Thesis Proposal: Improving the Performance of Synchronization in Concurrent Haskell\textsuperscript{1}

Ryan Yates  
ryates@cs.rochester.edu  

May 3, 2014

\textsuperscript{1}This work was supported in part by the National Science Foundation under grants CCR-0963759, CCF-1116055, CNS-1116109, and CCF-1337224.
## Contents

1 Introduction  

2 Background  
   2.1 Introduction to Functional Programming  
      2.1.1 Purity  
      2.1.2 Functional Programming  
      2.1.3 Evaluation Order  
      2.1.4 Parametricity  
   2.2 State and Effectful Computations  
      2.2.1 State Monad  
      2.2.2 IO Monad  
      2.2.3 ST Monad  
      2.2.4 Concurrent and Parallel Haskell  
      2.2.5 The Effect Zoo  
   2.3 Single-threaded Execution  
   2.4 Multicore Execution  
      2.4.1 Run-time System Support  
      2.4.2 Distributed Processes  
   2.5 Hardware Transactional Memory  
      2.5.1 Intel TSX  

3 Preliminary Work  
   3.1 GHC's STM  
      3.1.1 The Effect API of STM  
      3.1.2 Run-time Support for STM  
   3.2 Hybrid TM for GHC  
      3.2.1 The Hybrid TM Design  
      3.2.2 Improving Performance  
      3.2.3 Results  
      3.2.4 Supporting retry and orElse.  

4 Thesis Work  
   4.1 Cross-project Goals  
   4.2 Projects  
      4.2.1 TM Projects
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.2.2 Runtime System Concurrency</td>
<td>35</td>
</tr>
<tr>
<td>4.3 Research Methods</td>
<td>36</td>
</tr>
<tr>
<td>5 Timeline</td>
<td>37</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

The Haskell programming language has proven to be a useful platform for research in parallel and concurrent programming. It is also a language that, with its leading compiler GHC and various backends, produces high performance programs. At times, however, this performance only comes by using low-level primitives rather than idiomatic functional programming techniques. This thesis explores the run-time costs of highly composable interfaces for concurrent programming and what changes can be made to improve performance without sacrificing high-level properties expected in Haskell programs. Indeed, we expect to find performance benefits from the high-level expression of concurrent programs. High performance run-time system support for concurrent programming involves well crafted synchronization, memory use, and allocation. Currently GHC’s run-time system works with a few basic synchronization primitives that incur significant invocation overhead.

We begin this proposal in Chapter 2 with an overview of Haskell programming with an eye toward concurrent and parallel programming. A description of GHC’s run-time system and execution will be helpful for the descriptions of various APIs and frameworks that follow. We highlight the importance of several properties of Haskell for this thesis. In particular it will be helpful to understand the way that Haskell controls effects and allows pure functional code to freely intermix with other code. If the reader is already familiar with Haskell much of this chapter can be skimmed. Note that at the end of the chapter in Section 2.5.1 there is an overview of hardware transactional memory that will be useful for the subsequent chapter.

Our preliminary work has focused on using hardware transactional memory to boost the performance of GHC’s software transactional memory implementation. Along the way we have discovered several aspects of GHC’s run-time system that impede performance. Haskell’s transactional memory also supports some features not supported in most TM systems. These features are a challenge to support with hardware transactional memory. In Chapter 3 we will explore the preliminary work and detail the next steps toward completing that effort.

The work that we seek to do, our approach to this research, and motivating questions are in Chapter 4. This is followed by a timeline for accomplishing the work.
Chapter 2

Background

We will start this chapter by saying that the world that Haskell lives in is quite different from the world in which most programming languages live. While there are many ways one could approach bridging the difference between these worlds, we have taken an approach that focuses on Haskell’s relationship to effects. An aptly timed article by Erik Meijer, who has been working to bridge these worlds for a long time, gets to the heart of Haskell’s approach by describing it as additive, rather than subtractive [24]. That is, Haskell starts with a non-effectful core of computation in pure functions and adds effects while explicitly denoting the effect in the type. The subtractive approach starts with all possible effects and labels restrictions in the type. We chose to spend a significant amount of time in this chapter building up this world, which we find to be a compelling world in the face of concurrent and parallel programming. Side-effects are a notorious difficulty for programming in that setting and Haskell offers a different approach as we will see.

Haskell offers a rich set of APIs for expressing deterministic parallel computations. Additionally there are APIs for handling concurrent, asynchronous, and distributed operations. Many of these are defined on top of some minimal parallel and concurrent APIs with little specialized run-time support. In Section 2.1 we will begin with a description of functional programming in Haskell and some of the properties and guarantees of its type system. This is followed in Section 2.2 by an overview of the way effectful Haskell programs are written. Section 2.3 details the low-level implementation details of executing pure Haskell programs. And Section 2.4 will look at some of the details of multicores execution and support for concurrent and parallel programming primitives. Finally in Section 2.5.1, we give an overview of hardware transactional memory as implemented by Intel’s Transactional Synchronization Extensions (TSX).

2.1 Introduction to Functional Programming

Statically typed functional programming languages are known for their expressiveness, correctness, and safety. Haskell is also known for its composability, having the meaning of programs and expressions follow directly from the meaning of the parts that make up these programs and expressions. The language does not, however, achieve composable performance, though it some cases it is able to recover algorithmic performance gains under composition through lazy evaluation. Reasoning about performance is difficult in Haskell, but thanks to many years of effort, the Glasgow Haskell Compiler (GHC) often achieves excellent performance in both single-threaded and multi-threaded execution. There are still many gaps to fill and this proposal looks to fill some of the gaps already
identified as well as find more opportunities for improvement of existing parallel and concurrent programming constructs. In particular we are looking to improve performance while maintaining the benefits of the Haskell language.

We will start with a high-level view of the properties of Haskell programs that we seek to maintain while improving performance. This will also point toward statically enforced properties of Haskell programs that can be exploited for performance gains.

2.1.1 Purity

There are many ways to define purity and these definitions lead to endless debates over small details. For our purposes a good intuition of what is meant by purity in Haskell is that values do not have side-effects. This is true of many other languages as well. For instance in C, a numeric literal does not have any side-effects and a variable of type `int` does not have side-effects. What is different in Haskell is that the language treats more things as values. For instance, functions and actions are first-class values. Functions in Haskell are mathematical functions, not procedures like in a language like C. Actions are like programs or procedures which may have effects but the program itself can be handled as a value. Larger actions can be composed together by sequencing or binding, but actions cannot be executed directly. To execute actions, they must be composed with the action that is assigned to the program entry point.

Haskell does allow for non-terminating values. This is denoted by ⊥ (pronounced bottom) and can be viewed as an effect that is allowed in a pure value. Partial functions are also allowed and exceptions give a way to define a partial function. It is useful for exceptions to also be denoted with ⊥. A pure function is not allowed to determine if a value is ⊥ or not. That is, its behavior should be deterministic. A pure function can, however, choose to ignore an argument value and avoid non-termination associated with that value. We can classify functions by their handling of a non-terminating value as an argument into strict and non-strict. A function is strict if the following is true:

\[ f \perp = \perp \]

Out of practical considerations, Haskell also includes a combinator to force the sequencing of evaluation. This operator is called `seq` and its behavior is defined by two equations:

\[
\begin{align*}
\text{seq } \perp b &= \perp \\
\text{seq } a b &= b
\end{align*}
\]

The remarkable thing about this operator is that it is polymorphic. If we know the type of the fist argument we could write our own monomorphic strict function with the same behavior. A polymorphic `seq` removes the need to write such functions, but also removes the ability to prevent such a function from being written (by, for instance, not exposing the constructors of a type).

2.1.2 Functional Programming

A simple definition of functional programming is that functions are first-class values. Many languages have this functionality, but not every language expects that the majority of code written
will be in a functional style that exploits functions as first-class values. When functional pro-
gramming is combined with type-enforced purity and an expressive type system as in Haskell, the 
language enables equational reasoning and the ability to express many mathematical structures 
directly. This has most notably been seen with the introduction of monads as a way to express 
effectual computations [25, 35, 36] but also more recently there have been further development and 
connections with category theory that have regular use in the community [23].

2.1.3 Evaluation Order

Perhaps the most unintuitive aspect of Haskell programs is lazy evaluation. The evaluation of 
any pure value is delayed until the actual value is demanded by a case statement, pattern match, 
or strict function such as addition on Int values. When evaluation does occur, the result is 
memoized into the value’s heap object. All references to that value will no longer need to perform 
a computation when execution demands the value.

Lazy evaluation motivates the need for pure expressions that are deterministic and side-effect 
free. When laziness is used in settings with effects or non-determinism it becomes very difficult 
to reason about the ordering of those effects. It also becomes difficult for the compiler to perform 
useful optimizations such as common sub-expression elimination. Eliminating expressions may 
eliminate effects. This is not the case in a language free of effects.

2.1.4 Parametricity

 Highly generic code in a pure functional language has an interesting mathematical property. When 
no information is know about a generic parameter to a function there is nothing that a pure 
program can do with that value. Fully generic arguments are fully opaque. In most languages 
there is usually a way to probe such arguments for some information. For instance in C you have 
a memory address for a pointer. In C# you can ask for the type of any object or compute a hash 
of the object. Based on this probing a function can change its behavior. In Haskell this is not 
the case. Consider, for example the function with the type $a \rightarrow a$ which means it gets a single 
argument of some fully generic type $a$ and results in a value of the same type. There is only one 
terminating implementation for this function in Haskell, namely the identity function that results in 
the argument it was given. This sort of reasoning about the possible implementations of programs 
based on their types is called parametricity [34].

2.2 State and Effectful Computations

While Haskell has always had a strong notion of pure functional programming, it has taken a while 
to discover a notion of how effectful computations fit into this world. There are other approaches 
to the problem of managing effects and much more research remains to be done. The managing of 
state with monads as described below, however, is an effective and powerful way to write programs 
with effects while still holding true to pure functional programming.

2.2.1 State Monad

We can connect functional programming back to imperative style programming by directly con-
structing state transformers. State can be modeled by pure functions that act on a state parameter
resulting in a value and a new state. This is done in Haskell by first giving a name to functions that fit this form. In Figure 2.1 this is the `newtype` declaration that defines the type constructor `State` parameterized by the type of the state `s` and the value that a particular state transformer will return `a`. It also defines on the right-hand side of the equals a value constructor also named `State` that, when given a function that fits the form, will build an object\(^1\) that has a field `runState` that contains the value. The form a state transforming function must take is \(s \rightarrow (a, s)\) where the first `s` is the argument state, the arrow is the function type operator and the pair \((a, s)\) is the result type of a value of type `a` and a new state of type `s`.

We are particularly interested in Haskell with how to compose state transformers. Remember that the meaning of a Haskell program should follow from the meanings of its parts. We would like to understand state transformers in the same way. To this end, we define two operators that form different compositions of these transformers. The first is the `(>>)` operator which can be called the “and then” operator. It is much like the semicolon found in a language like Java or C. In C each statement can be thought of as transforming the state of the machine. In a code block each statement is separated by a semicolon and no information implicitly flows between statements other than the state of the machine. The definition of `(>>)`) given in Figure 2.1 takes two state transformers \(f\) and \(g\) on the left-hand side of the equals and builds a new state transformer on the right-hand side. This new transformer evaluates \(f\) with the state parameter `s` (introduced with a lambda abstraction given by a slash “\("\). The result of evaluating \(f\) is bound to the values `a` and `s'` in a let expression. The state value `s'` is then used as the argument to \(g\). The result of the newly built composition of state transformers is the result of \(g\). We can see too that there might be a more general form of composition because we are ignoring part of the result of \(f\). We do nothing with the `a`. This is like writing an assignment in C where we are ignoring the “value” of the assignment. We can also write an assignment, inside a while condition such as:

\[
\text{while } (x = x + 1) \{ \ldots \}
\]

The assignment in the condition both changes the state of the machine and has a value that is checked by the while. A composition that takes into account the value as well as the state is the bind operator `(>>=)`. It takes a state transformer and a function that can produce a state transformer as arguments. This function will take the result of type `a` from the evaluation of the first transformer and produce the transformer that we will then feed the state `s'`. A subtle but very important point about the second argument to bind is that it is a function in the pure functional sense. It will always produce the same state transformer given the same argument and will not produce any effects. The state transformer that it returns, however, can be run to produce its effects. In fact any type built out of the function arrow `(->)` is a pure function.

In Figure 2.2 we can see some examples of building state transformers, composing them with bind and “and then”, and evaluating the composition by extracting the state transformer function with the field accessor `runState` and giving an initial state as an argument to that function. We first build to basic state transformers \(f\) and \(g\). These simply result in strings and result in a new state with an incremented value. These are combined with `(>>)`) and given the initial state zero. We see the result has the value from \(g\) and the state incremented by both \(f\) and \(g\).

We can also make a function that produces a state transformer given a string. In our example, \(g'\) takes a string and concatenates “ world”. When composed with \(f\) via bind we get a result that

\(^1\)Because this is a `newtype` declaration, this object will only exist as a compile-time way of distinguishing `State` values from values of type `s -> (a, s)`. Both will have the same run-time representation.
newtype State s a = State { runState :: s -> (a, s) }

instance Monad (State s) where
  (>>) :: State s a -> State s b -> State s b
  (>>) (State f) (State g)
    = State (\s -> let (a, s') = f s in g s')

  (>>=) :: State s a -> (a -> State s b) -> State s b
  (>>=) (State f) stateProducer
    = State (\s -> let (a, s') = f s
                  State g = stateProducer a
                           in g s')

  return :: a -> State s a
  return a = State (\s -> (a,s))

Figure 2.1: State Monad definition.

f = State (\s -> ("hello", s + 1))
g = State (\s -> ("world", s + 1))
g' a = State (\s -> (a ++ "world", s + 1))

ghci> runState (f >> g) 0
("world",2)
ghci> runState (f >>= g') 0
("hello",2)

Figure 2.2: State transformer examples.

has the concatenation of values and the same incrementing of states.

Given this formulation of state transformers we can build some generic transformers that capture
the sort of effects that are inherent to state transformers. Effects here mean the changes to the
state value of the function that distinguish a state transformer from a function from state to a
value: s -> a. The API is given in Figure 2.3. The get transformer simply returns the entire
state as a value and the put function takes a state and makes a transformer that will replace
the state with the given one. The resulting value of put is the unit value () that is much like void
in C. It should be clear that we can use these two primitive operations to build up any particular
manipulation of state that we could care to make. For instance, if our state is a table of values
indexed by names that represent all the variables in an imperative computation, we can use get to
get access to the table, then make a new table with a variable modified to a new value. That new
table can then replace the old one with a “call” to put. We can capture these fine-grained effects
with state transformers for any particular setting that we may want.

Haskell provide special syntax called “do”-notation for this sort of programming. It applies to
monad’s in general, but in our context we can limit ourselves to only talking about state transform-
ers. This notation allows us to write what look like statement with names being bound to values on
the left hand side of left arrows <- and expressions on the right hand side. These can be thought of
as effectful expressions and we are binding the resulting value while implicitly moving the state to
the next statement. We can include any pure computation in the expressions on the right hand side.
get :: State s s
get = State (\s -> (s,s))

put :: s -> State s ()
put s = State (\_ -> ((),s))

Figure 2.3: Effect API for the State monad

complex = do
    a <- f
    b <- g
    s <- get
    put (s + 40)
    return (a ++ "␣" ++ b)

ghci> runState complex 0
("hello␣world",42)

Figure 2.4: State transformer built with “do”-notation.

and any transformer who’s value can be ignored can be a top level statement without a binding. An example that builds up a more complicated state transformer can be seen in Figure 2.4. In that example complex is a state transformer defined by the composition of state transformers.

2.2.2 IO Monad

Let us now consider the other end of the effect spectrum. Haskell’s IO monad can be thought of as a state monad where the state being passed around is the entire state of the machine we are running on. Conceptually we get access to all the capabilities of the machine and we simply manipulate state to perform the desired effects. But we do not have access to the entire state of the machine in a pure non-destructive form! We can keep the Haskell interface to state but in the run-time system perform effects directly. We then provide an API to the effects that the IO monad provides. For instance, mutable memory references are given by the IORef API in Figure 2.5. We will use the term “IO action” for what we have been calling “state transformer” in the context of the state monad.

In the context of allowing any effect we can no longer provide the get and put API as we

newIORef :: a -> IO (IORef a)
readIORef :: IORef a -> IO a
writeIORef :: IORef a -> a -> IO ()

openFile :: FilePath -> IOMode -> IO Handle
hGetChar :: Handle -> IO Char
hPutChar :: Handle -> Char -> IO ()

forkIO :: IO () -> IO ThreadId

Figure 2.5: Some effectful API’s in the IO Monad.
```haskell
data ST s a

runST :: (forall s. ST s a) -> a

newSTRef :: a -> ST s (STRef s a)
readSTRef :: STRef s a -> ST s a
writeSTRef :: STRef s a -> a -> ST s ()
```

Figure 2.6: The ST monad and its effects.

```haskell
v :: STRef s Int
v = runST (newSTRef 1)

update :: ()
update x = runST (writeSTRef v x)

main = do
  forkIO (print (update 1))
  forkIO (print (update 2)) -- Data race!
  print (runST (readSTRef v))

-- Compile error on the definition of 'v':
-- cannot unify s' with s:
-- runST :: (forall. s' ST s' (STRef s Int))
--        -> STRef s Int
```

Figure 2.7: Without the higher-rank type in runST this code would compile. With the higher-rank type, state cannot “leak” from an ST action.

cannot meaningfully represent the entire state of the machine. We also cannot perform an effectful computation in the pure world of Haskell functions. This means we cannot have a function with the type `IO a -> a` where we perform an IO action and then look at its result. This type does not capture that the state of the world is changing. Its resulting value is likely to be non-deterministic as well! GHC is going to compile under the assumption that pure code is deterministic and has no side-effects. There are certainly situations where we can locally work with mutable state and have a deterministic result. We can capture this sort of private mutation and work with the result as a pure value as shown by Launchbury and Peyton Jones [17] with the ST monad seen in Figure 2.6.

### 2.2.3 ST Monad

With ST can use mutable references and have the implementation be actual mutation, but be assured that these references themselves will remain private to the particular thread that provided the initial state for the computation. This is controlled with a higher-rank type for runST. Its definition says that the ST action must work for all possible input states. This means that we cannot write computations that look directly at the state and we cannot have any result values whose type includes the state type. Figure 2.7 shows an example of a data race that is prevented at compile-time by the ST monad.
par :: a -> b -> b
pseq :: a -> b -> b

Figure 2.8: The API for deterministic pure parallel computations in Haskell.

parMap f [] = []
parMap f (x:xs) = y 'par' (ys 'pseq' y:ys)
    where
    y = f x
    ys = parMap f xs

Figure 2.9: Apply a function to all the values of a list in parallel.

2.2.4 Concurrent and Parallel Haskell

What we have looked at so far has mostly been in the context of a single thread of execution. We have seen that complex computations, including effectful ones, are built out of simple APIs and pure functional expressions. The underlying execution can match the systems destructive mutation while still preserving the semantics of the expressed Haskell program. We will use the same techniques to build up frameworks for controlling and expressing concurrent effects and parallel computations.

The ideas of Multicore Haskell are developed in two papers [21, 19] that introduce two primitive operations for parallel computation seen in Figure 2.8.

In this API lazy pure values are treated like futures. The par combinator will place the evaluation of the first parameter into a task queue called the “spark pool”. These evaluations are called sparks and typically the first parameter will occur somewhere within the second parameter. While before we noticed that the evaluation of pure expressions can be delayed or moved to where the value is demanded, we note here that par allows us to move this evaluation to a concurrent thread of execution. The second combinator pseq enforces an ordering to the evaluation of two parameters. The first must be evaluated before the second is attempted. This is different from seq in that seq is defined only by its strictness and gives no guarantees about its implementation. This flexibility allows GHC to do interesting code improvements and not be bound to a particular ordering. With these two combinators we can build an API for parallel computation over standard data structures. For instance a parallel application of a function to a list can be written as given in Figure 2.9. Notice that pseq must be used to ensure that the spawning of work is not driven by the lazy evaluation of the result, but instead is forced to happen for the whole list when the first constructor the result list is demanded.

This interface can be somewhat fragile, with regard to performance, to use in practice. On top of these primitives and others, however, more robust frameworks have been built that have much more predictable parallel performance.

2.2.5 The Effect Zoo

We can view the APIs that Haskell gives to parallel and concurrent programming as limited domains with refined APIs for effects and common pure evaluation and composition through monadic bind. After compilation the resulting code is indistinguishable from code written in the IO monad. Some
particular APIs that we will discuss include the following:

- **IORef**’s giving shared memory in the IO monad.
- **TVar**’s giving transactional memory references in the STM monad.
- **MVar**’s giving a shared mutable cell like a single element concurrent queue in the IO monad.
- **IVar**’s giving a mutable reference that can only be written once in the Par monad.
- **LVar**’s giving a lattice of mutable values that can only be written to by writes that increase the value monotonically. **LVars** live in the LVish Par monad.

In this thesis proposal we are concerned with supporting and refining these APIs to have better performance while still maintaining their restricted domains and flexible interface. We will also be looking for ways to exploit the high-level domain specific information captured by restrictive APIs. For example, we could track at compile time if a transaction is known to be read-only. This information could inform further transforms. There may be other effects that can be included alongside the minimal necessary effects that capture the desired behavior. For instance, the outcome of a series of STM transactions may be non-deterministic and dependent on the particular schedule for the threads involved. Given this non-determinism it may make sense to have a more relaxed method of distributing random numbers among threads that avoids the overhead of high quality splittable random number schemes [4].

### 2.3 Single-threaded Execution

The roots of GHC’s execution model for Haskell are detailed in the paper “Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine” [27]. Under this model Haskell is compiled to an explicitly typed intermediate language called **Core**. This language closely matches the underlying type theory for GHC Haskell with all of its extensions, System $F^\uparrow_C$ [41]. Many program transformations (both high and low level) are applied at to this intermediate representation [37]. Having a typed intermediate language has the tremendous safety benefits, helping check that transformations preserve program semantics.

For code generation, the Core program is transformed into another intermediate language called STG (taking its name from the Spineless Tagless G-machine). This language is very similar to Core, but with additional annotations that facilitate translating to machine code. STG has the nice properties of having a denotational meaning as well as clear operational semantics. The operational semantics of STG code can be informally thought of where let bindings are allocation and case expressions force evaluation. Expressions in STG are built from let bindings, case expressions, lambda abstractions, function application, and data type constructor applications. Any function arguments that are not a name or literal are given a let binding before the application. This can be seen as allocating a placeholder for the argument value.

The free variables of any lambda form are also identified explicitly in STG. Let bindings additionally indicate whether the name bound is an “update” closure that will memoize any eventual evaluation. Finally the language supports unboxed values by supporting “primitive” values and operations. Machine supported operations need not be built in (in the sense of built into the STG language) but instead can be primitive. Primitive values and operations are indicated by a trailing “#”.
Let's consider a very simple example from the STG paper:

\[(f \ x) + y\]

This Haskell expression will be turned into the following STG:

```haskell
case (f x) of  
  I# x# ->  
  case y of  
    I# y# -> I# (x# +# y#)
```

When execution reaches the first case expression it must evaluate the application \(f \ x\) to produce a value to bind to the pattern match. Control flow must enter the function application and when it is complete continue on to the evaluation of \(y\). Note that \(y\) must still be evaluated as a lazy value. This form is naturally amenable to a stack based implementation as the active frame is the environment needed for the continuation consisting of a branch of the case expression.

Another important feature for executing Haskell code is partial application of functions. STG can express partial application without special consideration though code generation must take this into account and handle all the cases. With polymorphic code, it may not be known to a particular function until run-time if an application will have all arguments (fully saturated) or only some of its arguments. It may also be the case that it has extra arguments that are to be applied to the result. GHC can specialize polymorphic functions to reduce the overhead of generic application, but a generic application method is always available.

There have been several incremental developments to single threaded execution since the original STG paper. One of these is pointer tagging to distinguish fully evaluated values and avoid run-time calling and update overhead. Additional bits of pointer tags are also used to specify a particular constructor form from a sum type that an evaluated value takes. This allows direct execution of the appropriate continuation in a case statement when applicable. These mechanisms are allowed to be imprecise in that pointers are allowed to falsely indicate that a pointer is unevaluated, but they are not allowed to incorrectly indicate that a value is evaluated when it is not, or indicate an incorrect constructor. As a simple example, Haskell uses a sum type (tagged union) for its \texttt{Bool} type defined as:

```haskell
data Bool = False | True

data Record = MkRecord { field1 :: Bool, ... }
```

As a non-strict field, \texttt{field1} will be subject to lazy evaluation. This may be the intent of the programmer and the boolean field may be hiding some much more complicated and costly computation. But this also means that once the value is evaluated we will continue to pay for the indirection of having a reference to a heap object to simply represent a single bit value. In practice, however, pointer tagging will fully distinguish \texttt{True} and \texttt{False} in the pointer stored directly in the \texttt{Record} object so we can get the benefits of lazy evaluation and avoid the indirection.

Another development was to switch from a “push/enter” style evaluation to “eval/apply” [20]. With push/enter, functions statically know their arity and determine at run-time the number of arguments it has been passed. This is similar to C’s “varargs” calling convention. With eval/apply, callers statically know the arguments and at run-time example the function closure to determine its arity. This evaluation methods is, of course, only concerned with the cases where the arity is not statically known at the call site. For eval/apply GHC has an evaluation stack where each stack frame represents a continuation. These continuations come in several forms. An update continuation has a return address and a pointer to the heap object to update. When executed it
simply performs the update, pops the update frame off the stack and jumps to the return address. A case continuation represents the choice of case branches. It contains a return address and the free variables of the case alternatives. This is much like a stack allocated function closure. Finally a call continuation has a return address and arguments. Execution will reach a call continuation when it finishes evaluating a function value. Each of these forms are specialized to the number of arguments to be applied along with a generic handler for larger numbers of arguments. Unboxed arguments further complicate this. The common cases as empirically determined are all specialized.

GHC employs a uniform representation for heap-allocated objects regardless of their current state of evaluation. Each object is a code pointer along with zero or more fields giving the free variables of the code. With a uniform representation there is no need for tags indicating what form an object takes, hence the Tagless part of the STG name. Figure 2.10 shows the layout of a heap object.

Another important part of execution are “blackholes”. When a value needs to be evaluated an update frame is pushed on the stack so that we can memoize the result. This model would be sufficient, but is susceptible to a subtle but important space leak [16]. An extra step is performed instead where the value to be updated is overwritten with a special “blackhole” heap object. The update frame will then overwrite the blackhole back to a value. In the intervening time, if a garbage collection pass is needed, intermediate values that are not referenced by the executing code or other heap objects, but were referenced by the heap object under evaluation will be collected. Without the blackhole they would still be reachable. All the values necessary to the computation are still reachable from the execution stack. Blackholes will be an important topic we will revisit in Section 2.4.

One final topic for single-threaded execution is the implementation of the IO monad. IO action execution requires sequential execution of IO primitive operations. As we saw in Section 2.2, this can be modeled with state transformers and with an implementation that actually performs effects in place. GHC’s implementation uses a type called RealWorld for the phantom type. A value for RealWorld can never be generated but the dependencies on this state enforce the sequential execution via case expressions that “evaluate” the phantom value. Clearly GHC needs to be good at
eliding these useless evaluations to have any reasonable performance. Having a generic mechanism
to represent this sequential enforcement is extremely useful for supporting other computations that
require sequential ordering. Indeed the ST monad for mutable access to private data and the STM
monad for atomic shared memory both rely on the same mechanism for ordering of effects. In the
end the boundaries between IO code and pure functional code are erased.

2.4 Multicore Execution

GHC Haskell first included support for concurrency [28] allowing user threads executed on a single
core and then later introduced multicore execution [21]. In the multicore run-time GHC uses the
name Haskell Execution Context (HEC) for each OS thread used by the system. Each HEC will do
work by running Haskell threads which have a heap allocated growable stack. The design assumes
that these threads must have very low costs for context switching and a small memory footprint.
In this section we will look at the run-time system support for multicore execution and the low
level APIs that it gives to users for building concurrent and parallel programs.

2.4.1 Run-time System Support

As described in Section 2.2.4 Haskell provides the \texttt{par} and \texttt{pseq} primitives to “spark” evaluation
tasks. This is implemented by having a work stealing dequeue on each HEC. Communication
in concurrent execution is performed through explicit mutable references. The garbage collector
must consider these references differently than the immutable default heap objects. Currently
Haskell’s run-time system provides few primitive operations for synchronization and supporting
parallel execution. We will now describe some of the important features of the run-time system
that have an effect on concurrent and parallel programming.

Threading

Haskell threads are implemented with very small overhead and are intended to be used as language
level units of work. Applications will often spawn thousands of Haskell threads. To support this
each thread is given a small heap allocated stack (this is not an execution stack, but a continuation
stack) that can grow as needed. The state of the thread and a pointer to its stack is stored in a
heap allocated Thread State Object (TSO). Each HEC has a run queue of threads ready to run,
a message queue containing messages from other HECs, and a spark pool containing values ready
for parallel evaluation. Execution of a HEC proceeds by servicing in priority order the message
queue, run queue, spark pool, and finally looking for threads that have become runnable in a global
blackhole pool. Lower priority actions like the spark pool are only serviced if there is no higher
priority work such as a message in the message queue or thread in the run queue.

We saw earlier that in single threaded execution, when execution reaches an unevaluated value
it overwrites the object with a blackhole. This will in turn be updated to the final value. If before
that final update execution encounters a blackhole and needs the actual value the thread will block
and be added to a global blackhole pool. HECs will eventually find threads in that pool that are
now ready to run as the blackhole they are referring to is updated to a final value. There are several
interesting trade-offs in the design of blackholes detailed in [21]. Part of our work in this thesis
will be to investigate the consequences and costs of this arrangement. The Intel Haskell Research
Compiler [26] notably does not have this design and uses an OS thread for each Haskell thread. We suspect that many idiomatic Haskell applications simply cannot run with this design.

**IORef**

An IORef is a mutable cell that can be read or written to by an IO action. It can also be atomically modified with `atomicModifyIORef` which is implemented with an atomic compare and swap. An important use case for this primitive operation is when the user couples its use with lazy evaluation. While the function always modifies the contents of the IORef we can think of the work that needs to be done as to simply put a placeholder for the computation. The actual computation need not be performed before the winner of a race to perform the swap is determined. The danger of this is that the value held by the IORef eventually will be a long chain of deferred computations. At some point, someone must force the computation to occur. We will be seeking in our thesis work to better understand the use patterns of `atomicModifyIORef` to determine if there are ways to improve the implementation and if there are ways to use this same pattern effectively in other settings such as STM.

**MVar**

MVars are low level primitives that provide a blocking queue with a single cell. This is used to build locks, unbounded queues, and other concurrent data structures. It is also used internally in the run-time system for synchronization.

MVars are implemented with a FIFO queue of waiting threads. The particular structures used are often a significant bottleneck. In some cases MVars built using STM (called TMVars) outperform MVars at the cost of sacrificing ordering of servicing waiting threads. Improving MVar performance could be beneficial to many Haskell programs as well as the run-time system itself which relies on MVars in many key places including the asynchronous IO manager.

The run-time system has an IO manager that must handle OS asynchronous IO requests and responses, mapping responses back to a blocked Haskell thread. Non-trivial scheduling issues arise in the design and programs like web servers benefit greatly from improvements to the concurrent data structures used by the IO manager. We will investigate the impact of changes to this structure and look for potential uses of hardware transactions to improve the performance of this performance critical part of the run-time system.

**STM**

Our exiting work has shown that GHC’s existing STM can be improved by extending it to use hardware transactional memory. GHC’s existing STM implementation was introduced by Harris et al. in *Composable Memory Transactions* [13]. The implementation has grown as the run-time system gained features including multicore support. We will talk about STM in more depth in Section 3.1. There are many opportunities to improve STM performance at all levels.

**par and pseq**

The work-stealing deque that holds computations for parallel evaluation (sparks) from calls to par is modeled after the work of Chase and Lev. The pseq combinator does not need any run-time support and just affects code generation. Some work must be done to efficiently execute sparks
and avoid the overhead of thread creation for each spark. We will look further into the memory implications of sparks and the scheduling of their execution.

**Par Monad**

The Par monad gives a model for writing deterministic parallel programming by explicitly expressing dependencies between computations. In the Par monad values are distributed by writes to IVars. Each IVar can only be written once. Reads of IVars are blocking if the IVar is empty. IVars are defined completely in Haskell and built out of an IORef. Values put into IVars are either fully evaluated or evaluated in weak head normal form.

There is no explicit run-time support for the Par monad. It is implemented with IVars as IORefs and uses of atomicModifyIORef. Work is scheduled by pulling ready work from a queue and enqueuing tasks that block waiting for a value to fill an IVar.

**Strategies**

The goal of the strategies work is to separate algorithms from particular applications of par and pseq. Parallel evaluation patterns are abstracted into reusable Strategies. The implementation is purely uses of par, pseq and various forms of evaluation forcing. Some work has been done to redesign the GHC run-time to allow for library-level custom schedulers for threads [32, 18]. The goal with such a design is to be able to write much more effective “strategies” for parallel computation informed by high-level information. The work done in this direction so far assumes an efficient STM implementation for users to safely and easily write scheduling operations. Our improvements to STM can potentially be of benefit here. In such a performance critical setting we will look to be more restrictive and potentially avoid supporting some STM features such as retry and orElse.

**LVars**

Like the Par monad with IVars except allowing for multiple writes as long as the writes are monotonic and no information is lost in subsequent writes. Similar to IVars, LVars are implemented using atomicModifyIORef. The most recent work on LVars to appear at PLDI 2014 builds an extensive collection of composable effect domains to allow for the expression of complicated deterministic parallel programs while allowing for disjoint parallel in-place mutation, deadlock resolution, cancellation, memoization, and other effects. We see the potential to include significant run-time support to aid the performance of these effect domains.

**Data Parallel Haskell**

Data Parallel Haskell (DPH) introduces parallel arrays and operations on these as well as language integration building on existing list comprehension syntax. In particular DPH is an experiment in exploring ways to efficiently handle nested data parallelism and better exploit both instruction-level parallelism and multi-core parallelism. We hope that in our work we may discover insights that help the run-time support for these computations.
2.4.2 Distributed Processes

The distributed process library extends Haskell to support Erlang-like processes that communicate with message passing. These processes can be distributed or together on a single shared memory machine. There may be potential to better express isolation and control sharing in this setting when executing on machines with non-uniform memory access costs.

2.5 Hardware Transactional Memory

In the final section of this chapter we will briefly look at hardware transactional memory. Hardware transactional memory systems can vary greatly in design, but they seek to take advantage of opportunities to detect conflicts efficiently and at a level that is not accessible to software. This generally leads to integration at some cache level in the cache coherence mechanisms.

Additional benefits might be available when restrictions on what executions are allowed inside a hardware transaction free the hardware from some costly constraints. For example, if a checkpoint of architectural state is required at the beginning of a transaction and exceptions are not allowed to be observed inside a transaction, then the starting checkpoint may be the only restore point needed for the duration of the transaction [2].

2.5.1 Intel TSX

The 4th generation core architecture (Haswell) from Intel is their first processor family to include hardware transactional memory support named Transactional Synchronization Extensions (TSX) [15]. TSX supports two forms of transactions: Hardware Lock Elision (HLE) and Restricted Transactional Memory (RTM). Both use the same underlying mechanism and they are compatible with each other, but HLE does include some additional support to ensure backward compatibility at the cost of some flexibility. The implementation leverages the cache coherence protocol and is confined to working in the L1 data cache. If writes exceed the space in the L1 or if a write must be evicted due to associativity or running out of room, the transaction will be aborted. The total number of reads in a transaction, however, have been observed to exceed the space available in the L1.

Hardware Lock Elision

Hardware Lock Elision is a backward-compatible mechanism for marking lock acquire and release instructions specifying that the hardware should speculatively execute these critical sections. On entering a critical section under speculative execution the protecting lock value is read but not written. While in speculation, memory reads and writes are tracked locally and conflicting reads or writes from other processors are detected. If a conflict is detected the speculation is aborted and the speculative changes to memory will be discarded. After some number of failed attempts HLE will fall back to actually writing to the lock variable and executing non-speculatively.

To indicate a lock that should use HLE the new prefix \texttt{XACQUIRE} is added to the instruction that writes to the lock. The write to release the lock must have the prefix \texttt{XRELEASE} added and must write back the value that was in the lock prior to the \texttt{XACQUIRE} prefixed instruction. Even though the lock is not written in a speculative execution reads of the lock variable inside the executing HLE transaction show the locked value rather than the unlocked state seen from code executing.
outside the transaction. This ensures backward compatibility for code that checks that a lock is held. From outside the transaction it appears that the lock is free. It is worth noting, however, that programs with certain data races that were not previously observable may become practical problematic executions with HLE.

**Restricted Transactional Memory**

TSX also provides a more flexible interface that allows programs to define transactions directly with a `XBEGIN` and `XEND` instructions. These regions mark a region of transactional execution and do not necessarily have a lock address associated with them like in HLE. When a transaction aborts, the writes from the transaction are discarded and execution resumes at the fallback address given to `XBEGIN`. When no conflicts occur and the transaction successfully commits, all the writes from the transaction are made visible to other processors with the same memory ordering semantics as a `LOCK` prefixed instruction. From the outside, the transactions appear to execute atomically. An additional new instruction `XTEST` gives the programmer a way to test if the current execution is transactional.

In both RTM and HLE some instructions will always cause a transaction to abort. Some conditions outside of the programmer’s control will also cause transactions to abort. For example, x87 floating point instructions are not allowed (this is not surprising given the implementation details of x87). Unfortunately, the C calling convention for 32-bit x86 involves x87 instructions. The other instruction restrictions are less likely to be an issue. Hardware interrupts will cause a running transaction to abort before control is transferred to the handling code.

The transactions in TSX work with a cache-line granularity and is susceptible to false sharing. Transactions are not unbounded and are limited to the “best effort” of the hardware. The uncommitted state of the transaction must fit within the cache implying that access patterns causing uncommitted data to be evicted must cause an abort of the transaction. This is generally unpredictable and uncontrollable and indeed some transactions will never be able to commit. Finally, writes of the existing value already held in a memory location may cause an unnecessary abort. It may be worthwhile to check that a write is an actual change of value before performing it. If a check is performed, it can prevent a read from becoming a write when it does not need to be one.

**HTM Challenges**

Because hardware transactions may fail, we must have some sort of software fallback to ensure progress. In the simplest case this can just be a lock held for the duration of the transaction. This is the automatic fallback of HLE. For RTM we must ensure that we read the lock variable for a lock based fallback to include that value in the read set of the transaction. When we read that lock will be a topic we visit in Section 3.2.1.

There is some overhead to starting and committing hardware transactions but the cost of tracking accesses in software will be much higher than the cost of the same tracking in hardware. When considering HTM as a possible form of transaction in a larger Hybrid TM implementation that includes both STM and HTM, we want the typical transaction to successfully commit using the hardware. When a transaction cannot commit in hardware we must gracefully fallback to a software code path. It may not be the case that both software and hardware transactions can run together. Previous hybrid systems have appealed to the availability of non-transactional reads to coordinate phases of software and hardware transactions to avoid false conflicts [5]. TSX has no
facility for non-transactional reads. Due to this we must tolerate some level of false conflict due to unfortunate timing. The danger with these conflicts is that hardware transactions aborting due to a transaction in the fallback path may then choose to take the fallback path causing subsequent hardware transactions to abort. This is particularly problematic because the software fallback is expected to be slower, meaning hardware transactions may require many repeated attempts to speculate before it will have a chance to succeed. This effect is called “avalanche behavior” [1] or the “lemming effect” [8].
Chapter 3

Preliminary Work

We start this chapter with a description of GHC’s STM implementation in Section 3.1. Much of
the details of the implementation were undocumented prior to our efforts to learn and describe the
system in the GHC commentary. In this process we were able to address some correctness issues in
the commit protocol and describe some issues with the semantics of the data invariants feature. Our
main contribution has been to build a preliminary hybrid transactional memory for GHC. This is
described in Section 3.2. In the process we have identified several issues that hinder performance.
Some solutions and more directions to research are described in Section 3.2.2. Finally, our hybrid
TM is not complete. In Section 3.2.4 we describe the steps we will take to complete support for
\texttt{retry} and \texttt{orElse} in our hybrid TM.

3.1 GHC’s STM

GHC’s software transactional memory interface is the preferred method of writing concurrent pro-
grams in Haskell. It includes some features not commonly found in other STM implementations and
at least some of its popularity comes from the ease in which the Haskell language allows compile-
time restriction of allowed effects in transactions. In this section we will see the features of GHC’s
STM and some details about its implementation and run-time support.

3.1.1 The Effect API of STM

Haskell’s transactional memory API lets users write transactions by composing STM actions which
are just like IO actions, but restricted to memory operations on \texttt{TVar}s as seen in Figure 3.1. All
transactional variables are explicit. A transaction is run by using the \texttt{atomically} function to
produce an IO action. The action given to \texttt{atomically} will be transactional with all of the various
actions on \texttt{TVar}s happening such that all succeed or all fail and no other transaction sees the
intermediate state.

There are two operations in the API that are a bit unusual for a transactional memory system.
The first is \texttt{retry} which will suspend the transaction until a change is made to one of the \texttt{TVar}s
that it read. Semantically \texttt{retry} is the same as simply aborting the transaction and starting again
from the beginning. The second is \texttt{orElse} which composes two STM actions so that it tries the
first action and if it results in executing \texttt{retry} it aborts the actions of the first and moves on to
the second transaction. These two operations give the ability to develop blocking and non-blocking
data TVar a

instance Monad STM where ...

newTVar :: a -> STM (TVar a)
readTVar :: TVar a -> STM a
writeTVar :: TVar a -> a -> STM ()

atomically :: STM a -> IO a

retry :: STM a
orElse :: STM a -> STM a -> STM a

Figure 3.1: The API for the effects of STM.

transactional APIs in a library. Given a blocking API, a user can use orElse to give a non-blocking fallback to the action that would have blocked.

Transactions are built like IO actions, composing pure expressions and reads and writes to TVars. A simple example in Figure 3.2 shows transactions transferring values from accounts built out of the composition of withdraw and deposit STM actions.

3.1.2 Run-time Support for STM

In its current form there are two implementations: a coarse-grain lock version which serializes the commit phase of transactions with a global lock and a fine grain lock version which has a lock for each TVar where these locks are only taken and held during the commit phase. The implementation has its roots in Frasier and Harris’ work [11, 12] but notably is not a lock-free implementation.

Data Structures

An executing transaction keeps a transactional record (TRec) of accesses performed during execution. The TRec entry contains a pointer to the particular TVar it references, the value seen when the TVar was first accessed in the transaction, and a new value that holds any writes to the variable.

Each TVar is a heap object with a pointer to the value it contains. In the fine-grain lock version of the implementation this pointer is used as the lock variable for each TVar. The TVar is locked when its value points to a TRec rather than some other heap object.

Execution

When a transaction begins a TRec is initialized and an ATOMICALLY_FRAME is added to the stack. Execution then enters the user code of the STM action. Each call to readTVar will search the TRec for an entry matching the given TVar. If it is found it will result in the value from the entry. Otherwise a new entry is added and the value found in the TVar is recorded. For fine-grain locking the implementation will spin if the TVar is locked. When a TVar is written the value of the write is recorded in the record. Once all the users code has run execution will reach the ATOMICALLY_FRAME pushed at the beginning. This will start the validation and commit of the transaction.

In the coarse-grain lock implementation the global lock will be acquired before validation and commit. For the fine-grain lock version, the process of validating will include acquiring locks for
```haskell
deposit :: TVar Int -> Int -> STM ()
deposit account value = do
  balance <- readTVar account
  writeTVar account (balance + value)

withdraw :: TVar Int -> Int -> STM ()
withdraw account value = do
  balance <- readTVar account
  writeTVar account (balance - value)

transfer :: TVar Int -> TVar Int -> Int -> STM ()
transfer a b value = do
  withdraw a value
  deposit b value

main = do
  a <- atomically (newTVar 100)
  b <- atomically (newTVar 100)
  c <- atomically (newTVar 0)
  forkIO (atomically (transfer a b 100))
  forkIO (atomically (transfer b c 100))

Figure 3.2: Running a transaction atomically. Three accounts are initialized then two threads run transactions concurrently. The end result must always be \( a = 0 \), \( b = 100 \), and \( c = 100 \).

all the `TVar`s that the transaction will write. Validation will succeed when reads of all the `TVar`'s values match the expected value seen when the variable was first accessed. In the fine-grain lock implementation this validation must happen after successfully locking the write `TVar`s.

If validation succeeds the transaction can commit by performing its writes. The coarse-grain lock version will release the global lock while the fine-grain lock version releases each `TVar` by writing the new value to each `TVar`. Execution then continues by handing the transaction’s STM action result to the next continuation on the execution stack. When a transaction fails to validate it will be aborted by simply throwing away the transactional record and starting transactional execution again.

Implementing retry.

The `retry` action is supported by keeping a watch queue on each `TVar` that contains threads that are waiting for that `TVar` to be written to. When a transaction commits it will also wake up all the threads waiting in the watch queues of any `TVar`s that it writes. When execution reaches a `retry` an exception is thrown which searches the execution stack for a handling frame. If it reaches the `ATOMALLY_FRAME` it will validate and acquire locks for all the `TVar`s in the record. For the transaction to make progress one of these values must change. The transaction will then add itself onto each `TVar` and release the locks and put the thread into a blocked state. Care must be taken to ensure that no intervening commits can happen that attempt to wake the thread before it is safely blocked.

When a thread blocked due to `retry` is woken it first performs a validation and if the validation succeeds it goes back to sleep as the state of its read set has not yet changed and execution will still result in a `retry`. If validation fails, it starts the transaction again from the beginning.

23
Implementing orElse.

When orElse is executed it will place a CATCH_RETRY_FRAME on the stack and jump to the code for the first alternative in the orElse. This action will run and if it reaches a retry the exception raised will search the stack and find the CATCH_RETRY_FRAME. At this point the effects of the failed alternative need to be rolled back. To do this, GHC’s TRecs are “nested” with a new level of nesting introduced by each orElse. The CATCH_RETRY_FRAME will then reject the writes from the top level TRec and pop off that layer. It does need to merge the reads into the next level of TRec, however, as we need an accurate read set for the whole transaction. The semantics of orElse are that the second alternative is taken only if the first does not succeed. These conditions are required to both hold atomically. Merging the reads ensures that validation will fail with a conflict when a change happens to reads only performed on the rejected first alternative.

3.2 Hybrid TM for GHC

Our work has focused so far on implementing a hybrid TM for GHC. We chose to work with the coarse-grain lock version of the implementation and use hardware transactions to run full Haskell transactions. As we will see in this section there are several challenges to using hardware transactions in Haskell. Hardware transactions are generally sensitive to memory use and as such require an implementation with minimal unnecessary memory accesses to be competitive. Some continuing challenges will be to find ways to reduce the number of memory accesses incurred by the extra indirection of lazy evaluation and to reduce or avoid the overhead of calling into a C library to perform the transactional memory operations. These preliminary results were presented at TRANSACT 2014 [39].

3.2.1 The Hybrid TM Design

With a single global lock, the coarse-grain lock implementation of GHC’s STM is a natural fit for HLE. Doing this allows commit conflicts to be detected in hardware, allowing independent transactions to commit concurrently. With this scheme we still need to build and maintain the transactional record and we still need to detect conflicts with that record. We avoid doing the computation of the transaction in a hardware transaction, but still must read and/or write all the transactional variables touched by the transaction. If we consider a transaction that has minimal overhead for the computational aspects of the transaction, this approach will have more work to do inside the hardware transaction, then running the transaction directly with TSX. This observation motivated us to pursue an all hardware code path for executing transactions. We do, however, have to overcome the semantic differences between this hardware code path and the software fallback. In particular we need to control what happens when a hardware transaction attempts to commit concurrently with a software transaction’s commit.

Reading the Software Lock in Hardware Transactions

In our coarse-grain lock implementation the writes of a software transaction are only atomic with respect to the global lock. If the hardware transaction ignores the global lock, it can see a partial write of a software transaction’s write set. The clear solution is to read the lock variable at the beginning of the hardware transaction. When any software transaction begins its commit, it will
immediately abort all hardware transactions. But these transactions may not actually conflict. A solution described in recent work by Matveev and Shavit [22] is to use lock elision on the coarse-grain lock of the software commit. This has the effect of making the writes of the software transaction appear atomic, removing the possibility of seeing inconsistent state in a hardware transaction. There will still be a fully software code path where the global lock is acquired and all hardware transactions will abort.

Matveev and Shavit propose reading the lock variable at the end of the hardware transaction and checking if the variable indicates that a software transaction is committing. If we happen to have bad timing, and we observe the lock as held, we must abort. If, however, the lock is free, we can commit. A non-conflicting fully software fallback can commit while hardware transactions are running in this scheme. In the Hybrid NOrec work of Dalessandro et al. [5] non-transactional reads were used to spin in the hardware transaction until the lock variable is again free before committing. This would allow the software transaction to fully commit before attempting to commit the hardware transaction. With TSX we do not have non-transactional reads, however, so we cannot employ this scheme. When reading the lock late we can still observe inconsistent state.

To see an execution where this can happen look at Figure 3.3. Here we see a software transaction A writing to variables y then x. A concurrent hardware transaction B reads x and y between A’s writes. The hardware transaction will then act on this inconsistent data. What is the worst thing that can happen? We are still going to observe either the lock before the XEND or the software transaction’s eventual write to x. Either of these events will abort the hardware transaction. Indeed, such transactions are referred to as “doomed” transactions. One unfortunate outcome of reading such inconsistent state would be to read the values of x and y compute an address for a jump or call. This jump could land on an XEND instruction, committing the hardware transaction in its inconsistent state!

We would like to retain the benefits of reading the lock variable late and narrowing the window where software transactions abort hardware transactions, but we need to know what conditions will lead to bad executions. In the original Haskell STM work [13] it is noted that Haskell’s STM can execute after observing inconsistent state (validation normally only happens immediately before commit). The only effects that the original authors were concerned with was when execution entered an infinite loop or when an exception was raised. To handle both these cases transactions are validated at appropriate times. For exceptions, validation happens before the exception is propagated and for infinite loops validation will happen with garbage collection and any doomed transaction is aborted. There are other possible executions that can lead to more disastrous effects. As far as we can tell, these executions will always involve reads from at least two TVar’s and values from those TVar’s need to be used together as an argument or arguments to a function with the prefix “unsafe”. For instance, Haskell’s array implementation includes an API for reading without a bounds check. This unsafeRead function could be given an array from one TVar and a size from another. This could quickly lead to a failed execution, jumping to evaluate a closure read from uninitialized memory off the end of the array.

Use of unsafe functions in STM is generally not done but there is a possibility of unintentionally using unsafe functions in code written to be polymorphic over the underlying monad. We hope to revisit this issue in our work developing this thesis and write a formal description of the conditions that ensure to good uses of unsafe. This may lead to guidelines for using unsafe functions or it may indicate that we should change the implementation to avoid acting on inconsistent data through sandboxing as described in [6].

25
Hardware Transaction Details

We can now describe our hybrid system. When atomically is executed we will first attempt several times to run the entire transaction in a hardware transaction. Execution enters the atomically# primitive operation written in Cmm and we call the C function stmStartTransaction. In here we call XBEGIN and return. When the primitive operations readTVar# and writeTVar# are executed they call into the C functions stmReadTVar and stmWriteTVar. As we will see in Section 3.2.2 this will be a source of poor performance for hardware transactions. In these functions we use XTEST to determine if we should update the transactional record or simply read and write directly to the TVar’s value. Finally execution reaches ATOMICALLY_FRAME which will call the C function stmCommitTransaction. In here we use XTEST and when in a hardware transaction, read the global lock, check that it is free, then call XEND.

We do not have support for retry and orElse in this scheme. When retry is encountered we simply abort the transaction and try again with the software fallback. Similarly an orElse that reaches a retry will fall back. If the first alternative succeeds we can proceed in hardware. Unfortunately with this scheme we have broken retry support. Remember that in the STM when a transaction commits all writes will also wake up threads in the watch queues of the written TVars. If we simply perform writes in hardware transactions we will fail to wake up threads that are blocked.

A simple solution to this that lets us begin to evaluate performance is to make retry in the STM simply actively abort and restart the transaction. This is contrary to the semantics of the original work and will perform quite poorly in many existing applications which rely on an efficient minimal wakeup mechanism using TVars as condition variables that trigger further computations when they are written.

3.2.2 Improving Performance

If implemented as described above, hardware transactions suffer an unacceptable amount of capacity aborts with transactions as small as accessing ten transactional variables. The solution to this problem was to avoid the overhead of calling into C functions when running hardware transactions. Haskell threads do not have a C call stack and making a call to C from Cmm is essentially a foreign function call. We did not track down the particular details of the problems here; it is likely that it was either touching a large amount of memory to establish a C stack or some unfortunate layout that made associativity aborts likely. We did, however, move the code for hardware transaction
reads and writes into an inline primitive operation written in Cmm. This change allowed us to successfully commit hardware transactions that accessed hundreds of TVars with some writes and transactions that read thousands of TVars. More work remains to further streamline both STM and HTM code paths. Additionally Haskell execution may benefit in general from determining the source of the overhead related to C calls. It may be the case that hardware transactions are particularly sensitive to the access patterns and general execution has little benefit.

An additional source of overhead can be seen in Figure 3.4 in the representation of a typical heap object, a TVar referencing that object, its watch queue, and the transactional record of a software transaction. The values referenced by a TVar must be heap objects and cannot be unboxed values that live in the TVar itself. In the example in the figure we can see that each red-black tree node must pay for an extra level of indirection through the explicit transactional variable. While the fine-grain lock implementation relies on this behavior an implementation could avoid the need quite easily. It also may be possible to unbox Tvars into other larger structures. There can be positive and negative aspects to either of these abilities. On the one hand there is less memory that needs to be accessed, but putting more TVars together could result in false conflicts in hardware transactions. Also, in a language where immutability is the default, mutable heap objects carry some overhead in the garbage collection. We suspect that we will be able to allow the user to express their intentions for unboxing, giving flexibility to explore these trade-offs.

Another feature that needs improvement is the transactional array TArray. It is implemented as an array of TVars rather then a more compact and efficient structure. We would like to extend support to at least include unboxed arrays with shared metadata. With HTM we would get conflict detection at the cache-line level while we could sacrifice the granularity for the STM. This would allow for use with interesting compact data structures and should help HTM performance by more compactly representing data.

There are several improvements to the performance of STM that we expect could be of great benefit to current users. A good starting point for investigating this will be to implement a version
of NOrec as the underlying STM implementation. This will also allow us to more directly measure the cost of the Haskell execution machinery versus a C++ implementation of NOrec. Of course the difficulty will be supporting `retry` and `orElse`. We will look more into that in shortly in Section 3.2.4.

### 3.2.3 Results

These results are from our TRANSACT 2014 paper [39].

Results were obtained on a 4-core, 8-thread 4th generation Intel Core processor (Haswell) system. Each data point is the average of several runs. Figure 3.5 shows the performance of non-conflicting read-only transactions. The benchmark first builds a red-black tree and then spawns four worker threads. Each thread executes 5,000 transactions, each of which performs 40 lookup operations on the tree. In Figure 3.6 we show the performance of a write-heavy workload, which replaces all the nodes in a red-black tree, in parallel, with new nodes, repeating the process 10 times.

In the read-only benchmark our hybrid TM performs better up to a size where the transactions start to rely on the fallback due to conflict aborts. We suspect that these conflicts stem from the metadata writes inherent in executing Haskell code. Performance of the hybrid system was dramatically worse before we altered the hardware transaction code path to avoid calling out to a foreign C function on every load and store.

The write-heavy benchmark shows our hybrid performing roughly 30% slower than coarse-grain locking or the HLE fallback. The default fine-grain locking performs very poorly; on larger trees it appears to live-lock, as the commit phases of software transactions cause mutual aborts.
3.2.4 Supporting retry and orElse.

We have explored the design for supporting retry and orElse but have not yet implemented this aspect of our hybrid TM. As described above, the support of these features requires coordination between both hardware and software transactions. First we will look at what approaches we will need to support waking blocked threads. Next we will see some options for handling a retry encountered while executing in a hardware transaction without falling back to software. After that we will look at some options for improving orElse with hardware transactions.

Waking Up Blocked Transactions

Fundamentally approaches that avoid active retry will need to record the read set of the transaction to know which changes to the system should trigger a new attempt. For now we will leave this to the software code path. Similarly all transactions that write will need to record a write set to know which changes have been made. If our software fallback will handle retry with the same wakeup efficiency as the original implementation we will need to do this tracking in hardware transactions. Our preliminary measurements indicate that the overhead for keeping this record is significant. While the original STM performs minimal wake up operations, we hope to get the benefit of few wake up operations while keeping a constant space transactional record. To do this we will implement a bloom filter to approximate the read set for a running software transaction. In hardware transactions we will track the write set in a similar way. We can then quickly compare the write approximation with the read approximations of blocked threads and know who to wake up. The approximation can yield false positives, but will not give a false negative that could lead to a missed wake up. We expect that there will be many tunable parameters to this setup and
having good benchmarks refined from existing programs will be a useful guide for making decisions as we go along.

**Handling retry.**

We would like to explore handling `retry` directly when it is encountered in a hardware transaction. This means that we need to track the read set, but also we need to keep a record of the read set while also rolling back the effects of the hardware transaction. If we invoke `XABORT` we only get 8-bits of information to send back to the handler. We could use a non-transactional write to store the read set before aborting, but unfortunately TSX does not support such writes. The cost of such writes is often high on systems that do support it, but in this particular case we can tolerate higher costs as long as it is cheaper then repeating the transaction as a software transaction. Our bloom filter idea does still fit well with this scheme as we can compress our bloom filter to 8-bits at the cost of more false positives.

Another approach to using our 8-bits would be to have them indicate some particular set of `TVar`s that have been profiled to be common wakeup sources. Programs that employ some handful of `TVar`s as condition variables could benefit from such an approach. It may also be common that transactions that execute `retry` are read-only. When this is the case we can commit the hardware transaction with `XEND`, retaining the larger read set approximation.

Key to all of these approaches will be determining if the cost of tracking the read set in hardware transactions out weights the benefit of not falling back to software. We will need to rely on good benchmarks to indicate the right balance.

**Handling `orElse`**

Handling `orElse` requires the partial rollback of effects. It is quite common, however, to compose transactions using `orElse` where it is determined with only reads that the first alternative will execute `retry`. A simple and very common example of this is shown in Figure 3.7. In the code for `dequeue` we can see there is only one case that leads to `retry`. This is when both the read and write stacks are empty, which we determine by reading both `TVar`s. When there are no writes, there are no effects to rollback so a hardware transaction can handle this case by simply jumping to the alternative branch’s code. Retry is implemented as an exception and potentially incurs some unnecessary overhead to get to the alternative branch. Looking at an example of code a user might write in Figure 3.8 we see a transaction that makes a choice with `orElse` between two queues preferring to read from `queueA`. Ideally we would like to elide the `orElse` altogether in these simple cases.

In cases where there are writes that happen in the first branch of an `orElse`, we can imagine transforming the code so that the writes are delayed until after the decision to `retry`. Certainly we could not always perform this transformation, but there should be many simple cases where we can trivially do this. Delaying a write would involve binding the value to write to a new name in a `let` expression and after the `retry` decision, performing a write. In the code between the binding and the write we need to be able to ensure that reads of the `TVar` in question are satisfied by the binding, not by a read from the variable.

Finally, we can view uses of `orElse` as a binary tree with alternative code paths as the branches. STM actions can be factored so that the left (first alternative) branch of `orElse` never contains a further nested transaction using the laws given in in the following equations. Note we are using
-- A queue with a read and a write end.
data TQueue a = TQueue (TVar [a]) (TVar [a])

-- Simply accumulate in the write end.
enqueue :: TQueue a -> a -> STM ()
enqueue (TQueue _ write) w = do
  ws <- readTVar write
  writeTVar (w:ws)

dequeue :: TQueue a -> STM a
dequeue (TQueue read write) = do
  -- First check for data in the read end.
  rs <- readTVar read
  case rs of
    (r:rs') -> do
      -- Pop off a value from the read end.
      writeTVar read rs'
      return r
    [] -> do
      -- Check the write end for values.
      ws <- readTVar write
      case reverse ws of
        [] -> retry -- Both read and write ends are empty.
        (r:rs) -> do
          -- Take all the values from the write end.
          writeTVar write []
          writeTVar read rs
          return r

Figure 3.7: Code for a simple transactional queue that keeps data in read and write stacks. Enqueued data is simply pushed onto the write stack while reads either take data from the read stack, or if that is empty, flip the write stack and take from that.

queueA :: TQueue Int
queueB :: TQueue Int

threadLoop = do
  value <- atomically (dequeue queueA `orElse` dequeue queueB)
  process value
  threadLoop

Figure 3.8: Simple example program composing queues with orElse
the operator (⊙) for \texttt{orElse}, (≫) for bind, and (≫) for “and then”.

\[
(a \odot b) \odot c \equiv a \odot (b \odot c) \tag{3.1}
\]
\[
a \gg (b \circ c) \equiv (a \gg b) \odot (a \gg c) \tag{3.2}
\]
\[
a \gg(\lambda x. b \circ x) \equiv (a \gg b) \odot (a \gg c) \tag{3.3}
\]

The first law (3.1) is one of the monoid laws which applies to implementers of the Alternative typeclass [40]. The second law (3.2) follows from the fact that branches that hit a \texttt{retry} roll back. Distributing the action \texttt{a} inside the \texttt{orElse} has the same meaning as performing \texttt{a} then choosing between \texttt{b} and \texttt{c}. It is also important that both sides of the equation have the same read set for any particular outcome. Using the right-hand side of (3.2) will result redundant work in the computation of \texttt{a} in both branches, but gives the opportunity to refactor the entire STM action into a tree that only continues to branch on the right. If the effects of \texttt{a} are buffered, the redundant work is avoided. The third law (3.3) generalizes the second to the bind operator. We may find it easier to manipulate programs based on laws written for the applicative typeclass rather than in terms of monad’s bind operator. There is ongoing work\(^1\) to automatically transform monadic code to the equivalent applicative functions.

Our planned approach for handling \texttt{orElse} is to build compiler passes that can elide \texttt{orElse} by discovering read only branches and creating read only branches by buffering writes. We plan to use the HERMIT compiler pass framework to explore this aspect of \texttt{orElse} [10]. We can also evaluate the expected effectiveness of particular transformations by applying them by hand to code. Some improvements may be expressible using GHC’s rewrite rules [29].

We will also note here that some of this work will be similar to the work by Xiang and Scott [38]. We hope that in the setting of Haskell this work will be easier. Similar additional compiler passes will be investigated that do not pertain directly to \texttt{orElse} but do deal with moving some execution outside of transactions and validating this preliminary work inside the transaction.

\(^1\)See \url{https://ghc.haskell.org/trac/ghc/wiki/ApplicativeDo}.  

32
Chapter 4

Thesis Work

In this thesis work we will explore the run-time costs of highly composable interfaces for concurrent programming. In this exploration we will look for ways to improve performance without sacrificing high-level properties expected in Haskell programs.

One of the aspects of the Haskell language that makes it an attractive research vehicle is its highly expressive type system. This allows ideas to be expressed in the language that in other languages would require modification to the language itself. For some of our work we will look to exploit this aspect of the language like many researchers before us. But the majority of our work will be looking at filtering down more of the high-level properties asserted by the type system and exploiting those properties in the run-time system. For instance, GHC’s STM could be implemented completely as a Haskell library [9, 14, 31]. The designers, however, choose to implement significant run-time support for STM. Implementations that do not have run-time support have significantly poorer performance [31]. While the list of primitive concurrency operations supported by GHC has been growing, we see much work to be done directly supporting or better supporting the high-level concurrency and parallel frameworks.

In this chapter we will see in detail the opportunities for continuing our preliminary work and extending it to other areas of the run-time system. We will start by discussing some broad questions that we are looking to help answer in our work in Section 4.1. This is followed by a description of some concrete projects we will work on in Section 4.2. Finally we will finish the chapter with a focused discussion of the approaches we will take to make progress on these questions and other questions we encounter along the way in Section 4.3.

4.1 Cross-project Goals

Some overarching goals that cross many projects we will be working on include:

- Gaining a better understanding of the interplay of lazy evaluation and concurrent execution in the context of Haskell. Haskell naturally has a support for futures, but its interface for the user is essentially implicit. What ways can we leverage the ability to move around the evaluation of pure code to the best benefit? How far can we push this support to minimize the code executed in critical sections and transactions? What inherent overhead comes with the implicit mutation of memoization? GHC relies on strictness analysis and user strictness annotation to avoid laziness when it is harmful to performance. What can work toward
laziness analysis such as [3] tell us about lazy-by-default languages? Can our work inform the profiled laziness approach and make it more robust in concurrent and parallel programs?

- Discover the benefits of expressing finer grained effects in concurrent programming. What compile-time distinctions can be added to TM actions or other concurrent API’s to improve performance without sacrificing the ease of programming applications and libraries?

- Research what issues arise when executing parallel and concurrent Haskell programs on machines with non-uniform memory access (NUMA) costs. Does Haskell’s purity by default help or harm? Does laziness fundamentally get in the way of locality?

- Research how programming with restricted effects could give rise to new ideas in hardware. What hardware changes can be made or features added that improve Haskell’s concurrent execution? What features could augment hardware transactional memory to make it more useful to Haskell? Where else would such changes be applicable?

These goals and questions are not tied to any particular project but will all be informed by the work we plan on doing. We have a start at answers to some of these questions and there is existing work that looks at answering some of the same questions. One project in particular that has produced some interesting results is the Intel Haskell Research Compiler. A paper to come out of that work seeks to measure the overhead of expressing programs in Haskell as compared to programs given in the C language [26]. An aspect of our thesis is similar to their question: what is the cost of expressing a program as a composition of robustly generic parts? Their approach in particular, however, sacrifices some of the benefits of GHC’s implementation by not supporting lightweight Haskell threads. They do, however, have instances where high-level type information is propagated further along in their compilation, enabling improvements that GHC misses.

We know that much of the performance of programs on current hardware is dependent on the efficient use of memory. Use of shared memory comes with significant performance costs that become more evident when scaling to more processors. What aspects of Haskell programs impact locality? How does composability impact memory use?

Haskell values are immutable and give abundant opportunities for shared substructure. Even data structures with the same run-time layout can be shared as values. Lazy evaluation, however, introduces mutation into these data structures in the form of updates to overwrite indirection with evaluated values. It is not clear in a multicore setting and much less in a NUMA setting where this sharing is beneficial and where it is harmful.

Haskell’s execution model involves significant indirection. Values may be in fully evaluated form or may be computations waiting for evaluation on demand. This is all transparent to the programmer and there are few opportunities to predictably influence the memory layout or sequencing of evaluation. Programmers also are in danger of harming composability by introducing evaluation ordering or removing indirection. In the context of STM there is no way to reliably influence the granularity of metadata. Each transactional variable must pay the full cost of this metadata.

4.2 Projects

The following is a list of projects that we will work on completing as we explore this thesis. The details and background ideas for these projects have already been covered in previous chapters.
4.2.1 TM Projects

These projects are largely about the user facing aspects of TM in Haskell.

- Finish retry and orElse support in the hybrid TM.
- Implement compiler passes for improving orElse.
- Develop benchmarks for retry and orElse that reflect real world uses.
- Develop an efficient TArray implementation and investigate opportunities for improving in-
direction and conflict detection granularity.
- Implement a NOrec based implementation of GHC’s STM. This will provide a better point
of comparison between non-GHC TMs and GHC’s existing implementations.
- Investigate benefits of further static distinctions such as read-only transactions.
- Improve random supply in transactions. Must be retry safe and have better performance
and interface than the current options. There may be other opportunities for supporting the
mixing of effects with TM.
- Formalize Haskell TM’s interaction with unsafe operations. What conditions are needed for
unsafe operations to lead to a bad execution in a doomed transaction?

4.2.2 Runtime System Concurrency

These projects are focused on particular areas of the run-time system that may benefit from im-
proved implementations including applying HTM, concurrent data structures, and improved mem-
ory use.

- Follow up on the work of Sivaramakrishnan et al. [32, 18] which adds the ability for library
writers to include custom schedulers for parallel execution. These schedulers are written in
some form of transactions but do not need the full TM interface. We will investigate using
HTM or a restricted STM to improve the current STM performance bottleneck.
- The parallel IO manager from Voellmy et al. [33] may benefit from a more appropriate con-
current data structure than what is currently implemented with MVar based implementation.
- Improve MVar implementation and look for opportunities to apply the lessons learned from
our work on TM in GHC to this API.
- Provide direct support for critical synchronization operations in existing parallel frameworks
such as LVars and Data Parallel Haskell. For instance, LVars are implemented without changes
to the run-time system. It employs a heavy use of atomicModifyIORef while at the semantic
level it can naturally express operations that may be highly amenable to an implementation
that performs a more efficient atomic fetch and increment instruction.
4.3 Research Methods

We will proceed in our research by continuing to make changes to the GHC run-time system while exploring how existing work in other areas has progressed the state of the art for concurrent programming and synchronization. Haskell has the benefit of a significant body of real-world applications that rely on its existing STM implementation. We will look to use some of these applications to build benchmarks that reflect performance in real world settings. We are also aware of uses of concurrent programming internal to the run-time system that are not implemented using STM due to its poor performance. We hope that having existing applications using STM will aid greatly in our evaluation of various changes and improvements that we will be making. We are also interested in making comparisons to work in other languages. To that end, some standard benchmarks for STM will be ported to Haskell. Translating to this setting will require some trade-offs. We will evaluate these trade-offs keeping our goals in mind. Additionally, we will look to include an STM algorithm closer to an existing implementation in another language to further illuminate performance compared to other systems.

The timeline given in Chapter 5 shows the particular projects we will be working on. They are ordered so that we start with projects with clearly defined goals and known improvements to be made. Through the process of working on these early projects we hope to discover techniques that work toward our goal, problems that lead to more fundamental questions, and insights into our existing questions. This process will greatly inform our approach to later projects. As we progress, it will be important to continually document our work and seek to publish our progress in appropriate venues.

In our preliminary work we have been in consultation with other researchers working on GHC. We will continue to collaborate where we have opportunities and work with researchers needing run-time support for their higher-level systems. The ongoing work on LVars is one opportunity for such collaboration. These opportunities may dictate changes to our timeline and may influence our goals to some degree. If a summer internship with researchers in the areas important to our work is available and will further our research goals, we will definitely consider the opportunity.

Where possible, we will try to evaluate our performance on the best available systems for the task. Some of our work would benefit from exploring hardware transactional memory with more features than are given by Intel’s TSX. If the work to have GHC running on these systems is not too large, we will explore these features. Larger systems that include TSX will also be of great value if and when they become available.
Chapter 5

Timeline

Summer 2014

- Finish retry and orElse support in the hybrid TM.
  - Include all HTM primitives in the code generator.
  - Make performance-critical code inline by writing it as appropriate primitive operations rather than foreign C calls.
  - Implement bloom-filter-based retry and measure its performance to find the most appropriate parameters and determine the viability of using the 8-bit abort code in HTM for indicating a degenerate bloom filter.

- Start implementing benchmarks that exercise retry and orElse based on existing program use.

Fall 2014

- Finish benchmarks for retry and orElse.
  - Investigate how programs use retry and orElse for event signaling, broadcast messaging, and barrier synchronization.
  - Refine these real-world uses into benchmarks that can flexibly expose the overhead and costs of the work done by retry. This will include known programs that have an STM bottleneck such as Sivaramakrishnan et al.’s parallel scheduler work [32, 18] and Voellmy et al.’s software defined network controller work [33].
  - Investigate how, if at all, such programs can be expressed in other TM implementations. Do these benchmarks have any place in efforts to make a new standard set of transactional benchmarks [30]?

- Implement compiler passes for improving code with orElse.
  - Use the HERMIT framework [10] to build compiler passes that right factor orElse trees and delay writes until after retry decisions when possible.
While implementing this, look for other opportunities to improve TM code with compiler passes.

Look for what it will take to build passes that minimize code in a transaction or passes that move code out of a transaction with extra validation inside the transaction similar to work by Xiang and Scott [38].

- Start work on efficient TArray support.
  - Initial implementation that sacrifices STM performance by coarsening the granularity of conflict detection while gaining better HTM performance by removing metadata and indirection.
  - Look for ways to also improve the STM performance.
  - Develop benchmarks that use TArray and convert some of the widely used STAMP transactional benchmark suite into a form that can benefit from these improvements [30]. For instance, using an array-based compact representation for a red–black tree.

- Continue communication with the LVar researchers about run-time system improvements to benefit their framework.

### Spring 2015

- Finish work on efficient TArray support.
  - Extend TArray ideas to records with unboxed transactional fields or unboxed values in TVars.
  - Explore trade-offs of pure functional persistent data structures and mutable concurrent data structures. Attempt to identify settings where one approach is more appropriate than another.

- Implement a NOrec-based implementation of STM and Hybrid TM for GHC as a point of comparison [7, 5].

- Apply what we have learned so far to the Lightweight Concurrency branch of GHC which allows for schedulers for parallel code to be written as transactional code in a library [32, 18].

- Explore improvements to IO manager’s concurrent data structures.

- Explore further compile-time distinctions between transactions.
  - Start by tracking read-only transactions in the type.
  - Explore further distinctions.

### Summer 2015

- Implement improved random supply inside transactions and explore other missing features and opportunities for extending Haskell’s TM interface.
– Example use case is building a concurrent skip list.
– Other opportunities could include inevitable transactions, lifting data in and out of transactional variables without copying, and extended debugging and event tracking abilities.

• Formalize the Haskell TM’s interaction with unsafe operations.
  – Define conditions that lead to bad executions in a doomed transaction.
  – Explore ways of including unsafe operations safely by either indicating what minimal properties must hold or providing an interface that ensures the correct executing in the face of inconsistent data by proof or sandboxing.

Fall 2015

• Apply what we have learned to other areas of the run-time system and other parallel and concurrent libraries.
  – LVar implementation.
  – MVar implementation.
  – Data parallel Haskell.

• Explore transactions in the context of distributed process.

• Begin writing thesis.

Spring 2016

• Continue work on unfinished projects.

• Explore new opportunities discovered in the work done so far.

• Finish writing thesis and prepare defense.
Bibliography


