The Erlang version.

```
-module(skynet).
-compile(export_all).

process(0, Pid) -> Pid ! 1 ;
process(R, Pid) ->
Self = self(),

% spawn three children
lists:foreach(fun(_) -> spawn(fun() -> process((R-1), Self) end) end, [1,2,3]),

% construct three receives and then evaluate them
Recvs = lists:map(fun(_) -> receive M -> M end end, [1,2,3]),
Msgs  = [ M || M <- Recvs],

% transmit sum to parent
Pid ! lists:sum(Msgs) .

printer() -> receive M -> io:format("~w~n", [M]) end.

main() ->
Pid = spawn(fun() -> printer() end),
spawn(fun() -> process(6, Pid) end),
ok.
```

A.2 Message bombing

The Standard ML version.

```
structure A : ActorConc = A
structure P : PICKLE    = Repickle

datatype Msg = Msg of int * int
val msgPickler : Msg P.pu =
let
  fun index _  = 0
  fun fun_M pu = P.con1 Msg (fn Msg p => p)
                        (P.pairGen(P.int,P.int) : (int * int) P.pu)
in P.dataGen("Msg", index, [fun_M]) end

datatype Badmsg   = Badmsg of int * int
val badmsgPickler : Badmsg P.pu =
```

```
let
  fun index _  = 0
  fun fun_M pu = P.con1 Badmsg (fn Badmsg p => p)
                        (P.pairGen(P.int,P.int) : (int * int) P.pu)
in P.dataGen("Badmsg", index, [fun_M]) end

fun replicate 0 a = []
  | replicate n a = a :: replicate (n-1) a

(* ********** Boilerplate code above this point ***********)

fun processA 0 a pid () =
      A.send (let val msg = P.pickle P.int a
              in (msg, pid) end, fn () =>
      A.endP)
  | processA n a pid () =
      A.recv ([A.conv (fn i => i) msgPickler], fn (Msg (num,den)) =>
      processA (n-1) (a + Int.div (num, den)) pid ())

(* This process sends messages that are not received *)
fun processB n pid () =
      let val msgs = let val value = Badmsg (6,3) in
                       let val serialise = P.pickle badmsgPickler in
                         let val msg = serialise value in
                           replicate n (msg, pid)
                         end
                       end
                     end
      in A.send_many (msgs, fn () => A.endP) end

fun processC n pid () =
      let val msgs = let val value = Msg (6,3) in
                       let val serialise = P.pickle msgPickler in
                         let val msg = serialise value in
                           replicate n (msg, pid)
                         end
                       end
                     end
      in A.send_many (msgs, fn () => A.endP) end

fun run () = let
  val pid = A.pid (A.self ())
  val a   = A.pid (A.spawn (processA 200 0 pid))
in (
  A.spawn (processB 200 a); (* Comment out this line to only
                             * send messages that are received. *)
```

```
  A.spawn (processC 200 a);
  A.recv ([A.conv (fn i => i) P.int], fn r =>
  (print ((Int.toString r) ^ "\n");
  A.endP))) end

val _ = (A.spawn run; A.run ())
```

The Erlang version.

```
-module(messagebombing).
-compile(export_all).


processA(0, A, Pid) -> Pid ! A;
processA(N, A, Pid) ->
    receive
      {msg, Num, Den} -> processA((N-1), A + (Num / Den), Pid)
    end.

processB(0, _)   -> ok;
processB(N, Pid) ->
    Pid ! {badmsg, 6, 3},
    processB((N-1),Pid).

processC(0, _)   -> ok;
processC(N, Pid) ->
    Pid ! {msg, 6, 3},
    processC((N-1), Pid).

main() ->
  Pid = self(),
  A = spawn(fun() -> processA(200, 0, Pid) end ),
  spawn(fun() -> processB(200, A) end), % comment out this line to
                                        % only send messages that will be received
  spawn(fun() -> processC(200, A) end),
  receive
    Res -> Res
  end.
```

A.3 Bitonic Mergesort

The Standard ML version.

```
structure A : ActorConc = A
structure P : PICKLE    = Repickle
```

```
(************** List Utilities ***************)
fun split n xs = (List.take (xs, n), List.drop (xs, n))
fun zip xs [] = []
  | zip [] xs = []
  | zip (x::xs) (y::ys) = (x, y) :: zip xs ys

(************** Either type ******************)
datatype ('a, 'b) Either = Left of 'a | Right of 'b

fun leftPickler pa =
let
  fun index _ = 0
  fun fun_L pu = P.con1 Left (fn Left i => i | _ => raise P.UnpackError) pa
in P.dataGen("EitherLeft", index, [fun_L])
end

fun rightPickler pb =
let
  fun index _ = 0
  fun fun_R pu = P.con1 Right (fn Right i => i | _ => raise P.UnpackError) pb
in P.dataGen("EitherRight", index, [fun_R])
end

fun bothPickler pa pb =
let
  fun index (Left _)  = 0
    | index (Right _) = 1
  fun leftP  pu = P.con1 Left  (fn Left i  => i | _ => raise P.UnpackError) pa
  fun rightP pu = P.con1 Right (fn Right i => i | _ => raise P.UnpackError) pb
in P.dataGen("Either", index, [leftP, rightP])
end

fun myappend [] []      = []
  | myappend [] (y::ys) = y :: myappend [] ys
  | myappend (x::xs) [] = x :: myappend xs []
  | myappend (x::xs) ys = x :: myappend xs ys

(************** Sequential Bitonic Mergesort ****************)
fun bitonic_merge_sequential []  = []
  | bitonic_merge_sequential [x] = [x]
  | bitonic_merge_sequential xs  =
  let
    val (left, right) = split (Int.div (length xs, 2)) xs
    val zipped = zip left right
    val mins = map Int.min zipped
    val maxs = map Int.max zipped
```

```
  in myappend (bitonic_merge_sequential mins) (bitonic_merge_sequential maxs)
  end

fun bitonic_mergesort_sequential []  = []
  | bitonic_mergesort_sequential [x] = [x]
  | bitonic_mergesort_sequential xs  =
  let
    val l = length xs
    val (left, right) = split (Int.div (l,2)) xs
    val leftsorted  = bitonic_mergesort_sequential left
    val rightsorted = List.rev (bitonic_mergesort_sequential right)
  in bitonic_merge_sequential (myappend leftsorted rightsorted) end

(************** Concurrent Bitonic Mergesort ****************)
fun bitonic_merge xs cutoff pid () =
let
  val l       = length xs
  val pickler = P.listGen P.int
in if l <= cutoff

   then let val msg = P.pickle pickler (bitonic_merge_sequential xs)
        in A.send ((msg, pid), fn () => A.endP) end

   else let val (left, right) = split (Int.div (l,2)) xs
            val zipped        = zip left right
            val mins          = map Int.min zipped
            val maxs          = map Int.max zipped
            val parent        = A.pid (A.self ())

            fun merge_maxs () =
            let val parent' = A.pid (A.self ())
            in (
              A.spawn (bitonic_merge maxs cutoff parent');
              A.recv ([A.conv (fn i => i) pickler], fn r =>

              let val msg = P.pickle (leftPickler pickler) (Left r)
              in A.send ((msg, parent), fn () => A.endP) end))
            end

            fun merge_mins () =
            let val parent' = A.pid (A.self ())
            in (
              A.spawn (bitonic_merge mins cutoff parent');
              A.recv ([A.conv (fn i => i) pickler], fn r =>

              let val msg = P.pickle (rightPickler pickler) (Right r)
```

```
              in A.send ((msg, parent), fn () => A.endP) end))
            end

        in (A.spawn merge_maxs;
            A.spawn merge_mins;
            A.recv ([A.conv (fn (Left i) =>  i | _ => raise P.UnpackError)
                            (leftPickler  pickler)], fn maxsmerged =>
            A.recv ([A.conv (fn (Right i) => i | _ => raise P.UnpackError)
                            (rightPickler pickler)], fn minsmerged =>

            let val res = myappend minsmerged maxsmerged
            in A.send ((P.pickle pickler res, pid), fn () =>A.endP)
            end)))

        end
end

fun bitonic_mergesort xs cutoff pid () =
let
  val l       = length xs
  val pickler = P.listGen P.int
in if l <= cutoff

   then let val msg = P.pickle pickler (bitonic_mergesort_sequential xs)
        in A.send ((msg, pid), fn () => A.endP) end

   else let val (left, right) = split (Int.div (l,2)) xs
            val parent = A.pid (A.self ())

            fun sort_right () =
            let
              val parent' = A.pid (A.self ())
            in (A.spawn (bitonic_mergesort right cutoff parent');
                A.recv ([A.conv (fn i => i) pickler], fn r =>

                let val msg = P.pickle (rightPickler pickler) (Right r)
                in A.send ((msg, parent), fn () => A.endP) end))
            end

            fun sort_left () =
            let
              val parent' = A.pid (A.self ())
            in (A.spawn (bitonic_mergesort left cutoff parent');
                A.recv ([A.conv (fn i => i) pickler], fn r =>

                let val msg = P.pickle (leftPickler pickler) (Left r)
```

```
                in A.send ((msg, parent), fn () => A.endP) end))
            end

        in (A.spawn sort_right;
            A.spawn sort_left;
            A.recv ([A.conv (fn (Right i) => i | _ => raise P.UnpackError)
                          (rightPickler pickler)], fn (rightsorted) =>
            A.recv ([A.conv (fn (Left i) => i  | _ => raise P.UnpackError)
                          (leftPickler  pickler)], fn (leftsorted)  =>

            let val res = myappend leftsorted (List.rev rightsorted)
            in (A.spawn (bitonic_merge res cutoff pid); A.endP)
            end)))
        end
end


fun receiver () =
   A.recv ([A.conv (fn i => i) (P.listGen P.int)], fn res =>
   A.endP)

fun descList 0 = [0]
  | descList n = n :: descList (n-1)

fun main () : unit =
let
  val pid = A.spawn receiver
  val _   = A.spawn (bitonic_mergesort (descList 127) 2 (A.pid pid))
in
  A.run ()
end

val _ = main ()
```

The Erlang version. The Erlang version of this benchmark was inspired by this GitHub repository. https://github.com/vanHavel/Parallel-Functional-Programming-Benchmarks

```
-module(bitonicmsort).
-compile(export_all).

descList(0) -> [0];
descList(N) -> [N | descList(N-1)].

% Important: XS length must be a power of 2!
bitonic_merge(Xs, Cutoff) ->
  L = length(Xs),
```

```
  if
    L =< Cutoff ->
      bitonic_merge_sequential(Xs);
    true ->
      {Left, Right} = lists:split(L div 2, Xs),
      Mins = [min(A,B) || {A,B} <- lists:zip(Left, Right)],
      Maxs = [max(A,B) || {A,B} <- lists:zip(Left, Right)],
      Parent = self(),
      R = make_ref(),
      spawn_link(fun() ->
        Result = bitonic_merge(Maxs, Cutoff),
        Parent ! {R, Result}
      end),
      MinsMerged = bitonic_merge(Mins, Cutoff),
      MaxsMerged = receive
        {R, Result} -> Result
      end,
      lists:append (MinsMerged, MaxsMerged)
end.

bitonic_merge_sequential([]) -> [];
bitonic_merge_sequential([X]) -> [X];
bitonic_merge_sequential(Xs) ->
  {Left, Right} = lists:split(length(Xs) div 2, Xs),
  Mins = [min(A,B) || {A,B} <- lists:zip(Left, Right)],
  Maxs = [max(A,B) || {A,B} <- lists:zip(Left, Right)],
  MinsMerged = bitonic_merge_sequential(Mins),
  MaxsMerged = bitonic_merge_sequential(Maxs),
  lists:append (MinsMerged, MaxsMerged).

% Important: XS length must be a power of 2!
% more precisely, the length must be of the form Cutoff * 2^n for some n
bitonic_mergesort(Xs, Cutoff) ->
  L = length(Xs),
  if
    L =< Cutoff ->
      bitonic_mergesort_sequential(Xs);
    true ->
      {Left, Right} = lists:split(L div 2, Xs),
      Parent = self(),
      R = make_ref(),
      spawn_link(fun() ->
        Result = lists:reverse (bitonic_mergesort(Right, Cutoff)),
        Parent ! {R, Result}
      end),
      LeftSorted = bitonic_mergesort(Left, Cutoff),
```

```
      RightSorted = receive
        {R, Result} -> Result
      end,
      bitonic_merge (lists:append(LeftSorted, RightSorted), Cutoff)
      %receive _ -> ok end
  end.

main() -> bitonic_mergesort(descList(127), 2).

bitonic_mergesort_sequential([]) -> [];
bitonic_mergesort_sequential([X]) -> [X];
bitonic_mergesort_sequential(Xs) ->
  L = length(Xs),
  {Left, Right} = lists:split(L div 2, Xs),
  LeftSorted = bitonic_mergesort_sequential(Left),
  RightSorted = lists:reverse(bitonic_mergesort_sequential(Right)),
  bitonic_merge_sequential (lists:append(LeftSorted, RightSorted)).
```