Reducing resource utilization when simulating hardware in CLaSH

Bram Wesselink
Master Embedded Systems
b.h.t.wesselink@student.utwente.nl

MSC THESIS
Computer Architecture for Embedded Systems (CAES)
University of Twente

July 4, 2023

Acknowledgement

This thesis has been written as part of my master's programme Embedded Systems at the University of Twente. The research we present in this document was conducted for the Computer Architecture for Embedded Systems (CAES) group. Additionally, this research was conducted in collaboration with QBay-Logic, the company that develops the Clash language.

I would like to thank my main supervisor Hendrik for helping me find this interesting topic for my research and assisting me with the entire process of my thesis. During the weekly meetings that I had with Hendrik, he was always able to poke at my research and help guide me in which topics I should explore, while at the same time ensuring that I did not get lost in the endless directions this research could have gone. Secondly, I would like to thank Christiaan, the co-founder of QBayLogic, for his help in my research. Once every two weeks, Hendrik and I visited QBayLogic to have a meeting with Christiaan about the progress of my work. During these biweekly meetings, Christiaan was able to provide me with great insights into the inner workings of both the Clash ecosystem but also the internals of GHC. Additionally, I would like to thank Marco and Vadim for being part of my thesis committee and assessing and providing feedback on my work.

Lastly, I would like to thank my parents, with whom I still live together. Not only during this research but during my entire studies I have always been able to rely on my parents to support me and enable me to pursue my passions. Doing an individual research project like this thesis, can from time to time get a bit lonely, but at these times my parents were pretty much always there to drink a cup of coffee and have a chat.

Contents

Chapter 1

Introduction

1.1 Context

Functional programming has proven to be a useful tool for hardware design [22]. With a language like Clash, a hardware description language using Haskell, you can design, simulate, and compile circuits to VHDL or Verilog. For simulation Haskell uses lazy evaluation. Lazy evaluation allows the Haskell runtime to postpone evaluating an expression until it needs to use the result. This lazy evaluation strategy of Haskell allows for simulating compact descriptions that closely map to and from hardware data-flow graphs.

Simulation is an important aspect of hardware design, as a simulation is used to verify that the hardware description behaves correctly. Due to this, the simulation must be quick, such that a hardware engineer can quickly verify their changes to the hardware. In some cases, hardware simulation using lazy evaluation can be an improvement over strict simulators regarding performance. The reason for this improvement is that certain computations are maybe not required and thus can be omitted, whereas a strict language likely would evaluate these computations.

1.2 The Problem

However, Haskell's lazy evaluation is not without problems. Whenever there are computations of which the result is not immediately required, Haskell will place these as suspended calculations in memory. For some programs, the number of suspensions will grow as the runtime explores more of the program. In this case, the memory usage will also increase. This effect is called a *space leak*.

To keep memory usage to a minimum Haskell uses a garbage collector. Whenever the memory usage passes certain thresholds, this garbage collector is invoked in an attempt to free unused memory. When a space leak is present these thresholds will be frequently surpassed and the garbage collector has to run frequently. This has the effect that a space leak is expensive both in terms of memory and time.

The usual approach to solving this problem in Haskell is to force the evaluation of these growing expressions early, causing them to reduce. It can however be difficult to find the precise location, where in the code this forcing should occur. Moreover, it is not ideal for a hardware engineer to make these decisions, as they require deep knowledge of the evaluation mechanisms of Haskell to understand the cause of the problem. This requirement is not desirable to have.

1.3 Research objectives and questions

The primary goal of this thesis is to research, and explore solutions to, the problem of space leaks in Haskell within the context of hardware simulation.

For the solutions explored in this thesis, an effort is made that they do not require annotations in the hardware descriptions themselves. Instead, we focussed on solutions that can be implemented in the internal Clash APIs such that they are completely transparent to hardware engineers using Clash. Additionally, for the explored solutions, the goal is to limit the negative impact on simulation performance in the case that no space leaks would occur.

Based on these objectives, this thesis will seek to answer the following research questions:

- To what extent does a space leak in Haskell harm Clash simulation performance?
- Under what conditions can a space leak occur when simulating hardware in Haskell?
- How can we remove or reduce the impact of these space leaks, while minimizing the impact on simulation performance for circuits without space leaks?

1.4 Structure of the thesis

In this thesis, first some background knowledge on lambda calculus, Haskell and Clash is summarized in Chapter 2. Next, in Chapter 3 we present existing work related to the topic of the thesis. In Chapter 4 an analysis of the problem is given. In this chapter, we present a hardware description that reproduces the problem of a space leak when simulated using Clash. In Chapter 5 we provide a set of three conditions that a hardware description has to meet for a space leak to occur. Based on these conditions we propose a set of solutions that remove the space leak or mitigate the negative effects of the space leak. In Chapter 6 we perform several experiments using the solutions described in Chapter 5 on the model presented in Chapter 4. In Chapter 7 we conclude the thesis and give recommendations and mention some topics that can be explored in future research.

Chapter 2

Background

2.1 Introduction

In this chapter, we will discuss the core background knowledge that is required for the understanding of the rest of the work in this thesis. We will cover the following topics:

1. Normal Forms and Call-By-Need
    - Lambda Calculus
    - Normal Forms
    - Lazy Evaluation
    - Call-By-Need
2. Haskell and Clash
    - Heap usage in Haskell
    - Garbage Collection
    - The Clash language and its core primitives

2.2 Normal Forms and Call-By-Need

In this section, we will cover the strategies that Haskell uses to evaluate expressions and the mathematical background behind these strategies.

2.2.1 Lambda calculus

Many concepts in Haskell are based on a formal model in computer science called the lambda calculus. To understand how Haskell expressions are reduced it is useful to understand the basics of lambda calculus. It should be noted that the material described in this section only covers the basics, and certain rules are omitted for compactness. A more complete description of lambda calculus is given in [5].

The syntax of lambda calculus consists of three rules:

1. Variables: conventionally denoted by lowercase letters like x, y, ...
2. Abstraction: if E is a λ-*expression* and x is a variable then $(\lambda x.E)$ is a λ-*expression*.
3. Application: if both E and F are λ-*expressions* then (EF) is a λ-*expression*.

In an abstraction of the form $(\lambda x.E)$, λx is called a *binder*, and the variable x is called bound in the expression.

To perform a calculation described by a lambda expression, two types of conversions are performed:

1. α-*conversion*: Renaming the bound variable x in a λ-*term* of the form $\lambda x.E$.
2. β-*reduction*: Substituting the argument F in the abstractions body E: $(\lambda x.E)F \to_\beta E[x := F]$

Another important term from lambda calculus is *redex*, which is short for "reducible expression". A lambda expression is a *redex* if it can be β-*reduced*. For example, the λ-*expression* "$(\lambda x.x)a$" is a redex as it can be β-*reduced* to the expression "a".

In Section 2.2.4 it will become clear how these concepts are used to implement lazy evaluation in Haskell.

2.2.2 Normal forms

In Haskell, expressions are evaluated to weak head normal form (WHNF). An expression is said to be in WHNF if it is either a lambda abstraction or a data constructor.

The following lambda expression is in WHNF:

$\lambda x.((\lambda y.y)x)$

Note that the body of the outer lambda abstraction can be reduced:

$\lambda x.((\lambda y.y)x) \to \lambda x.x$

Because this reduction is still possible the expression is in WHNF.

In Haskell lambda expressions where the body can still be reduced are also said to be in WHNF. Additionally, data constructors in Haskell where the arguments can still be reduced are also in WHNF.

The following Haskell expressions are in WHNF:

- Just (3 + 2)
- $\lambda x.\sqrt{2} \times x$

The first example is a data constructor of the type Maybe located in Haskell's standard library (also called the prelude). The data constructor Just takes one argument, which is the optional value, the Maybe type also has a second data constructor called Nothing, which takes no arguments. In the example we used the expression 3 + 2 as the argument, note that although 3 + 2 can be further reduced the expression Just (3 + 2) is still in WHNF as it is a data constructor.

The second example is a lambda abstraction, this expression is similarly in WHNF even though the body of the abstraction can be further reduced (by evaluating $\sqrt{2}$ to a constant floating point value). This is also where the term weak in WHNF comes from, the body of the abstraction is not yet fully evaluated. If it was fully evaluated, then the expression is also in head normal form (HNF). Note that in the case of a data constructor, there is no significant difference between HNF and WHNF[21].

An expression can also be in normal form (NF). An expression is said to be in normal form when no further reductions are possible, i.e. it is fully evaluated.

Take for example the following Haskell expression:

```
(1+2,3)
```

Listing 2.1: Haskell expression in WHNF

This expression is in weak head normal form because (a, b) is syntactic sugar for $(,)\ a\ b$ where $(,)$ is a two-element tuple constructor with a and b as arguments. This expression however is not in normal form because the first argument $1 + 2$ is not fully reduced. It can still be reduced because a sufficient number of arguments are applied to the built-in (+) operator.

2.2.3 Lazy Evaluation

Haskell uses a technique called lazy evaluation to evaluate expressions. The ideas behind lazy evaluation are:

1. to only evaluate an expression if it is necessary

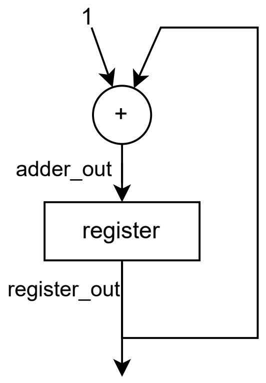


Figure 2.1: Diagram of a simple accumulator

2. to never evaluate a binder twice

The idea of never evaluating an expression if it is not necessary introduces the concept of *call-by-need* function application (in contrast to the more common call-by-value mechanism) [8]. The idea of never evaluating a binder twice introduces the concept of expression *sharing*. When an expression is represented as a graph in memory sharing follows naturally. This is because common subexpressions can be represented by multiple references to one node in a graph. The call-by-need mechanism can then be implemented by reducing this graph in a specific order [11]. The reduction order that achieves this call-by-need mechanism is discussed in more detail in Section 2.2.4.

One of the main benefits of lazy evaluation for hardware simulation is the fact that it allows recursive descriptions without a base case. Lazy evaluation makes such a description possible because it only has to evaluate these recursive descriptions up to the point that is required for the number of output samples requested. In contrast, when a strict language, i.e. a language that evaluates function arguments before substituting the body of the function, would try to evaluate such a recursive description it would never complete the evaluation, hence it never produces a result. When expressions are represented as a graph these recursive descriptions would create a cycle in that graph. This is similar to the data-flow graph (DFG) for that piece of hardware, which also would contain a cycle.

Lazy evaluation for hardware simulation

To realize the simulation and compilation of hardware descriptions written in Haskell, Clash was introduced. Clash is a Haskell package containing a library with common hardware primitives such as registers. Additionally, it also contains a compiler that translates Haskell code to Verilog and VHDL.

Consider the hardware shown in Figure 2.1. The Clash description for this hardware is given in Listing 2.2. In this Haskell code, first, the Clash prelude is imported to get access to the register primitive defined in the Clash library. Next, the system is defined by using let bindings to create named references of specific signals in the hardware (similar to the names shown on the arrows in Figure 2.1), then these names are used in the definitions of these signals to create recursive data dependencies. When comparing Figure 2.1 and Listing 2.2 we can see how the Clash code is a near one to one translation of the graphical representation, the evaluation of such a circuit is possible due to lazy evaluation.

```haskell
import Clash.Prelude

system = let adder_out    = register_out + (pure 1)
             register_out = register 0 adder_out
         in register_out
```

Listing 2.2: Clash representation of the circuit shown in Figure 2.1

2.2.4 Call-by-need

To implement lazy evaluation, the call-by-need mechanism is required. Call-by-need can be implemented by always reducing an expression in *normal-order*. To reduce an expression in normal-order the left-most outer-most redex is selected and then *β-reduced*.

```haskell
e = (\a -> (\b -> a * a + b)) (1 + 2) (2 - 1)
```

Listing 2.3: Simple Haskell expression which can be reduced using normal-order reduction

To understand how this results in the desired call-by-need mechanism, we can go through normal-order reduction on expression e in Listing 2.3.

The left-most outer-most redex is the application : `(\a -> (\b -> a * a + b)) (1 + 2)`. This results in the bound variable a being substituted with the expression `(1 + 2)`. Resulting in `(\b -> (1 + 2) * (1 + 2) + b) (2 - 1)`. Next, the left-most outer-most redex is the expression itself and can be reduced by substituting b with the expression `(1 + 2)`. This results in `(1 + 2) * (1 + 2) + (2 - 1)`.

For readability purposes, we have presented this as a simple substitution, but in reality, Haskell keeps track of the binders the expressions belong to. This allows Haskell to implement sharing, once the (1+2) expression requires evaluation, which, in the original lambda body was the binder λa, Haskell will substitute the value 3 for both instances of the expression. The simple substitution without keeping track of the bound expressions is called *call-by-name*.

This reduction order shows, that the arguments of the initial lambda function were not evaluated until they were required by the (+) function in the body of the lambda expression. This delayed evaluation of arguments creates the desired lazy behaviour.

It should be noted that this reduction strategy always starts with selecting the left-most outer-most redex, regardless of where that redex is. This means that an expression in WHNF can still be reduced using normal-reduction. In fact, if an expression has a normal form, normal-order reduction will find that normal form [5]. However as was stated before, Haskell will not evaluate all the way down to NF, instead, it stops its reduction once an expression reaches WHNF.

2.3 Haskell and Clash

In this section, we will cover features and implementation details of Haskell and Clash that are required to understand this thesis. In Section 2.3.1 we will cover how and why Haskell uses heap memory. In Section 2.3.2 we will cover how Haskell deals with unused memory. In Section 2.3.3 we will cover a variety of primitives in the Clash library that are important for this thesis. Lastly, in Section 2.3.4 we will cover why undefined values are relevant in the context of hardware design and how Clash deals with them.

2.3.1 Heap usage in Haskell

As mentioned in Section 2.2.4 Haskell needs to keep track of the state of evaluation of the different binders to implement call-by-need. This requires a dynamic data structure and as a consequence Haskell makes heavy use of heap memory. Every time an expression is bound to a binder, this (still unevaluated) expression is placed on the heap. These binders are frequently utilized in functional languages like Haskell, which in turn results in a considerable number of heap allocations. These frequent allocations give rise to the problem of freeing unused heap objects.

2.3.2 Garbage Collection

To accommodate freeing unused heap objects, Haskell uses a garbage collector. The fact that data in Haskell is immutable causes a frequent allocation of temporary data. Due to this, the garbage collector has to frequently check young data to determine if it is still in use. There will, however, also exist long-living data, for this data it is beneficial to not check it as frequently as it likely won't need to be freed. To solve this, Haskell uses a generational garbage collector. When objects are first allocated they are placed into a piece of memory called the nursery. Once this small section of memory (by default 512kb) is exhausted, the garbage collector will scan this data and copy all the (still) live data into the main memory section. This main memory section is then also divided into multiple generations (by default 2). These generations are handled similarly to the nursery. Once certain thresholds, either a specific value or a specific amount of memory growth are reached, the garbage collector will traverse this generation and copy all live data to the next generation. An important thing to note here is that the live data is copied, and thus the runtime of the garbage collector is proportional to the amount of live data on the heap. The more live data there is, the longer a garbage collection pass will take. Additionally, if the data on the heap is growing, these thresholds will be exceeded increasingly frequently, and thus the garbage collector will be invoked more frequently.

2.3.3 Important Clash primitives

To assist hardware designers in describing hardware using Clash, Clash has a library (called the prelude) which contains types and components that are commonly found in hardware. Examples of symbols that are present in the prelude are:

- The *Signal* data type
- register
- blockRAM
- Signed / Unsigned number types

Signal

An important concept in Clash is signals. These are implemented as linked lists using the `:-` data constructor. Each value in the linked list corresponds to the value of that signal at a specific clock cycle. I.e. the first value in the linked list is the value of that signal at $t = 0$.

The Signal data constructor is implemented recursively, similar to how a regular linked list in Haskell is implemented. A simplified implementation of the Signal data type is shown in Listing 2.4.

```haskell
data Signal a = a :- Signal a
```

Listing 2.4: Simplified implementation of Signal

The constructor takes a single argument a. Note that a is generic in its type, this allows a hardware engineer to use different types in the design and even custom data types created by the hardware engineer. Of course, for synthesis, the data type used in a signal should be finite in nature.

The Signal implementation in the Clash prelude is slightly more involved than what is shown here, as it also stores information about properties of the hardware, like clock and reset behaviour.

In reality, a hardware engineer is not supposed to create these signals manually using the data constructor. Instead, the input signal is constructed by the *simulate* function from a regular Haskell list. The Signal data constructor is hidden away in an internal Clash module.

To implement combinatorial logic Clash implements the Functor and Applicative type classes for signals. These type classes are a concept in Haskell that allows a programmer to "lift" a function into a specific "domain". In the case of the Signal domain, this means that an arbitrary function can be applied to every value in a signal.

```haskell
greater :: Signed 32 -> Signed 32 -> Signed 32
greater a b
        | a > b = a
        | otherwise = b
```

Listing 2.5: Haskell function that takes two numbers and returns the greater of the two

For example, consider the Haskell function shown in Listing 2.5, which takes two numbers as arguments and returns the greater one of the two. If we have two signals of Signed 32 numbers and want to create a third signal that represents the greater one of the two, we can use the Functor and Applicative properties to lift greater into the Signal domain. The resulting "lifted" greater function will have the following type signature and definition:

```haskell
greaterSignal :: Signal (Signed 32)
              -> Signal (Signed 32)
              -> Signal (Signed 32)
greaterSignal = liftA2 greater
```

The type signature of liftA2 is:
`liftA2 :: (a -> b -> c) -> Signal a -> Signal b -> Signal c`.

Register

To implement stateful behaviour in a circuit, Clash has the *register* primitive in the prelude. This primitive takes an input signal and produces a delayed version of that signal at the output. The register allows sequential computation with cyclic data dependencies, an example of this is an accumulator.

A simplified model of a register is shown in Listing 2.6.

```haskell
register :: a -> Signal a -> Signal a
register initial signal = initial :- signal
```

Listing 2.6: Simplified implementation of register

Here we see a nice usage of lazy evaluation, even though signals are infinite, we can still use it in an expression, as only a finite portion will ever be needed (and thus evaluated). From this model, it is clear that the input signal is delayed by one clock cycle as some initial value is appended at the head of the signal.

Again, this is simplified from the actual implementation in the prelude, as the registers also have to respect reset and enable signals.

2.3.4 Undefined values in hardware

When dealing with software, undefined values usually are undesired. However, when describing hardware, not every part of a circuit is always active, thus it is perfectly reasonable to have a signal that may contain an undefined value for some time. In case the Haskell runtime would evaluate such an undefined value, it would halt program execution and display an error message. To prevent this, Clash introduced a special kind of error function: `errorX`, that, when encountered by the simulate function, will display an 'X' to the user instead of displaying an error. Additionally, if an undefined value is created using `errorX` is explicitly forced using a strictness annotation, the simulation will not halt [4].

This fact is especially important for us because in Sections 5.2 and 5.3.1 we will look at forcing values using strictness annotations. If we take the careless approach of using regular Haskell errors, then this could lead to the undesirable effect of halting the simulation prematurely.

Chapter 3

Related Work

3.1 Clash

The functional hardware description language Clash was first introduced in 2009 in the Master's theses of Kooijman [12] and Baaij [2]. Kooijman focussed on the interpretation of Haskell and how to compile a description written in Haskell to a more traditional hardware description language (HDL), while Baaij focussed on the Haskell type system for hardware design and the simulation aspect of Clash.

In Baaij's thesis, a section was dedicated to how Haskell's lazy evaluation may have an impact on the behaviour of the simulator. He mentions a case where there is a difference in functional behaviour between Haskell's lazy evaluator and a traditional hardware simulator based on VHDL. The reason for this discrepancy, that is mentioned, is that in Haskell some expression (which would potentially result in a runtime error) might never be evaluated due to laziness, whereas in hardware, and thus in a VHDL simulation, the signals and dataflow will be instantiated and hidden away after a multiplexer. In case of an invalid computation, like one that results in overflow, the VHDL simulation might error whereas a Clash simulation will not.

3.2 Graph reduction

In Haskell, the graph reduction is done by the Spineless Tagless G-machine (STG), which was introduced by Peyton Jones in 1992 [10]. This abstract machine is capable of evaluating a small functional language, the STG language. The machine is based on a G-machine as described by Kieburtz in 1985 [11], the G-machine is responsible for evaluating functional languages by using graph reduction.

The STG builds a graph of an expression where each node in that graph is an object stored on the heap, these objects are called closures. In traditional G-machines, each of these objects would contain some form of tag, which indicates how the runtime should treat that object. In the STG however, the tags are eliminated in favour of code pointers to the correct instructions on how to handle them. This allows the runtime to simply follow that code pointer and execute the instructions rather than having to examine a tag first. There are two important types of closures:

1. head normal forms (which is either a function or some data)
2. not yet evaluated suspensions (also called thunks)

When the runtime system needs the value of a thunk, the thunk is forced. This means that the thunk is entered by following the code pointer of the thunk, when the instructions located at this pointer are executed the expression is evaluated. After evaluating the expression the code in the thunk has some instructions to arrange an update of the graph, this replaces the thunk node with a node containing the evaluated head normal form value. This updating behaviour is what allows sharing of expression results. If there were multiple references to a thunk, then after forcing that thunk once, it is replaced with the value and subsequently all references now point to that value.

3.3 Space leaks

The idea to delay evaluation until an argument is required can unfortunately result in a worse runtime performance than a strict language would have for that same computation. The main reason for this is a concept called a space leak. In Chapter 4 we will see an example of such a space leak in the context of hardware simulation.

Space leaks have been extensively explored and documented in reports, books and blog posts [14, 13, 24].

The main reason why a space leak can have such an impact on runtime performance is due to the workings of the garbage collector. As explained in Section 2.3.2 the execution time of the garbage collector is proportional to the amount of live data. If a space leak exists in a Haskell program, the amount of live data will grow over time. Because of this, a space leak causes the garbage collector to spend increasingly long on identifying objects to clean up. Additionally, the growing heap usage causes the garbage collection process to be involved more frequently.

The usual way to deal with these space leaks is to identify the location of the space leak, and then preemptively force evaluation to reduce a growing expression [18]. Forcing these expressions is done using a special function called *seq*. *seq* takes two arguments a and b, and before seq returns a value, it ensures that both a and b are evaluated to WHNF. This creates a sort of arbitrary data dependency between a and b. To make this more convenient GHC has a language extension called *BangPatterns*. This extension allows the programmer to add an "!" in front of the name of a binder, i.e. function arguments, which will force that binder to be evaluated before the function application is reduced.

While this approach works in most scenarios, the first step (identifying the location of the space leak) can be tricky and requires extensive knowledge of the evaluation model of Haskell. Part of this thesis will focus on identifying the main location where a space leak could occur in a hardware description, and we will consider how strictness annotations in the Clash prelude can decrease the likelihood of space leaks in Clash simulation.

3.4 Language optimizations

Another approach to improve the performance of lazy evaluation (or eliminate space leaks) is by modifying core systems of the Haskell language.

One such system that can be modified is the garbage collector, Wadler has shown that by using a simplified evaluator in the garbage collector a class of space leaks can be eliminated [23]. The core idea is that in the case of a pair, it is often beneficial to evaluate calls like fst and *snd* in a strict fashion, to prevent unnecessary thunks. With the modifications suggested by Wadler, whenever these calls are encountered by the garbage collector, it would evaluate them immediately. Although this does not address all space leaks, it can make it easier to transform a piece of Haskell code that has a space leak, into one that is covered by this modification, thus eliminating the space leak.

Another system that can be modified is the STG reduction machine, as mentioned in Section 3.2 the STG is a tagless machine. As pointed out by Marlow et al., this tagless approach has some consequences on branch prediction performance [17]. They proposed a system where the lower 2 to 3 bits of the code pointers, which on most architectures are always 0, can be used to add some information about the closure. Specifically, they suggest storing whether a closure is an evaluated data constructor (and if there are multiple for a given type, which) or if it is an unevaluated thunk. In the case of a Closure that represents data of type Maybe for example, the lower bits could be 00 in case it is not yet evaluated (a thunk), 01 in case it is Nothing and 10 in case it is Just a. By doing this, the STG can prevent branching al together and thus reduces the number of mispredictions by the branch predictor. This also allows *seq* calls to have very little runtime overhead when applied to a value that is already in WHNF, as the STG can quickly check whether the heap object is a thunk or not.

These optimizations are implemented in GHC which is used by Clash for simulation.

3.5 Different evaluation strategies

3.5.1 Eager evaluation

As eager languages do not suffer from space leaks, it might be worth looking at eagerly evaluating Haskell expressions. A system for doing this while also complying with Haskell language semantics was described by Maessen [16]. In his work, he described a strategy where expression bindings in Haskell are immediately evaluated instead of lazily when needed. To comply with Haskell language semantics, for example, allowing computations on infinite data structures, Maessen introduced resource bounds in the evaluator. Whenever a computation would exceed these resource bounds (e.g. a maximum stack size) it suspends the computation (creating a thunk). Maessen showed that the language semantics of Haskell can be achieved with better heap space behaviour than a lazy evaluator has, however for expressions where laziness is required the resource-bounded evaluator showed significant overhead when compared to GHC. Because Clash heavily relies on laziness with its definition of signals as described in Section 2.3.3 this method likely is not suitable for Clash simulation.

3.5.2 Optimistic evaluation

A similar idea to resource bounded eager evaluation was introduced by Ennals et al. called optimistic evaluation [6]. In this case, each let binding gets a runtime configurable switch specifying whether it should be evaluated eagerly or lazily. By default all bindings have this switch set to evaluate expressions eagerly. To maintain correct Haskell semantics a time limit is introduced on the evaluation of expressions. When this limit is reached the evaluation is suspended and a thunk is created such that it can be lazily evaluated in the future. Additionally, the switch for the let expression that took too long to compute is turned off such that it will always be evaluated lazily in the future.

The adaptive nature of optimistic evaluation is an advantage over the eager evaluation system shown by Maessen. On average optimistic evaluation shows a speedup of just over 15% and no slowdowns of more than 15%. These results are significantly more promising for Clash simulation than those of eager evaluation. However, as Ennals et al. also mention, their experiments do *not* show that *no* program will run much slower compared to the current GHC. So it might be the case that some property of this technique significantly hurts the performance of Clash simulations. To ensure that this is not the case for Clash, it either has to be proven that this is not the case for any program (which was mentioned as future work by Ennals et al.) or experiments with optimistic evaluation have to be done on Clash simulations. These experiments will not be performed in this thesis, but we will mention them in Section 7.4.

3.6 Arrowized Functional Reactive Programming

As explained in Section 2.3.3, signals are an important concept in Clash. Due to this, you could classify Clash as a Functional Reactive Programming (FRP) language. The most important concept in FRP are these time varying signals [9]. In a continuous context, these signals are usually defined as a function of time:

```
Signal a = Time -> a
```

However, in the context of hardware design, timing is discrete and has a fixed period between samples. This means that the time argument can be omitted and a signal can be fully represented as a list (stream) of values.

As shown in Section 2.3.3 Clash does this in the form of a linked list:

```
Signal a = a :- Signal a
```

Using these signals as a first class primitive does have a drawback, namely that it makes it easy to create space leaks [9, 15].

An example of a space leak that arises from using signals as first class values is shown in Listing 3.1.

```
accumulator :: Signal Int
accumulator = register 0 (fmap (+1) accumulator)
```

Listing 3.1: Compact Clash description of an accumulator

In this Clash description, the accumulator as shown in Figure 2.1 is written in a single line of Clash code. Although this is a very straightforward and compact way to write such a circuit, the simulation performs significantly worse than the more extensive description that was given in Section 2.2.3:

```
import Clash.Prelude

system = let adder_out    = register_out + (pure 1)
             register_out = register 0 adder_out
         in register_out
```

Listing 3.2: Accumulator description using let bindings (same as Listing 2.2)

The reason for this difference is the introduction of 2 binders, adder_out and register_out. These binders allow Haskell's call-by-need mechanism to share partially computed values.

In Listing 3.1, there are no such let binders, and Haskell is unable to introduce sharing with top level names like *accumulator*. This means that for every consecutive value that we try to sample from this accumulator all previous iterations always have to be computed.

A solution to this class of space leaks is to abandon signals as first class data types entirely [9, 15]. Instead, a circuit can be described by *signal functions*. A signal function can be described by the following definition:

```
SF a b = Signal a -> Signal b
```

In the case of the accumulator example we have no input signal so we can use the type

```
accumulator :: SF () Int
```

to describe this circuit.

To create these signal functions without being able to create signals directly (as we want to disallow signals from being used as first class entities), we need a set of primitive signal functions and a set of composition operators [9].

For combinational logic a primitive operator to "lift" regular Haskell functions into the SF domain is sufficient. This is called the arrow operator:

```
arr :: (a -> b) -> SF a b
```

For sequential logic, we also need a stateful SF primitive representing a register:

```
register :: a -> SF a a
```

Next, we need some functions to compose these signal functions:

```
-- Chaining two signal functions together:
(>>>) :: SF a b -> SF b c -> SF a c

-- Creating a feedback loop
-- c is the type in the loop
-- a is the input and b is the output
loop :: SF (a,c) (b,c) -> SF a b
```

Using these primitives we can define the accumulator:

```
adder :: SF Int Int
adder = arr (+1)

accumulator_reg :: SF Int Int
accumulator_reg = register 0

-- NOTE: To use the loop combinator we need to have the correct
-- signal function as an input type.
-- For this reason we create this helper function that ensures
-- the inputs and outputs are in the correct form.
loop_func :: SF ((), Int) (Int, Int)
loop_func = (arr snd)
                >>> adder
                >>> accumulator_reg
                >>> (arr (\a -> (a, a)))

accumulator :: SF () Int
accumulator = loop loop_func
```

In this case, the loop operator is responsible for creating the feedback loop. To prevent the memory leak that was previously present in Listing 3.1, the *loop* primitive can be defined using an internal binder such that the Haskell runtime uses sharing.

One potential drawback that may become apparent is the increased explicitness in the structure of the design. This explicitness makes the code significantly longer and can reduce its readability. To remedy this a special syntax for these arrows was introduced in [20]. This syntax is available in GHC using a language extension called *Arrows* [1].

Until now, we have not yet discussed how these signal functions are represented and simulated. There are two main approaches[19]:

- Stream based approach
- Continuation based approach

The stream based approach internally uses a data structure to represent signals (like the Signal data structure already implemented in Clash). The stream functions are then applied to every sample in this stream to produce an output stream:

```
type SF a b = Signal a -> Signal b
```

The continuation based approach does not implement the signals directly, but instead implements the signal functions using transition functions. These transition functions take one sample of the input and produce the corresponding output along with a new signal function that is used for the next input sample. This new signal function that is produced is called a *continuation.*

```
type SF a b = SF { tf :: a -> (b, SF a b) }
```

In the definition shown above we can see that the SF internally has a transition function (called tf) that takes an input and returns a pair of values (the output, and the continuation).

A continuation based approach, with accompanying implementations for the arrow primitives, for Clash was given by Gerards et al. [7]. In this work it is shown that the arrowized FRP approach provides a pleasant notation for synchronous hardware descriptions.

Although arrows are less prone to certain classes of space leaks (like the one mentioned in Listing 3.1), they are not completely immune. In fact, the specific type of space leak that we will discuss in this thesis (as described in Chapter 4) would still be present when Clash was to use arrows. However, FRP does provide other performance benefits too, one idea that was described in [19] is to implement the primitive arrow functions as data constructors. If this were done in Clash, it would be possible to use pattern matching to do some analysis on the structure of the circuit before simulation. This creates some opportunities to do dynamic optimizations on the circuit before simulating it, some of these opportunities are discussed in Section 5.4.2.

Chapter 4

Problem Analysis

4.1 Introduction

In this chapter, we will analyze the runtime behaviour of Clash simulations. In particular, we will look at how a specific description might be prone to space leaks, and how these space leaks can be recognized when looking at runtime statistics. We will also take a closer look at the heap closures that are constructed during a simulation with a space leak.

4.2 Model with a space leak

To understand how lazy evaluation can become problematic when simulating hardware in Clash, we design a model that reproduces the problem.

```
import Clash.Prelude

system = let adder_out    = register_out + (pure 1)
             register_out = register 0 adder_out
         in register_out
```

Listing 4.1: Simple Clash accumulator, (same code as shown in Listing 2.2)

The model as shown in Listing 4.1 does not contain a space leak. When it is simulated, Haskell uses a constant amount of memory and the runtime scales linearly with the simulation length.

There are two reasons why this simulation does not suffer from a space leak:

1. The result is needed every clock cycle
2. The data type used in the register is not complex

The first reason is quite straightforward: When the result is needed, the expression will be forced to evaluate. So, when Haskell tries to print the result to the user, it will force the calculation to occur. This means that the calculation was not delayed sufficiently long for a space leak to build up.

The second reason has to do with the actual implementation of the register function in the Clash prelude. This implementation already contains a call ‘seq‘ forcing the value to WHNF every simulation iteration:

```
register_model :: a -> Signal a -> Signal a
register_model out (in :- rest) =
    out ‘seq‘ (out :- register in rest)
```

This means that if a data type is used, where the WHNF is the same as the NF, no space leak can build up, as the calculation is not lazy at all. However, if a more complex data constructor is used, one where the WHNF is not the same as the NF, then a calculation could build up lazily in the arguments of this data constructor. One example of such a data constructor is a tuple. The WHNF of a tuple does not require the values stored inside the tuple to be evaluated, hence space leaks could occur there.

The code shown in Listing 4.2 is similar to the one in Listing 2.2, with the key difference that there now is a multiplexer at the output. This means that the first reason previously given as to why no space leak occurred, no longer is correct. As, in this case, the result of the accumulator is only needed when the output_enable signal is True.

```haskell
import Clash.Prelude

accumulator = let adder_out    = register_out + (pure 1)
                  register_out = register 0 adder_out
              in register_out

system output_enable = mux output_enable accumulator (pure 0)
```
Listing 4.2: Clash representation of an accumulator with a multiplexer on the output

In this specific case, a space leak still does not exist, as the register only stores a simple integer.

The code shown in Listing 4.3 does contain a space leak, as now a tuple is stored in the register, and forcing this tuple to WHNF is not sufficient to prevent the space leak.

```haskell
import Clash.Prelude

tuple_add :: (Int, ()) -> Int -> (Int, ())
tuple_add (a, _) b = (a + b, ())

accumulator = let adder_out    = liftA2 tuple_add register_out (pure 1)
                  register_out = register (0, ()) adder_out
              in fst <$> register_out

system output_enable = mux output_enable accumulator (pure 0)
```
Listing 4.3: Clash representation of an accumulator with a multiplexer on the output, where the accumulator state is wrapped in a tuple

When this description is simulated with an input signal that contains a large period of continuous False values, the simulation takes significantly longer and memory utilization is growing rapidly.

Table 4.1: Runtime statistics of simulations as described in Listings 4.2 and 4.3

	Int stored in register	tuple stored in register
Bytes allocated on heap	1,472,834,880	1,650,127,864
Bytes copied during GC	1,000,440	187,739,944
GC Gen 0 collections	1420	1569
GC Gen 1 collections	2	13
Mutation time in seconds	0.215	0.220
GC time in seconds	0.003	0.132
Productivity	98.5%	62.5%

4.3 Runtime statistics

When simulating the model, The runtime can provide statistics regarding the STG mutation time and garbage collection time. For both simulations the first 1 million cycles False was used as the output_enable and then for 1 cycle True was used. The statistics are summarized in Table 4.1.

When comparing the two sets of statistics it can be seen that the total number of allocations and the total mutation time is similar for both simulations. This is to be expected as both simulations perform roughly the same calculation, hence the STG needs to build up the same expression graph which would take equally long to reduce (mutation time).

However, the statistics regarding the garbage collector show a clear difference between the two. When the tuple was used significantly more bytes were copied, which meant that the garbage collector spent more time. This fact also leads to the low productivity shown in the last row.

The number of generation 1 garbage collections indicates that the data on the heap has a long lifetime, as shorter lifetime objects would have been cleaned up during a generation 0 collection.

4.4 Thunk buildup

In Section 4.3 we saw that in the case of a space leak, there was a large amount of long-living data causing a larger number of generation 1 garbage collections. This long-living data is the root cause of the long simulation time. When the Int type was used the calculation was reduced every clock cycle due to the strict input of the register. This means that the heap objects representing the calculation were instantly reduced to the computed value.

In the case of the Tuple the *seq* call in the register only forced the tuple to WHNF, which means that it only reduced it to the tuple data constructor and the actual value inside the tuple is not yet evaluated. This causes a build-up of thunks on the heap that can later be used to evaluate the result. This growing structure on the heap can not be cleaned up by the garbage collector and thus it requires an increasing amount of memory. Additionally, because the garbage collector copies the live data in a generation, the runtime overhead of the garbage collection process grows as the live data grows.

4.5 Visualizing the heap

To identify the exact location of the thunks, we can examine the heap visually using the Haskell package `ghc-heap-view`. Because Signals in Clash carry information about their domain (like clock speeds and reset behaviour), the heap is somewhat cluttered when viewing it with `ghc-heap-view`. For this reason, a simplified set of prelude functions with the same problem was used, the definitions of these functions are given in Appendix A.

When simulating a circuit, the output signal is sampled. To sample the output signal the following steps are performed:

1. reduce the signal to WHNF using pattern matching: `x:-xs`.
2. reduce the head `x` to NF and display it to the user.
3. recursively sample the tail `xs`.

After step two is performed we can inspect the tail of this signal to see what is happening on the heap.

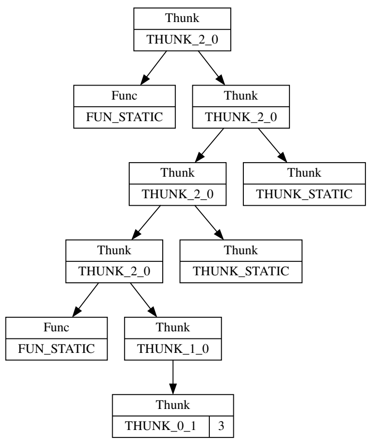
In Figure 4.1 the heap view of the tail of the signal for 4 simulation iterations is shown. It can be seen that the structure of each of the 4 graphs is similar. Each graph has a top-level thunk that points to a static function and another thunk. Essentially this is a function application where one argument (the second thunk) is applied to the static function. This static function is the recursive application of the sample function on the tail of the signal. This means that we are interested in this second thunk which should contain the computation corresponding to the simulation of the future clock cycles.

If we look at this tail, we can see that it is essentially a chain of three thunks with at the bottom another static function. This static function is the "lifted" mux function that takes three signals as an argument:

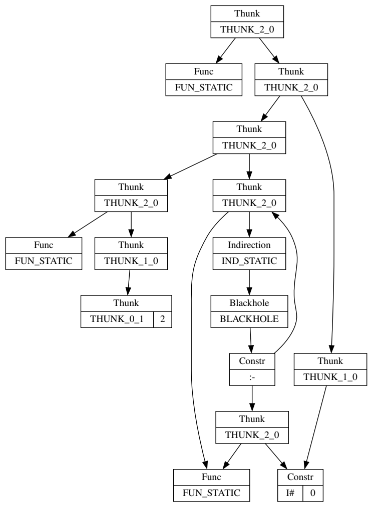
1. The signal with the boolean control signal of the multiplexer
2. The signal that the multiplexer should output if the control signal is True
3. The signal that the multiplexer should output if the control signal is False

4.5.1 First iteration

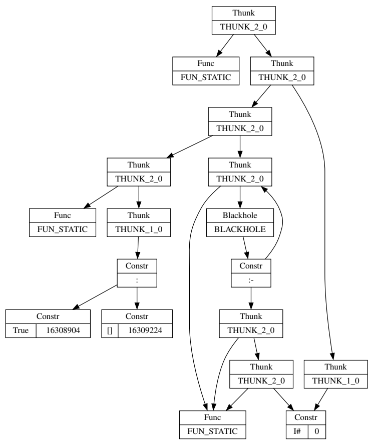
In Figure 4.1a the heap structure for the first simulation iteration is shown. In this case, none of the 3 inputs has yet been explored by the STG, hence they are just shown as thunks. You can see that the first thunk, which also corresponds to the first argument of the lifted mux function, has a reference to another thunk (which is the static value 3). This represents the fact that we used a repeat function to repeat the value *False* 3 times before providing a true value as this argument.



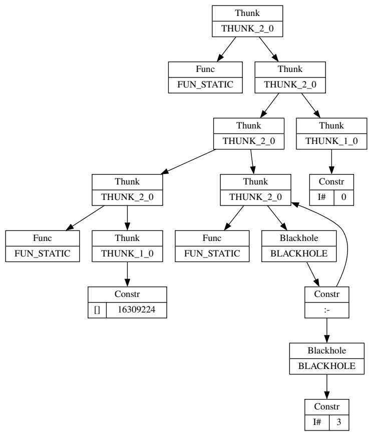
(a) Tail of initial signal, with False as input



(b) Tail after 1 iteration, with False as input



(c) Tail after 2 iterations, with False as input



(d) Tail after 3 iterations, this time with True as an input

Figure 4.1: Heap view of the tail of the signal after the input **[False,False,False,True]**

4.5.2 Second iteration

In Figure 4.1b the second iteration is shown. We again see that the first argument has a reference to a thunk, which this time contains the static value 2. This represents that there are more repetitions of the value *False* left.

The second argument now is expanded to a thunk that references 2 other objects. The first object it references is a static function, this is the (+1) function that we use in our accumulator. The second object is our actual accumulator signal. We can see the recursive nature of the signal implementation by the fact that the signal constructor *:-* forms a cycle with itself. Additionally, we can see that the head of the signal is an application of our (+1) function with the constant integer 0. This shows that the STG has prepared the computation of 0 + 1 on the heap, but it has not actually evaluated it yet.

The third argument is just a thunk referencing the value 0, which represents the value of our output signal if the control signal is false.

4.5.3 Third iteration

In Figure 4.1c the heap structure for the third iteration is shown. The first argument now is a thunk referencing the list `[`**True**`]`, this represents the fact that the next value of the control signal is True.

The second argument is almost identical to that of Figure 4.1b however this time we can see at the bottom of the figure that two thunks refer to the (+1) function. One of them is applied to 0 and the other is applied to the result of the previous application. This is where the space leak starts to appear, essentially this refers to the computation (0 + 1) + 1.

The last argument is identical to the last argument in the second iteration.

4.5.4 Fourth iteration

In Figure 4.1d the heap structure for the third iteration is shown. The first argument is now a thunk referencing the empty list `[]`, this represents the fact that there will be no more input values provided to the control signal.

The second argument is now substantially smaller. The reason for this is that, as we have provided a True this time, the actual computation had to be performed. So now the signal no longer references a chain of thunks that repeatedly apply the function (+1) now it just references the computed value 3.

Chapter 5

Design of potential solutions

5.1 Introduction

In this chapter, we will look at different ways to reduce the impact of space leaks and improve simulation performance. As mentioned in Chapter 1, it is preferred that hardware engineers do not need to modify their hardware descriptions to make it less prone to space leaks. Because of this, the main solution we will look at is how we can modify the prelude in such a way that Clash simulations in general are robust to space leaks.

5.2 Making everything strict

Initially, it may seem that lazy evaluation causes more harm than good in hardware simulation. However, this is not the case, lazy evaluation is what enabled us in the first place to create concise recursive descriptions. Additionally, it only goes wrong if three specific conditions are met:

1. The hardware description contains a recursive computation
2. The result of this computation is only infrequently required for the output of the simulation
3. The computation is performed using a data structure where the WHNF does not require the computation to be performed

If we were to make the entire language strict we eliminate this third condition and the problem can no longer occur. However, this would have large consequences for how hardware descriptions can be expressed. First of all the Signal data type could no longer be implemented as an infinitely long linked list and would require some maximum depth (limiting simulation duration). Additionally, it may also negatively impact performance in certain situations. For example, if a long chain of combinatorial logic is used at the input of a multiplexer, then this would always be evaluated. With lazy evaluation this is not the case, if the control signal of the multiplexer chooses the other branch, then this combinatorial logic does not need to be evaluated at all. For these reasons, it is likely not worth making the Clash language fully strict.

5.3 Selective strictness

If making everything strict is not ideal then maybe there exists some in-between solution where parts of the simulation are strict and other parts are lazy. As described in Section 3.3 this is generally the solution to performance issues with lazy evaluation. However, this is slightly tricky as the problem only occurs in specific designs and adding strictness annotations can have performance implications. It is not desirable that a hardware engineer has to add these annotations themselves as this would clutter the description and requires deep knowledge about the inner workings of Clash simulations. This means that the annotations should be added somewhere in the Clash prelude, but the question is: Where should one add these strictness annotations in the prelude, such that a space leak is no longer possible while minimizing the impact on performance? In Sections 5.3.1 and 5.3.3 we will take a look at several places where we can introduce these strictness annotations.

5.3.1 Registers

As the problem is related to a stateful computation, the state primitive *register* seems like the obvious choice. We have already seen that the input to a register is forced to WHNF, but we can instead force it down to NF. A problem that exists with this solution is that the STG does not know whether a data constructor is already in NF or not. This means that when something is forced to NF the entire data structure has to be recursively traversed. This means that this will always have some performance impact, regardless of the design.

5.3.2 Sample

Another idea is forcing every output to NF, this way all computations required for this output will have to be performed, even if we do not decide to print the simulation output. However, the problem is that a stateful computation is not necessarily required for computing the current output, but can instead be required for computing some future output. This is the case with the simple model presented in Listing 4.3. In fact, the sample function in the Clash prelude already does force every output to NF, and the problem persists.

5.3.3 Data constructors

In Section 5.3.1 it was mentioned that the STG does not know how deeply a data constructor is evaluated. There does exist a way around this with strict data constructors, these data constructors will force the arguments of the constructor to WHNF. This essentially means that a data constructor in WHNF also will have its arguments evaluated. One issue is that hardware engineers also can create custom data types. This however is not a problem as Haskell has a language extension that makes all custom data constructors strict by default.

5.4 Signals in Clash

When we take a look at the heap data structures it is clear that this space leak occurs in the tail of the signal. The idea is then that there may exist an alternative signal implementation that is less prone to this space leak.

5.4.1 Mutable signals

One possibility is to disregard the idea of purity that normally exists in Haskell and use a mutable data structure that changes over time. One obvious problem is that this adds a significant complexity to the implementation, if we want to keep lazy behaviour this will require unsafe Haskell APIs like *unsafeInterleaveST*. Additionally, it does not address the problem, laziness will still exist and this stateful computation still will be stored somewhere on the heap. The only way, in which this could help is if we again add strictness annotations, however in that case there is no clear benefit over using strictness annotations directly, and it only seems to add additional complexity.

5.4.2 Arrowized FRP

As discussed in Section 3.6 arrowized FRP could allow for some interesting optimizations regarding simulation performance. First, the explicit structure definition of Clash circuits makes it less prone to certain space leaks that could occur when using signals as first class values. The main reason for this is that in the *loop* arrow primitive, a binder can be used such that circular structures always have proper sharing behaviour.

However, arrowized FRP does not automatically solve every space leak. The space leak that we showed in Chapter 4 for example, would still be present when using arrowized FRP. Arrowized FRP does make it possible to make the idea of selective strictness presented in Section 5.3.1 a bit smarter. This is because the circuit description would be explicit when using arrows, instead of implicit (due to data dependencies) when using signals as first class values. Using this explicit description of the circuit it would be possible to detect the first of the three conditions mentioned in Section 5.2. Then, only for the registers where these conditions hold, we could make the registers strict. This would allow other registers to remain lazy and thus not have the additional overhead of forcing it every time to NF.

This explicit representation could also allow other performance optimizations. One possibility is the parallelization of the simulation:

Before simulating, the circuit could be inspected to detect parts of the circuit that can be parallelized, this would allow the Clash simulation to run that part in parallel with other parts of the circuit.

Another possibility is to implement checkpointing (saving the state of the simulation, such that it can be resumed later). This can be done by traversing the explicit representation of the circuit to find all registers, and then writing the value of each register to disk whenever the simulation is halted (possibly in conjunction with some type of identifier). Then when the hardware engineer wants to continue the simulation from this checkpoint Clash can again traverse the entire circuit to initialize each register with the value that was stored previously.

Chapter 6

Experiments and results

6.1 Introduction

In this chapter, we will perform Clash simulations using the model shown in Listing 4.3. In Section 6.2 we will perform this simulation without any of the proposed solutions applied to create a baseline of the runtime performance of Clash simulations when a space leak is involved. In Section 6.3 we will perform the simulation with strictness annotations inserted in some of the Clash primitives. The runtime performance of these simulations is then compared to the baseline, to find the effectiveness of the solution.

6.2 Baseline

The goal of this section is to measure the performance impact of the space leak that is present in our model. These baseline measurements can then be used to quantify the improvements of the various solutions proposed in Chapter 5.

6.2.1 Productivity

To gain an idea of how different solutions impact the performance we first present some baseline measurements. We simulate the hardware as shown in Listing 4.3. As the input of the simulation, we first provide a series of False values and then one True value to force the final computation. As we now know, when this sequence of False values is given a space leak builds up on the heap and the productivity will be low due to extensive garbage collector overhead.

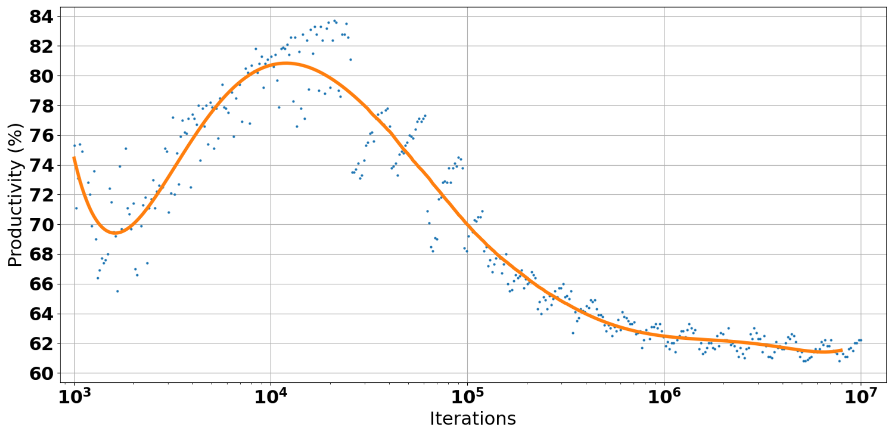


Figure 6.1: Productivity of the accumulator model with a tuple as state

The productivity of this baseline for a range of simulation iterations can be seen in Figure 6.1. With a small number of iterations, the productivity is relatively low (around 70%), this is due to the overhead that exists when initializing the Haskell runtime. Initially, as the number of iterations grows, so does the productivity as the initialization overhead is constant. However, at around 12000 to 13000 iterations it reaches a maximum productivity of around (81%). After this, as the iterations grow further, the garbage collection overhead due to the growing space leak starts becoming significant. This can be seen in the graph as productivity drops to around 62% at 10^7 iterations. This shows the severity of the problem. Even with a relatively simple piece of hardware, the runtime spends almost 40% of the time on tasks like garbage collection.

Next, we show the same simulations but the hardware description contains a larger data type. In this way, we can quantify the impact of forcing large data structures to NF. In this case, we use a 50-element linked list, where there is a stateful computation in the head of this linked list, the other values are the constant 0.

The type that was used for this experiment is `Vec 50 Int` and the computation that is performed is shown in Listing 6.1.

```
vec_add :: Vec 50 Int -> Int -> Vec 50 Int
vec_add (a :> as) b = (a + b :> as)
```

Listing 6.1: Accumulating a value in the head of a linked list

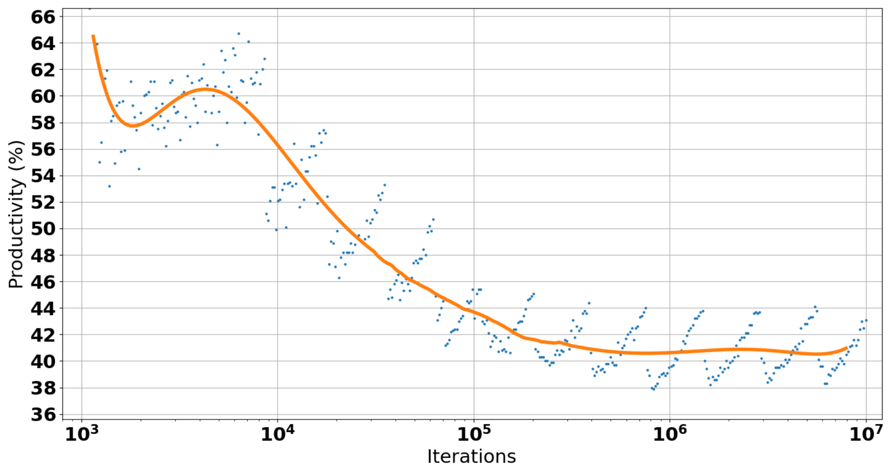


Figure 6.2: Productivity of the accumulator model with a 50-element linked list as state

The productivity of this experiment is displayed in Figure 6.2. The overall shape of the graph is similar to the one seen in Figure 6.1, however the productivity is lower, reaching a maximum of 65% at around 6000 iterations and a minimum of 38% at 10^6 iterations and up.

6.2.2 Mutation Time

While the productivity gives a good indication of whether space leaks exist in the simulation, it is not sufficient to only compare the productivity. This is because a solution that eliminates space leaks entirely could introduce additional runtime overhead. For this reason, we should also look at the actual mutation time (the actual time spent evaluating thunks and mutating the expression graph) of the STG.

In Figure 6.3 the mutation times for both baselines are shown. As expected, the mutation time for both simulations increases linearly with the number of

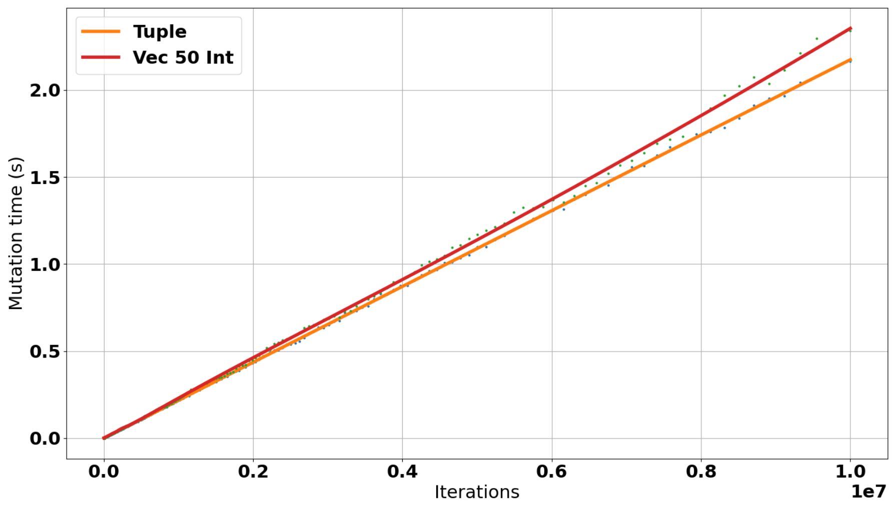


Figure 6.3: Mutation time for both the Tuple baseline measurement and the Vec 50 baseline measurement

simulation iterations. Additionally, the Vec 50 measurement is slightly slower than the Tuple measurement, but in general the timings of the two simulations are quite similar as the same computation is performed in both simulations.

6.3 Strictness annotations

6.3.1 Strict registers

As described in Section 5.3.1 one potential solution is to force values in a register to NF every clock cycle. In this section, we show the same tests as shown in the baseline but this time with a register that forces the input to NF.

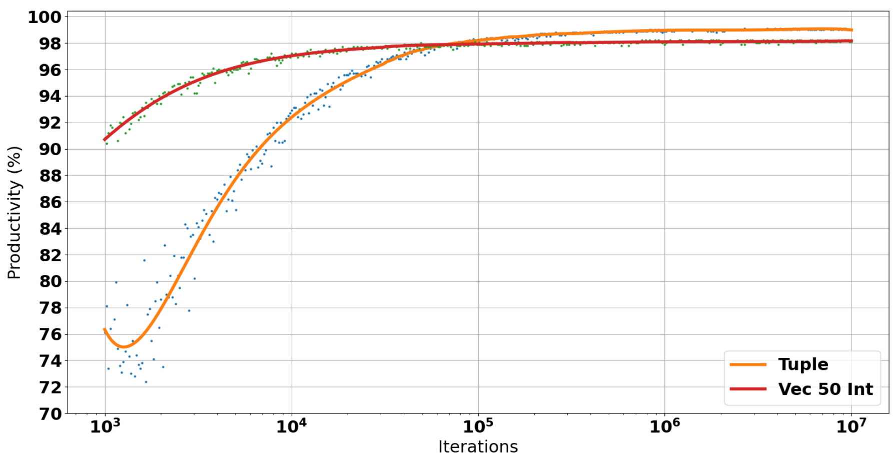


Figure 6.4: Simulation productivity for both the Tuple measurement and the Vec 50 measurement with fully strict registers

The productivities for both tests are shown in Figure 6.4. From these graphs, it is clear that the productivity gets close to 100% after the initial overhead of starting the Haskell RTS. When looking at the *Vec 50 Int* graph, an interesting phenomenon can be seen. The initial overhead seems to be much lower than in the case of the *Tuple* graph. The reason for this is that we are looking at a relative overhead, the absolute overhead of starting the STG is the same. This shows that the mutation time of the *Vec 50 Int* with strict registers test is significantly higher.

The mutation time of both of these tests can be seen in Figure 6.5. This shows a clear downside of this approach, forcing large data structures to NF is expensive. The Vec 50 Int test shows an increase of 10x mutation time when using a fully strict register as opposed to the default more lazy register in Clash.

Even when factoring in the low productivity of the baseline measurements the simulation was faster with the space leak than without. To find the total simulation time the following equation can be used:

$$T_{simulation} = \frac{T_{mutation}}{Productivity}$$

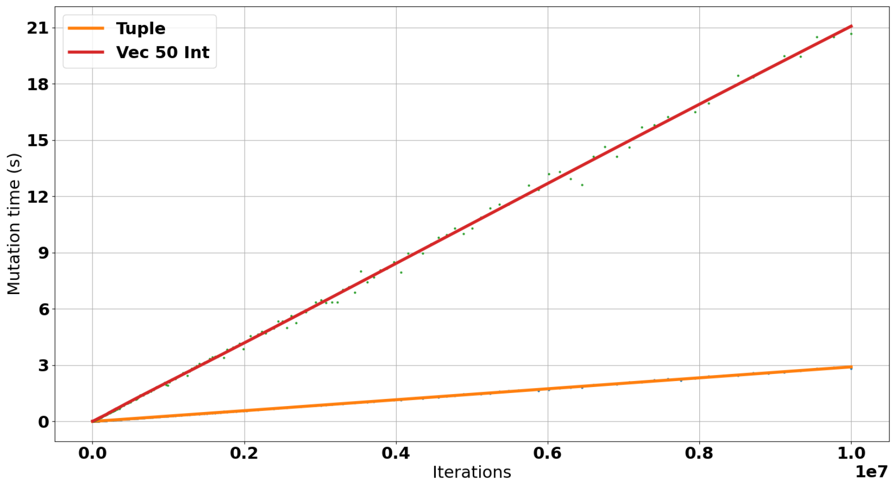


Figure 6.5: Mutation time for both the Tuple measurement and the Vec 50 measurement with fully strict registers

Table 6.1: Simulation timings

	Productivity (%)	Mutation time (s)	Total time (s)
Baseline, Tuple	62.3	2.168	3.481
Baseline, Vec 50 Int	43.1	2.341	5.459
Strict register, Tuple	99.0	2.824	2.852
Strict register, Vec 50 Int	98.1	20.685	21.079

The total simulation times (with 10^7 iterations) for both the baselines and the strict register experiments are summarized in Table 6.1. From this, we can see that for a small data structure, like a 2-element tuple, eliminating the space leak by forcing the value in the register to NF every iteration is beneficial to simulation performance. However, this is not the case for larger data structures like a 50-element linked list, where performance was nearly 4 times worse than with a more lazy register.

Part of this large slowdown when using deepseq in combination with the Vec data type was due to a bug we discovered in the implementation of deepseq for the Vec data type [3]. The implementation of deepseq used a lazy foldl to reduce every element in the vector to NF. The lazy nature of this foldl caused a chain of *rnf* (reduce normal form) calls to build up, before finally evaluating all of these function calls. In the case of the 50-element Vec that was used in our experiments, this meant a chain of 50 rnf calls (which does not contribute to a huge loss in productivity, however for larger vectors it would).

The results of the previous experiments after updating the implementation of deepseq for the Vec data type to use a more strict version of foldl (foldl') is shown in Table 6.2.

Table 6.2: Simulation timings, with updated deepseq for Vec

	Productivity (%)	Mutation time (s)	Total time (s)
Baseline, Tuple	62.3	2.168	3.481
Baseline, Vec 50 Int	43.1	2.341	5.459
Strict register, Tuple	99.0	2.824	2.852
Strict register (foldl), Vec 50 Int	98.1	20.685	21.079
Strict register (foldl'), Vec 50 Int	99.3	8.978	9.037

With this change to the deepseq implementation, the total time of the strict register with a Vec has been reduced by 57%. However, it is still slower than the baseline with a lazy register. As a result, for large data structures, it is still more beneficial to accept a space leak, as opposed to forcing the value in the register every clock cycle to NF.

6.3.2 Strict data constructors

As discussed in Section 5.3.3 it is also possible to make data constructors strict. Doing this results in a push based strictness as opposed to the pull based strictness of deepseq (meaning that the value is forced to NF when a value is updated in the register, as opposed to when it is read from the register). This has the benefit that it allows the traversal of the data structure to be lazy while allowing the data stored inside the data structure to be strict. In the case of our Vec experiment, this results in a strict computation without traversing the remaining 49 elements of the Vec. This is possible because it is sufficient to only make the head of our Vec constructor strict and leave the tail of our Vec constructor lazy. This has the effect that when the data structure is traversed all elements that are reached are also forced to NF. If a computation was building up halfway in the Vec, it would be traversed until that point anyway, as it needs to do so in order to instantiate the computation (even though the result may not be needed yet), and this traversal forces the values that it traverses to NF. This prevents a space leak from occurring, while simultaneously not traversing the parts of the data structure that are not required for that clock cycle.

As we can see from Figure 6.6, the productivity again is near 100% for longer simulations meaning that there is no space leak generating over multiple clock iterations.

As we can see in Table 6.3 this solution produces close to optimal results in this case with a high productivity and no additional mutation time from unnecessarily traversing the data structure. The attentive reader might notice that we have omitted strict tuple data constructors. The reason for this will be discussed in Section 7.2.2.

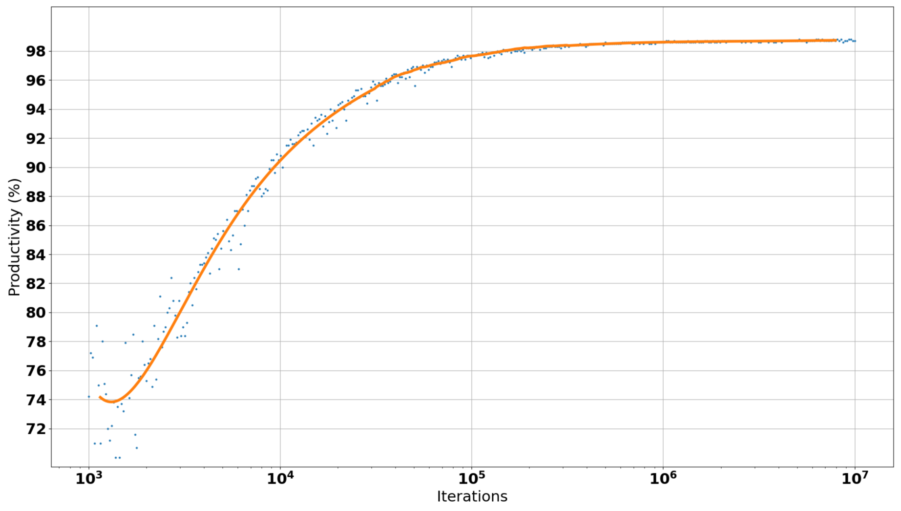


Figure 6.6: Simulation productivity for a 50-element Vec with a data constructor that is strict in the head argument

Table 6.3: Simulation timings, with a strict Vec constructor compared to the baseline

	Productivity (%)	Mutation time (s)	Total time (s)
Baseline, Vec 50 Int	43.1	2.341	5.459
Strict Vec 50 constructor	98.7	2.233	2.262

6.4 Arrowized FRP

As discussed in Section 5.4.2 arrowized FRP creates interesting optimization opportunities for Clash simulations. Unfortunately, due to time constraints, we were unable to experiment with these various ideas. However, we do think that it would be interesting to investigate whether arrowized FRP is beneficial for Clash. In Section 7.4 we will discuss this further.

Chapter 7

Conclusion and recommendations

7.1 Conclusion

While Haskell's lazy evaluation is very effective for hardware design and simulation, it can negatively impact simulation performance for certain hardware designs.

The primary goal of this thesis was to research, and explore solutions to, the problem of space leaks in Haskell within the context of hardware simulation.

The following research questions were presented in Section 1.3:

- To what extent does a space leak in Haskell harm Clash simulation performance?
- Under what conditions can a space leak occur when simulating hardware in Haskell?
- How can we remove or reduce the impact of these space leaks, while minimizing the impact on simulation performance for circuits without space leaks?

To answer the first question we designed a model written in Clash that reproduces a space leak as a result of lazy hardware simulation. Using this model we have found that in certain cases the Haskell runtime spent more than 55% of the time on tasks not directly related to computing the result, but instead on auxiliary tasks like garbage collection.

To answer the second question we identified the properties a hardware design should have for a space leak to occur. We found that three conditions have to be met:

1. The Clash description must contain a recursive computation.
2. The result of this computation is only infrequently required.
3. The computation is performed using a data structure where evaluating the weak head normal form (WHNF) does not require the computation to be performed.

To answer the third question we have looked at different solutions. One solution could be to get rid of laziness entirely and make the simulation fully strict. Although this does prevent the idea of an ever-growing delayed computation, it also removes the nice properties that lazy evaluation had to offer, like infinite recursion in hardware descriptions.

For this reason, we also considered adding strictness annotations to make selective parts of the simulation strict. The most obvious location to do this is in the register function, as this is where stateful computation will occur. When forcing the registers to normal form (NF) every clock cycle, we saw that the space leak that previously existed in the model did indeed vanish. However, we also noticed that, especially for large data types, this forcing to NF added a significant runtime overhead. This caused the simulation to perform nearly 4 times worse than it did when the data was not forced, and the space leak did occur.

We also considered eliminating the third condition by using strict data constructors. When a data constructor that has strict arguments is evaluated to WHNF, the arguments will also be evaluated. We implemented a version of the Vec data type with a strict data constructor and performed the simulation with such a Vec as state. In this case, we noticed that, again, the space leak was eliminated and contrary to when forcing the data to NF in the registers there was no significant impact on runtime performance.

7.2 Discussion

7.2.1 Are space leaks always a problem

In Section 5.2 three conditions were given under which a space leak will occur. The second condition "*The result of this (recursive) computation is only infrequently required for the output of the simulation*" is arguably quite vague. One question that may arise is, how infrequent is too infrequent for a space leak to cause noticeable problems. The productivity graphs of the baseline shown in Figures 6.1 and 6.2 provide some insight. In these graphs, we can see that only after around 10000 iterations the productivity starts to decrease. This means that if we have shorter simulations, the space leak still exists but does not yet cause any significant problems (or rather the initial overhead of starting the Haskell runtime is greater than the problematic effects of the space leak). This raises the question of how frequently certain outputs of hardware remain unused for sufficient long durations to have noticeable negative effects from a space leak. My suspicion is that for a large number of hardware designs no space leak will exist or if it does exist is unlikely to cause any problems. However, if a hardware engineer only simulates part of a circuit instead of the full design a space leak in the part that is not simulated will likely cause problems.

7.2.2 Strict data constructors and tuples

In Section 6.3.2 only the Vec data type was considered without taking a look at tuples. One reason for this is that tuples are used for more than just state in Clash. They are quite frequently used just to pass data around (for example as arguments to a function). When using tuples for data passing, laziness will likely not contribute to space leaks and in this case, is almost always beneficial. This means that if we would introduce strict tuple constructors in Clash, we would also need a supplementary lazy constructor just for data passing purposes. In this case, it makes more sense to not use tuples for state in the first place, as custom data types (as opposed to tuples) likely show better intent of what kind of data is stored in the first place.

Another reason is that there is no flexible way to make tuples strict, this is because the tuple data constructors are special cases handled by the GHC compiler and not affected by the StrictData GHC language extension (an extension that makes all data constructors have strict arguments by default).

7.3 Recommendations

Space leaks can have a detrimental effect on Clash simulation performance. For this reason, we recommend that an effort is made to eliminate these space leaks. Based on the experiments and results shown in Chapter 6 we can conclude that making registers fully strict using *deepseq* causes a significant overhead when larger data types are involved. For this reason, we do not recommend using deepseq on registers as a solution to the space leak problem. Instead using strict data constructors sufficiently address the problem without adding any significant overhead to the simulation. For this reason, we recommend that common data types used for state in the Clash prelude, are modified to contain strictness annotations in appropriate places (e.g. for *Vec* the head of the vector should be strict). Additionally, to address custom (implemented by the hardware engineer) data types we recommend that the *StrictData* language extension is enabled in the Clash starter template. Such that users who base their project on this template will automatically have this extension enabled in their project.

7.4 Future Work

Besides using strict data constructors for Clash simulation, there are also some other potential solutions that need more research before the benefits can be quantified. In particular, there are two interesting concepts that we think are worth researching further for Clash simulation:

- eager/optimistic evaluation (see Section 3.5)
- arrowized FRP (see Section 3.6)

Optimistic evaluation showed promising results on various benchmarks from the Haskell NoFib benchmark suite, however, this does not mean that it also performs well when implemented for Clash simulations. It would be interesting to run Clash simulations using optimistic or eager evaluation to see if Clash simulations also benefit from these evaluation strategies.

In Section 5.4.2 various interesting opportunities for Clash simulation using arrowized FRP were provided. It would be interesting to see whether these opportunities also work in practice. If this is the case it might be a sensible choice for Clash to fully get rid of first class signals and instead use arrows for designing Clash circuits.

Appendix A

Simplified Clash prelude

In Section 4.5 we took a look at the heap graph of a Clash simulation containing a space leak. To make sure the visualization was not too cluttered with irrelevant heap objects, a simplified set of prelude functions was used. In Listing A.1 the definitions of these functions are shown.

```haskell
data Signal a = a :- Signal a deriving (Generic)
instance NFData a => NFData (Signal a)

instance Functor Signal where
  fmap f (x :- xs) = f x :- fmap f xs

instance Applicative Signal where
  pure v = v :- pure v
  (f :- fs) <*> (x :- xs) = f x :- (fs <*> xs)

register :: a -> Signal a -> Signal a
register i s = i :- s

sample :: Signal a -> [a]
sample (x :- xs) = x : sample xs

sampleN :: Int -> Signal a -> [a]
sampleN c = take c . sample

fromList :: [a] -> Signal a
fromList [] = error "Finite list"
fromList (x : xs) = x :- fromList xs
```

Listing A.1: Simplified prelude of Clash primitives

Bibliography

[1] Arrow notation. `https://ghc.gitlab.haskell.org/ghc/doc/users_guide/exts/arrows.html`.

[2] C. Baaij. Cλash : from haskell to hardware, December 2009.

[3] Christiaan Baaij. Rnf and rnfx for vec have a spaceleak · issue #2482 · clash-lang/clash-compiler. `https://github.com/clash-lang/clash-compiler/issues/2482`, May 2023.

[4] M. Bastiaan. Undefined values, how do they work? `https://clash-lang.org/blog/0004-undefined-values/`, 2019.

[5] Zoltán Csörnyei and Gergely Dévai. An introduction to the lambda calculus. In *Central European Functional Programming School*, pages 87–111. Springer Berlin Heidelberg.

[6] Robert Ennals and Simon Peyton Jones. Optimistic evaluation: An adaptive evaluation strategy for non-strict programs. *SIGPLAN Not.*, 38(9):287–298, aug 2003.

[7] Marco Gerards, Christiaan Baaij, Jan Kuper, and Matthijs Kooijman. Higher-order abstraction in hardware descriptions with c?ash. In *2011 14th Euromicro Conference on Digital System Design*, pages 495–502, 2011.

[8] Peter Henderson and James H. Morris. A lazy evaluator. In *Proceedings of the 3rd ACM SIGACT-SIGPLAN symposium on Principles on programming languages - POPL ' 76*. ACM Press, 1976.

[9] Paul Hudak, Antony Courtney, Henrik Nilsson, and John Peterson. Arrows, robots, and functional reactive programming. *Defense*, 2638, 08 2002.

[10] Simon L. Peyton Jones. Implementing lazy functional languages on stock hardware: the spineless tagless g-machine. *Journal of Functional Programming*, 2(2):127–202, April 1992.

[11] Richard B. Kieburtz. The g-machine: A fast, graph-reduction evaluator. In *Functional Programming Languages and Computer Architecture*, pages 400–413. Springer Berlin Heidelberg, 1985.

[12] M. Kooijman. Haskell as a higher order structural hardware description language, December 2009.

[13] Dmitrii Kovanikov. Avoiding space leaks at all costs.

[14] Sumith Kulal, R Ganvir, et al. *Space leaks exploration in Haskell.* PhD thesis, Indian Institute of Technology Bombay Mumbai 400076, India.

[15] Hai Liu and Paul Hudak. Plugging a space leak with an arrow. *Electr. Notes Theor. Comput. Sci.*, 193:29–45, 11 2007.

[16] Jan-Willem Maessen. Eager haskell: Resource-bounded execution yields efficient iteration. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell*, Haskell '02, page 38–50, New York, NY, USA, 2002. Association for Computing Machinery.

[17] Simon Marlow, Alexey Rodriguez Yakushev, and Simon Peyton Jones. Faster laziness using dynamic pointer tagging. *SIGPLAN Not.*, 42(9):277–288, oct 2007.

[18] Neil Mitchell. Leaking space. *Queue*, 11(9):10–23, September 2013.

[19] Henrik Nilsson, Antony Courtney, and John Peterson. Functional reactive programming, continued. *Proceedings of the 2002 ACM SIGPLAN Haskell Workshop*, 10 2002.

[20] Ross Paterson. A new notation for arrows. In *International Conference on Functional Programming*, pages 229–240. ACM Press, September 2001.

[21] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages (Prentice-Hall International Series in Computer Science).* Prentice-Hall, Inc., USA, 1987.

[22] Mary Sheeran. Hardware design and functional programming: a perfect match. *J. UCS*, 11:1135–1158, 01 2005.

[23] Philip Wadler. Fixing some space leaks with a garbage collector. *Software: Practice and Experience*, 17(9):595–608, 1987.

[24] Edward Z. Yang. Space leak zoo, 2011.