# Normalizing the Core Representation in Helium

*Compacting the representation to improve performance*

Reinier Remco Maas

MSc Thesis (ICA-4131495)

July 24, 2019

Universiteit Utrecht

Center for Software Technology
Dept. of Information and Computing Sciences
Utrecht University
Utrecht, the Netherlands

*First supervisor:*
dr. J. Hage
*Second supervisor:*
dr. A. Serrano

## Abstract

This research started as an attempt to implement a heap recycling optimization in the Helium compiler. In the process, we identified various obstacles, which turned out to be bigger than we expected. While the end goal of this research was not achieved, we have brought formerly uncharted territory into view. Based upon this knowledge, we present a solid base upon which future investigation can continue in order to reach the goal of a heap recycling optimization in Helium. We focussed on improving the core representation. Therefore we measured and report the code generation improvements that we obtained with this work.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

This research attempts to make Helium an optimizing compiler. We are going to optimize Haskell, which is a functional language (see Section 1.1). We need a compiler to create a program from source-code (see Section 1.2). We introduce the type system (see Section 1.3) necessary for analyzing (see Section 1.4) the programs. Finally we discuss the research question answered in this work (see Section 1.5). We present a solid base upon which future investigation can continue.

## 1.1 Context: Functional Languages

Through a programming language a programmer instructs the computer to do a specific job. Different programming paradigms (see Section 2.1) have different trade-offs. The advantage of functional languages is that the programmer describes what they want instead of how they want it. This results in high-level code that is readable in the context of the problem. The disadvantage of functional programming is that the language concepts have a mismatch with the underlying hardware. Because of this mismatch it is harder to write highly performant code. When translating this high level language to machine code analyses (see Section 3.1) can regain some of the performance lost due to the abstractions.

### 1.1.1 Haskell

Haskell is a lazy functional language (see Section 2.3) this means that is uses lazy-evaluation. Lazy-evaluation is a strategy that can give performance improvements; this is because a value that is never needed is never evaluated. On the other hand if a value is needed the expression is evaluated and the thunk in which that computation was stored is updated with the answer; this leads to sharing and will avoid repeating the calculation when it is needed in the future.

Lazy evaluation is only safe when the language is pure (see Section 2.2). If a language is pure, a function called with a set of arguments always returns the same answer; this introduces another possible optimization: memoization. Memoization is the caching of the result of a function call based on the input parameters.

Memoization uses lookup tables a specialized form of caching that stores the previous function calls with the resulting answers, which can be an unevaluated thunk which is then consequently shared.

One of the harder problems with a lazy functional language is that it is hard to predict how much time and memory a function will use at runtime (see Section 2.5). Under a strict evaluation model it is more predictable how much is calculated at every point in the program. There are of course always trade-offs between space and time consumption. Assigning more memory to a Haskell program has a few effects: the memory can be used to memoize more, the pressure on the garbage collector gets reduced and the garbage collection cycles take more time as more

memory needs to be scanned and collected per cycle. When tuning Haskell programs there is an optimum when choosing the amount of memory to assign.

## 1.2 Context: Compilers

When a language is compiled by a compiler it is effectively translated to an architecture. Most compilers (see Section 2.5) target specific machine code, which can be either a virtual machine or directly target the hardware with assembly. Other compilers, also called transpilers, target other programming languages with possibly the same abstraction level as the original programming language.

Compilers consist of three stages, these stages are front-end, middle-end and back-end. The individual stages can consist of multiple phases. The front-end stage is responsible to get the source-code into the Intermediate Representation, or IR, that can be used by the middle-end. This is performed in the following steps: first, lexing the source-code to a stream of lexemes, then the stream of lexemes is parsed into an Abstract Syntax Tree, or AST, that represents the entire program. After the AST that contains all the language constructs is generated it is desugared into the IR which has a simplified AST. The middle-end type checks and analyses the IR. With this information the middle-end can perform machine independent optimizations on the IR. The middle-end passes the optimized IR and the corresponding analysis, that can be used for machine dependent optimizations, to the back-end. The back-end gets the optimized IR and the information from the static analyses that were performed and can perform machine dependent optimizations and then produce the machine- or byte-code that represents the program.

Some programming languages that give type error diagnosis only type check their intermediate representation. This results in sub-optimal type error messages; compilers tend to give better type error diagnosis if they perform type checking on the abstract syntax tree of the entire language. The errors given can then refer to the actual constructs that the programmer used. Helium, GHC and OCaml use the full language to type check in order to generate good error messages. After type checking it is lowered into the intermediate representation.

### 1.2.1 Helium

Helium is an implementation of a Haskell compiler; Helium is focused on generating the best error messages. It generates these error messages through the TOP framework. This framework first generates constraints from the AST that contains all the language constructs and the actual location where the parsed code came from: the file, line and column are known for each node in the AST. These constraints are solved by a separate solver. There are multiple solvers to choose from.

First there is a greedy solver that is optimistic in that it assumes that there are no type errors. Because of its optimism it is very quick in solving type errorless constraints, when it runs into a type error another solver is activated. This is the type graph solver which adds all the relevant constraints for the type error to a type graph, which can be type inconsistent, and only when all constraints are added starts to investigate the type error. In order to be able to provide the best in class type error messages these type graphs are introduced [18, 15, 11, 14, 10].

This second solver can report multiple locations that contribute to the type error. The heuristics choose the locations that are displayed and the explanation of the encountered error.

This ensures that the programmer gets a useful error message. More solvers can be build and used to test different experimental solver implementations.

### 1.2.2 LVM Machine

Currently the Helium compiler uses the Lazy Value Machine, or LVM, as its back-end (see Section 2.5.6). This back-end is capable of running LVM-assembly, which is a lazy functional language. During execution the LVM machine checks for various runtime errors such as overflows and unbounded recursion. When an exception is raised by a runtime check, such as a pattern

match failure, the LVM machine shows a *stack*-trace. This is actually a trace of demands that were executed and resulted in the error. Because the LVM-code is lazy the only way to start executing code is when there is a demand for that execution. Programmers get better runtime errors because of this trace, and the resulting messages are closely related to the code the programmers actually wrote.

## 1.3 Context: Type System

Type-and-effect systems enhance the underlying type system with additional effects. The additional effects describe part of the runtime behavior of the program. In short they try to model the dynamic behavior of the program. Counting analysis includes many effect analyses as a special case (see Section 1.4). Strictness, absence, sharing and uniqueness analysis are special cases of the counting analysis described by Verstoep and Hage [34]. The additional types describe static behavior of the program. With these additional annotations more information about the total behavior of the program is available to the compiler. These analyses results can be used to inform the optimizer of possible optimizations.

## 1.4 Context: Analyses

Strictness, absence and sharing (see Subsections 1.4.1, 1.4.2 & 1.4.3) analysis can be used to increase the performance of the runtime. A special application is reusing allocations on the heap, therefore called Heap Recycling (see Section 1.4.4). This needs an annotation to be introduced in the surface language. The annotation is used by the programmer and checked by the compiler.

Further explanation of these analyses are described in Section 3.1.

### 1.4.1 Strictness Analysis

Strictness analysis [19, 8, 36, 3], further described in Section 3.1.1, can determine whether evaluating an expression strictly will change the runtime behavior. If strictly evaluating the value does not change the behavior, the computed value can be passed to a function instead of creating a thunk. Any closure could possibly refer to more thunks on the heap. These thunks are now also evaluated on a call-by-need basis because of this analysis. They are either updated or no longer referenced and can be collected by the garbage collector.

A thunk is a specialized closure that accepts no further arguments but only awaits execution. If all references to the thunk are dropped without evaluating the computation it can be collected by the garbage collector. Not calculating unnecessary values is one of the main advantages of a lazy language (see Section 2.3).

### 1.4.2 Absence Analysis

Absence analysis [31, 37, 2], further described in Section 3.1.5, indicates whether an expression is used at all. These analysis results exhibits similar optimization opportunities as dead or unreachable code elimination would perform. The strength of absence analysis lies in analyzing that only a part of an argument is used. Computing the length of a list only uses the spine of the provided list for example but not the elements. There exist a lot of functions that only use parts their arguments.

### 1.4.3 Sharing Analysis

Sharing analysis [13, 23, 9], further described in Section 3.1.2, tells us whether a thunk that is evaluated needs to be updated to share the value with other computations that are done in the program; this is only necessary if the computation is shared. If a value is not shared after evaluation there is no need to update the thunk, indeed it would even be possible to reuse the

```
1 reverse :: [a] -> [a]
2 reverse l = rev l []
3     where
4     rev :: [a] -> [a] -> [a]
5     rev [] acc = acc
6     rev r@(x:xs) acc = rev xs r@(x:acc) -- heap allocation of 'r' is reused
```

Figure 1.1: Reverse Function: with Recycling Annotations

memory that is taken up by this specific heap allocation. In the paper by Hage and Holdermans [12] the programmer can make the decision whether his function would profit from a specialized implementation with heap recycling semantics (see Section 1.4.4).

### 1.4.4 Heap Recycling Analysis

With the knowledge gained from sharing analysis the generated code can be optimized to use heap recycling. By reusing memory for the return argument of a function for example. Normally one would request a location to store the newly generated data on the heap and the garbage collector would take care of cleaning up the previous allocation. When one knows that the memory would become garbage the memory location could be reused in order to relieve pressure on the garbage collector.

In the paper by Hage and Holdermans [12] there are some examples of code that can be specialized in this manner. We introduce the @ symbol on the right-hand side, to indicate reuse of that specific data constructor. The reverse function in Figure 1.1 can reuse the spine of the list for example if the argument list is used uniquely.

## 1.5 Research Questions

In this thesis, the main research question we aim to answer is:

**Does how we represent Helium programs in core affect the performance?**

Normalizations and optimizations in the compiler restructure the core. We hope that the updated representations increase the performance of our programs.

In order to confirm that any of our optimizations to core lead to a performance gain, in terms of code size, memory use, or plain performance, we need to establish a ground truth. Therefore our first research question refers to a baseline of the measurements we are going to take.

**What is a good baseline measurement for the Helium compiler?**

Our second research question refers to what we are going to measure. We need a testing framework in order to get reliable results. We will look into benchmark suites used by other haskell compilers.

**What is a good benchmarking suite for the Helium compiler?**

Our third research question refers to which normalizations can be created. For this we will research different core representations from different Haskell compiler. This will give us an insight into the possible normalizations that are performed on the core representations. We will also look into the core representation of the Helium compiler. How this is used will probably present other normalizations as well.

**Which normalizations will compact the core representation?**

Our last research question refers to typing core. The types that are reported by this type-system can be compared to the types generated by the TOP typing framework. With this new type-system we could discover type inconsistencies introduced in the code generation and normalization phases.

**How to ensure no type differences are introduced by normalizing core?**

In the future more optimizations can be added to Helium compiler. Currently there is a push to introduce a type-and-effect system for Helium-core. Creating an extendable constraint

based type-system makes it possible to add the effect analysis later into the type-system. This type-and-effect system may be used to do counting analysis. With the analysis results generated by counting analysis Heap Recycling becomes a possibility.

# Chapter 2

# Preliminaries

In programming languages a programmer can write programs. A programming language consists of expressions, declarations and statements. Together these are the syntax of the language. What these mean is defined by the semantics. The dynamic semantics describe what the runtime behavior of the program is, whereas the static semantics describe an approximation of the runtime behavior that can be made at compile-time. The type system is the most noteworthy part of the static semantics. Another way that the static semantics show themselves is through error messages returned by the compiler. As the static semantics may prohibit programs that would result in runtime errors, the programmer can be given statically known runtime information before even running the actual program. Type checking is a verifying analysis, based on the static semantics, that results in fewer runtime errors for a program.

## 2.1 Functional and Imperative Languages

Imperative programming languages enforce that the programmer writes exactly *how* the computer should perform the computation. With functional programming languages the programmer describes *what* should be performed rather than *how* it should be performed. This can result in highly reusable code. This way functional programming facilitates code-reuse. Haskell [20] and Ocaml [38] are functional languages for example.

## 2.2 Pure and Impure Languages

Impure languages can have a global state that effects the runtime behavior of functions. In an impure language we can write pure functions by restraining ourselves from changing global state and side-effects and only relying on the input that is provided through the function call.

In pure languages, code lives in a world where there exist no uncontrolled side-effects. A language that doesn't exhibit side-effects is quite useless. The pure languages are extended with restricted side-effects explained in Section 2.4. A pure language enforces us to separate changes to the global state and side-effects from our functions.

## 2.3 Lazy and Strict Languages

Laziness in languages indicates that the program is waiting as long as possible before evaluation. A closure represents an unevaluated function and the free variables from the enclosing scope where it was created. A closure may be shared between multiple call sites. When a closure is provided with all the needed arguments but the result is not yet demanded it is called a thunk. A thunk contains either an unevaluated closure or the result after evaluation. If an expression gets evaluated the result of the evaluation, the value, is written over the thunks memory. Future

requests for the value can thus be answered instantly with this already computed answer; this is called *sharing*.

Only evaluating expressions when their values are actually needed can result in a speedup when executing code. This speedup can be achieved because the expression is *not* evaluated at all when it is *not* needed. When choosing a branch where the result of an expression is not needed at all, the closure is never evaluated.

In case the result of an expression is needed in multiple places, sharing comes to the rescue. It distributes the computed value to all the locations where it is needed. Due to this sharing the expression only needs to be evaluated once.

Lazy or non-strict languages evaluate expressions following a call-by-need evaluation strategy. Strict languages evaluate on a call-by-value strategy: they first reduce the arguments to a function to their normal forms before passing these values to the function call. The order of evaluation during the execution is thereby inverted. A non-strict language executes because it needs the outer value. There is a demand for that value, that can consequently need inner expressions to be evaluated. However, sub-expressions that are not needed are not evaluated. Because a strict language always evaluates function arguments the programs written in the strict language can encounter infinite loops and errors that a non-strict language would not even evaluate. A non-strict language would delay the computation of *unused* arguments until they are no longer needed.

Laziness effectively increases the performance with sharing and non-strict evaluation. These optimizations are diminished because there is an overhead due to creating thunks on the heap for every unevaluated expression and updating them with their values after evaluation.

Strictness is a way to overcome this overhead while keeping the benefits of laziness. Strict evaluation ensures that expressions, that are known to be needed, are evaluated to weak head normal form. After being evaluated to whnf they are passed on, possibly even on the stack. But not everything can be evaluated strictly. What can be evaluated strictly depends on the semantics of the language.

## 2.4 Side-Effects

Most programming languages allow unrestricted side-effects to happen inside a function. Some programming languages restrict side-effects. For example Haskell uses the IO Monad (see Section 2.4.1), Clean uses uniqueness typing (see Section 2.4.2) and Rust uses ownership (see Section 2.4.3). Whether or not the restrictions are made visible in the type system or during compile-time errors depends on the approach. The following subsections describe of the different approaches.

### 2.4.1 IO Monad

Haskell, described by Jones [20], uses the IO Monad to abstract over I/O. Because Haskell is a pure lazy language and the order of the calls is *undefined* in a lazy language the IO Monad is used. The IO system is not allowed to break the purity, therefore the IO Monad sequentialises the IO operations. Monads represent sequential computations, that are chained together.

Purity ensures that we can assume that results of functions called with the same arguments can be cached. The result of the previous call can be returned without the computation. These restrictions limit side-effects thereby introducing a greater possibility for optimizations and verifications. For side-effects these restrictions are limiting. When you write to a file you want to ensure that $x$ is written before $y$ is. All this is encapsulated inside a Monad without losing referential transparency.

### 2.4.2 Uniqueness Typing

Uniqueness typing is used by Clean, described by Plasmeijer and van Eekelen [29], to ensure that a unique value has only one reference to it. The unique types are used to enforce single-threadedness, therefore it makes side effects semantically unproblematic. With uniqueness typing

the ability of destructive updating data-structures is brought into a pure functional language. I/O is handled in the same way as doing a destructive update. When a destructive update is executed on a specific file handle a new, or the same, unique handle is returned. Uniqueness guarantees that no other references exist to the value.

Uniqueness typing can lead to an efficient interface between the functional language and the non-functional world or the non-functional implementation details. An example of such an implementation detail is in-place updates.

Heap recycling is going to use a uniqueness analysis to determine whether an in-place update function can be called instead of the default allocating function.

### 2.4.3 Ownership

Rust's ownership, described by Matsakis and Klock II [25], dictates that there exists only one owner who is responsible for deallocating a particular piece of memory. This deallocation happens when the owner falls out of scope. Mutable borrows exist but during the mutable borrow all other references (the previous mutable borrow and the one held by the owner) are invalidated for the lifetime of the mutual borrow. Having a single valid path to mutable data ensures that there can only be one writer at any time. Shared immutable borrows also exist. Instead of having only one active path to the immutable data there can exist multiple paths and there can be many readers active on the same data at the same time. If there exist multiple active paths the data must be immutable. Therefore no side effects are visible to the rest of the program.

Uniqueness can be seen as a restricted form of ownership. Where uniqueness ensures that a variable is always used uniquely, ownership ensures that a variable is used uniquely when that variable is mutable, however when a variable is immutable it can be freely shared. The lifetimes of immutable borrows need to *end* before the variable can be mutated again.

## 2.5 Offline, Ahead-Of-Time or Just-In-Time

Compilers have different names depending on the moment the compilation is performed.

The standard way of compiling is offline compilation. If the code is compiled offline it is distributed as a binary or as byte-code for a virtual machine.

An alternative is compiling on the machine that is going to run the code. Source-code or byte-code distributions can be compiled on the machine that is actually going to run the resulting code. Compiling after distribution but before execution is called Ahead-Of-Time, AOT, compiling. Compiling during the program run is called Just-In-Time, JIT, compiling. Directly running source-code can be done in an interpreter. Because of the moment and the information available different compilers can use different analyses, an offline or ahead-of-time compiler must use static analyses while a just-in-time compiler can also use dynamic analyses.

The time, resources and runtime information available for the compilation depend on when and where it is performed. More time and resources for the compilation are available if this happens before distribution in offline compilers (see Section 2.5.1). When performing AOT-compiling (see Section 2.5.2) the system is statically known. When performing JIT-compilation (see Section 2.5.3) the actual runtime characteristics of the code can be taken into account. An interpreter (see Section 2.5.4) directly executes byte-code in an evaluation loop.

### 2.5.1 Offline Compiler

Offline compilers compile the source code and the generated binaries are distributed; these compilers have more time to optimize the machine code but less runtime information, i.e. is this new instruction set supported or how wide is the SIMD[1] -bus.

[1] SIMD stands for "Single Instruction Multiple Data". A single vector instruction operates on multiple data elements at the same time. Currently the SIMD registers in consumer-grade hardware are 256 bits wide. Giving the SIMD instruction the possibility for $8 \cdot 32$ bits operations or $4 \cdot 64$ bits operations. Double the width, 512 bits, is available on server-grade hardware. This results in being able to execute 16 parallel data operations.

Nuzman et al. [27] describe splitting the heavy lifting of auto vectorization to an offline compiler. They leave the execution details to the JIT compiler which has more information about the system on which the code is running an can thus decide on the width of the SIMD instructions to generate.

Machine-agnostic optimization can be performed when generating the binaries or byte-code before distribution. An offline compiler can spend much more time on machine-specific optimizations if it knows what it is targeting. When machine-specific optimizations are performed for a certain platform that binary cannot be run on another target anymore. For each supported platform another binary distribution optimized for that platform can be provided. In order to also incorporate runtime characteristics into the optimizations, that are performed by an offline compiler, we can use profile-guided optimizations. There is just one caveat with profile-guided optimizations, namely that used profile must be a statistical representation of the actual usage of a program.

A well known language that uses an offline compiler is the C language.

### 2.5.2 Ahead-Of-Time Compiler

Ahead-Of-Time, AOT, compilers [28] have more information about the system that the code will be run on, but less time and resources are available to perform optimizations. They compile (an intermediate representation of) the code down to machine instructions and do not interact with the running code. They have a restricted amount of time for optimizations due to the startup lag it would incur. Another option is to use persistent storage where the compiled code is cached. With the update of the AOT compiler on the users platform the cache is invalidated and recompiled. Recompilation can happen directly when the compiler is updated or before running the code for the next time. With this recompilation new optimizations will take effect without redistribution of the program code. An AOT-compiler can use machine specific optimizations because it is performed on the device where the actual code will be run.

One of the most widely spread uses of an AOT compiler is ART, the Android RunTime, it precedes the Dalvik runtime (see Section 2.5.3). This runtime decreased power usage on mobile devices by compiling the byte-code only once.

### 2.5.3 Just-In-Time Compiler

Just-In-Time, JIT, compilers [22] have the most information about the system and can aggressively optimize the code based on its runtime characteristics, when optimizing and/or reoptimizing the actual code. A JIT compiler is sharing time and resources with the actual application, thus even less time and resources are available to perform the actual optimizations. All information is destroyed on exit of the program, and needs to be recollected, recompiled and reoptimized once the application is rerun the next time. A JIT-compiler can use machine and runtime specific optimizations as it is performed during the runtime of the actual program.

Before Android received its AOT compiler it used Dalvik[2] with a trace-based JIT. A trace that was executed frequently was compiled into native machine-code. The remaining byte-code was interpreted.

### 2.5.4 Interpreter

An interpreter is a program that directly evaluates the program that is provided. A program can refer to the source-code or to the byte-code that was created from the source-code by an offline compiler. An interpreter has an evaluation loop that executes the instructions directly. The instructions are not compiled to machine-code but run in this virtual machine.

Lua[3] is an example of a fast interpreted language. It has a small runtime, which is designed to be embedded in larger applications. Lua is the most used embedded scripting language for games.

[2] `https://source.android.com/devices/tech/dalvik/`
[3] `https://www.lua.org/`

### 2.5.5 Runtime

When offline compilers generate byte-code, there are more steps needed in order to actually run the code. When the byte-code arrives on the machine it is going to run on, it needs to be compiled to machine-code. This compiler that compiles byte-code is referred to as runtime for these languages.

We take C# as an example pipeline. In the C# pipeline there exist multiple ways to generate machine-code. The first step is always the Roslyn[4] compiler, which compiles the surface language into CIL, the Common Intermediate Language.

Now there exist two options to compile CIL into native machine instructions. Namely the .NET runtime named CLR, which stands for Common Language Runtime[5], and NGen, which is the Native Image Generator[6].

CLR is an JIT compiler and is the default way to run CIL byte-code. There are some cases where the CLR is faster than the NGen. Namely the trust level is known when the JIT compiler is run, while the AOT compiler is unable to assume the level of trust that is granted. For this reason not all programs that are compiled Ahead-Of-Time result in a faster runtime. One of the solutions for the AOT compiler is to assume the highest trust level and otherwise resort to the JIT compiler.

NGen is an AOT compiler and runs when someone installs software on his or her computer. NGen generates a native image from the CIL. This native image is stored in the NIC, this is the Native Image Cache. Storing the generated assembly exchanges runtime overhead with storage costs.

### 2.5.6 Helium

We are researching how we can create an optimizing compiler from Helium, a Haskell compiler. It is important to realise where in the chain of compilers and compiler phases we are working. Which information is available at which moment can clearly influence the design a lot.

The Haskell compiler is an offline compiler that produces lvm-byte-code. There is the core generation phase which generates the core representation from the source-code. This phase does the expansion of the different language constructs. After the core is generated the next phase generates lvm-assembly. On the assembly there is a peephole[7] optimization pass.

The Haskell compiler has an accompanying lvm-byte-code interpreter, called LVM Runtime (see Section 1.2.2). This byte-code interpreter is invoked by "runhelium". It has some low level optimizations such as a look-ahead strategy where heap allocation is prevented if the value is instantly destructed, i.e. by a match. Generating machine-code from the byte-code would upgrade the interpreter to a just-in-time compiler with possible speed benefits.

Counting analysis is a complex analysis and will therefore cost a lot of time. Due to the time constraints in an interpreter it is better to perform this analysis in the offline compiler. The exact location chosen for this optimization is on the core representation directly after core generation.

[4] `https://www.github.com/dotnet/roslyn`

[5] `https://docs.microsoft.com/en-us/dotnet/standard/clr`

[6] `https://docs.microsoft.com/en-us/dotnet/framework/tools/ngen-exe-native-image-generator`

[7] A peephole is a window into the stream of assembly-code. Common patterns are matched in the stream and replaced by faster assembly instructions. This preserves the result of the functions.

# Chapter 3

# Related Work

The related work chapter has two main sections the analyses in Section 3.1 and Intermediate languages in Section 3.2.

## 3.1 Analyses

### 3.1.1 Strictness Analysis

There are multiple ways of adding strictness, see Section 2.3, to a language: either to allow the programmer to add strictness annotations to the language, or by analyzing the code, letting the compiler figure out what can be executed strictly without changing the semantics.

When the programmer is required to annotate his entire program with strictness annotations he is effectively required to analyse the entire source code and decide what parts of the program are always used. This analysis is repetitive work where it is easy to make a mistake by placing too many annotations. Annotations placed by the programmer can not be checked for correctness as the semantics of the written code changes when these annotations are placed.

Placing too many annotations could result in decreased performance, because something that is not used does not need to be evaluated. Evaluating something that does not need to be evaluated can even lead to non-termination. It would therefore be better to let a static analyser do the lion's share of the work. The analyser has to be conservative with respect to the unboxing of values, unboxing an undefined value will result in a runtime error where this error would be absent without this added strictness. To overcome the conservativeness of the analyser for the programmer it is still possible to add a few annotations.

By analyzing the code the compiler makes expressions strict if their values are guaranteed to be used. When the encapsulating expression is not strictly used in all code paths the normal non-strict behavior is exhibited. This way the dynamic behavior of the code is not changed.

If the programmer annotates their programs there is a chance that dynamic behavior of the program gets changed. Holdermans and Hage [19] take the annotations made by the programmer into account when analyzing the code. In their research they made sure to include the different semantics that are introduced when programmers selectively make their functions stricter. The authors introduce the principle of *Applicativeness* as "functions that are guaranteed to be applied to arguments". This means that if an argument is relevant to the body of a function and the function is guaranteed to be applied to all its arguments then the argument is guaranteed to be evaluated.

### 3.1.2 Sharing Analysis

Gustavsson [9] describes sharing analysis for update avoidance and optimizations. Because of the dynamic semantics of sharing, the runtime is required to overwrite a thunk with the value after evaluation of the expression. This behavior is needed because the runtime doesn't know

```
1 map :: (a -> b) -> [a] -> [b]
2 map _ []     = []
3 map f (x:xs) = f x : map f xs
```

Figure 3.1: Map Function

when this value is shared. With sharing analysis it can be proved that this is the only reference to the thunk with possible speed and memory benefits. The thunk is not overwritten with the value as this would be an unnecessary step.

There are a few things that could speed up the running characteristics of the code. It is possible to reuse the memory if the same amount of space is needed on the heap. This can happen when a constructor gets matched and a function is executed on the element. For example in map in Figure 3.1 the same constructor, the Cons (:) or the Nil ([]), gets destroyed and created. The memory allocator is required to find the same amount of unused memory and allocate it. Later the old constructors memory, that now has become garbage, would be found lying around and will be collected by the garbage collector. By reusing the matched constructor's memory this allocation and consequently garbage generation can be combined.

Reusing the memory of the constructor for the newly created constructor ensures that there is no interaction with the heap management. When reusing memory is not possible, the memory will still become garbage but no allocation of the same size has been made.

Updating memory that is no longer referenced is unnecessary, because it can never be accessed again. The memory has become garbage so it can be collected. This is normally done by the garbage collector during a garbage collection cycle. The garbage collector could exhibit an API to let the code inform it of the memory that can be freed. Then that memory could be freed without a collection cycle. The signal will consequently result in fewer garbage collection cycles during the execution of the code because the garbage collector will have less memory pressure that initiates the garbage collection cycles.

### 3.1.3 Usage Analysis

Hage et al. [13] developed a generic usage analysis. This analysis, based on qualified types, can be instantiated to either sharing or uniqueness typing. Both of these analyses count the number of times values are used. Partial application needs to be analysed as well. To handle this case containment is introduced. Containment is stated as follows: "if a value is contained within a structure, we must assume that it is used at least as often as the containing structure."

Sergey et al. [30] implemented a usage analysis for the GHC compiler. Gill [7] introduced a compiler hack to make short-cut deforestation work, the implemented usage analysis supersedes this compiler hack and resulted in an overall increase in performance. Call demands are newly introduced in this paper. The optimizations that are made possible with the analysis include one-shot lambdas, removing the update frame for one-shot thunks and removing unused thunks completely. Missed optimization opportunities are due to thunks being saved inside constructors (e.g. tuples, lists and arrays). This is specifically for constructors that are returned from function calls when the demand for the result is not known.

### 3.1.4 Uniqueness Analysis

Hage and Holdermans [12] enabled the programmer to recycle allocated memory. This is done by adding a "destructive assignment operator" which is only applicable if the allocated memory is no longer in use. The programmer gains the ability to performance tune code without having to sacrifice a functional style of programming. This is a form of "compile-time garbage collection". Heap cells that are allocated but no longer in use can be repurposed through explicit control. With the marker it is possible to write in-place algorithms while staying close to the idiomatic functional style. When adding such a marker, a language designer has to ensure that referential

$$\pi \quad = \quad 0 \quad | \quad 1 \quad | \quad \infty$$

Figure 3.2: Annotation Primitives

transparency is not affected. This is done with uniqueness analysis of the program. To ensure that the available memory is of exactly the same form as the needed memory, the constructors are required to match.

Uniqueness typing ensures that a value that is required to be used uniquely is in effect used uniquely. Uniqueness analysis doesn't unlock speed improvements for the compiler but instead restricts the number of compiling programs further; this applies for instance also to file handles. File handles are supposed not to be duplicated. To guarantee that they are used only once we could run a uniqueness analysis.

When creating functions with the "destructive assignment operator" the compiler has a few options. It could produce error messages when it finds out that the programmer tries to reuse the memory multiple times. Or it could generate a unspecialized function for these cases.

### 3.1.5 Absence Analysis

Absence analysis or dead code analysis can benefit the runtime of a program by removing any unused code from the binary. It is not often that the result of a computation isn't used. Nearly always the answer of a computation is at least partially used. Helper functions for a tree data structures could for example calculate the depth of a tree, when this happens only the spine of the data structure is used. Values that are stored in these data structures could remain unused for the rest of the program and thus subsequently removed from the binary. Especially generated code can benefit a lot from such an analysis. Generated code contains a lot of dead code that can be analysed away.

### 3.1.6 Counting Analysis

Verstoep [35] and Sergey et al. [30] combine different analyses into one analysis, what they called a counting analysis. The analysis tracks the never used for the absence analysis, at most used once for sharing and uniqueness analyses and at least used once for strictness analysis. Combining the different analyses in a single analysis has certain benefits. The performance of running a single analysis is higher than running a multitude and the analyses can support each other. Maintenance of a single analysis is easier and similarities are made explicit. The analysis described uses a type-and-effect system with constraints. There are two phases, one constraint generation phase and a constraint solving phase. The analyses that are captured by the described method can be used to accept and or reject programs the so called verifying analyses or derive more information which the compiler can use for optimizations and transformations of the program.

Annotation primitives used by the counting analyses describe how often something is used or demanded. Instead of choosing the infinite range of natural numbers the lattice is chosen in Figure 3.2. The counting analyses do not require a larger precision. $0$ indicates that it is not used at all while $1$ indicates exactly one use whereas $\infty$ indicates at least two usages. By using sets of annotation primitives it is possible to indicate ranges like unused ($\{0\}$), used exactly once ($\{1\}$), used at most once ($\{0, 1\}$), at least once ($\{1, \infty\}$) or no information ($\{0, 1, \infty\}$) also known as Top ($\top$).

### 3.1.7 Exception Analysis

In functional languages it is often allowed to create partial functions. Applying a partial function to a value on which it is not defined results in a runtime exception. Because of this well-typed programs can still go wrong. Many modern day functional compilers are able to warn

for partial functions purely based on syntactic checks that all the possible case statements need to be defined. These syntactic checks result in too many false positives. Programmers who are for example writing a compiler create one AST on which they define functions. This AST contains syntactic constructs that are needed for the parser. In a later stage in the compiler these constructs are desugared into a subset of the original AST, on this subset analysis and optimizations are performed with the invariant that certain constructors are not used anymore. When these stages are accidentally executed before the desugar stage runtime errors will occur that could not be caught by the compiler. With their constraint-based analysis Koot and Hage [21] try to formalize the informal reasoning used by the programmer. The programmer has certain invariants on the data and reasons with a subtype of inferred or given type where certain cases are impossible and possibly meaningless.

## 3.2 Intermediate Languages

### 3.2.1 Calling Conventions

Bolingbroke and Jones [1] designed a new intermediate language for faster function-calls. In this intermediate language calling conventions are directly encoded into the type system, they dubbed it $StrictCore$. In functional languages the use of extensive currying results in multiple functions, each accepting one argument and returning a function that accepts yet another argument, creating a lot of unevaluated thunks on the heap before the function-call is complete. In the new strict core functions can take multiple arguments and return multiple arguments without wrapping them in a tuple on the heap. Fast function-calls are especially important for functional languages because there are a lot of small functions that are passed as an argument and called repeatedly on different arguments. $StrictCore$ is a call-by-value language with explicit laziness annotations. By only using laziness where applicable a lot of thunks are "for free", i.e. by the use of n-ary functions. As they say: "A function may take multiple arguments simultaneously, and (symmetrically) return multiple results." Thunks, called multi-valued thunks, are able to return multiple values at once just as the strict functions. Multi-valued thunks enable deep unboxing as a possible optimization. Deep unboxing is an advanced form of unboxing in that when the components that are contained in a structure are definitely used, when the containing structure is used, they can be unboxed within the containing structure. Deep unboxing is only possible when thunks can return multiple arguments instead of tuples that are allocated on the heap. Two other optimization that are discussed in the paper raise the arity of the functions. Arity raising is important to reduce the number of function calls and thunks that are needed. This is done by combining the calling and returning arguments without the need for function calls or heap allocations.

### 3.2.2 Control Flow

Maurer et al. [26] focus on extending Administrative Normal Form, ANF for short, to include the best of Continuation Passing Style, CPS for short, without the complicated implementation of CPS. They mainly look at *join points*, places in the control flow graph where different executions paths can converge. In the paper the authors use the following example code in Figure 3.3 to clarify what a join point is on which many compilers perform a commuting conversion. Naively applying these conversions lead to the code in Figure 3.4. During the conversion e4 and e5 are duplicated. In practice commuting conversions are very important. Commuting conversions are so important because of the cascade effect that they have on further optimizations. Commuting conversions also have downsides, namely, that the outer code often gets duplicated. This is a

```
if (if e1 then e2 else e3) then e4 else e5
```

Figure 3.3: Nested If

```
if e1 then (if e2 then e4 else e5)
       else (if e3 then e4 else e5)
```

Figure 3.4: Nested If: Naïve Commuting Conversion

```
let { j4 () = e4; j5 () = e5 }
in if e1 then (if e2 then j4 () else j5 ())
          else (if e3 then j4 () else j5 ())
```

Figure 3.5: Nested If: Commuting Conversion with Join Points

problem as it can result in a code size explosion. The resulting machine code is bloated because it is not shared across the different branches anymore. By naming the expressions and referencing the expressions by these names this problem can be countered (see Figure 3.5). Effectively this creates two join points j4 and j5. The executions of the outer branch joins together again at these two points. There is now a new problem namely that the compiler can choose to allocate the closures, for these two functions, on the heap; this results in more allocations. The authors tried to express that the control flow needed to jump to the code that continued the computation, without the need for any allocations.

An extended $\lambda$-calculus style intermediate language is demonstrated in the paper by adding join points and jumps. Because the join points are preserved through subsequent transformations they can be better exploited to be more effective. The authors describes how to infer that ordinary bindings are in fact join points in their $\lambda$-calculus. In CPS the analysis to find join points is called contification. Let-bound functions can be converted to join points and jumps. All calls to these let bindings must be saturated tail calls. Join points can be recursive; these recursive join points result in unexpected opportunities for optimizations while also resolving tension between competing approaches to fusion. This is achieved during the optimization of case-of-case statements where the code can now be floated in without duplicating the code. This results, together with scrutinizing the cases, in no longer needing to generate the intermediate results. This makes it possible to fuse stepper functions when encountering a loop. Filter for example loops over elements and returns the next element that satisfies the predicate every step. These loops break the fusion of the stepper functions, with the join points and jumps these kinds of loops don't break the fusion.

Downen et al. [5] investigated if the sequent calculus is as good an intermediate language as $\lambda$-calculus is. A typed sequent core is developed and implemented in GHC. Optimization passes are reimplemented to work with the new sequent core. Optimizations that are normally implemented in a Continuation Passing Style compiler are easier without sacrificing direct-style optimizations that are ever present in the current core of GHC. In order to compare the results of the different cores the authors rewrote the simplifier "the central piece of GHC's optimization pipeline". The simplifier didn't get the reductions in code size they had hoped. The sequent core did qualitatively better with regard to join points. This fact was exploited by the same authors to extend ANF to get the best of CPS in the paper by Maurer et al. [26]. In sequent core they already implemented the idea that once a join point is found it should stay a joint point.

### 3.2.3 Intermediate Representation

Heeren et al. [17] introduced Helium a compiler designed for learning Haskell. The main purpose of the Helium compiler is to increase the quality of type error diagnosis. Heeren [18] added type-graphs to deliver better type errors. Error messages help programmers to digest a type inconsistency, this is especially true when a student is still learning the language.

In the back-end the currently used runtime is the Lazy Value Machine, or LVM, which interprets the LVM-byte-code. This machine is developed by Leijen [24]. For learning functional programming languages it is important to have a good understanding of what happens during

the runtime of the code. In order to provide this knowledge the LVM machinery uses a tracing mechanism that shows the demand trace of the code. The user has an overview of what was demanded during execution, with this demand trace the user can better understand the runtime error that occurred.

An important part of the runtime is the garbage collector. The garbage collector in the LVM runtime originates from the OCAML project. Because of the functionality that is provided by the garbage collector a lot of the runtime performance for Helium depends on the garbage collectors implementation. On every evaluation loop the garbage collector is asked whether or not it wants to perform an garbage collection cycle. The algorithms that decide whether or not a cycle is necessary take into account how fast the memory usage is growing and how high the memory pressure is.

# Chapter 4

# Helium Compiler

The Helium compiler chapter first explains the pipeline (see Section 4.1) and the available library files (see Section 4.2). Then we explain how data-types can be annotated in the core representation (see Section 4.3). And lastly the constraints that are generated (see Section 4.4) and solved (see Section 4.5). We present the constraint based type system with an example (see Section 4.6).

## 4.1 Pipeline

In the file $helium/src/Helium/Main/Compile.hs$ the pipeline is described. All the different phases in the pipeline are detailed in $helium/src/Helium/Main/Phase\{phase\}.hs$. The pipeline consists of multiple phases:

**Lexer** lexes the file and returns tokens

**Parser** parses the tokens and returns a module

**Import** adds implicit imports for the 'Prelude.hs' and 'HeliumLang.core'. Expands all imports to their exported functions and returns two things; First the declarations for these imported functions which are added to the desugared module and second the import environment which maps these functions to their respective types.

**ResolvingOperators** resolves operators based on their infix specification in lists of expressions.

**StaticChecks** collects three things; first the local environment of the module. Second the user defined type signatures. And the last warning and errors discovered in the module.

There are four kinds of warning:

- Scope warnings including unused and shadowed variables.
- Function bindings with similar names but only one type signature.
- Type variables that resembles type constants.
- Instance declarations, that have no default definition, with missing functions.

There are five categories of errors;

- Export errors which check whether exported functions and modules are in scope.
- Scope errors check for duplicate variables.
- There exist many errors that are described to the *misc* category.
- Kind errors check that type constructors are applied to the right number of type arguments and that type variables do not have any type arguments.

- Top-level errors include duplicated type and value constructors and fixity declarations, fixity declarations without a definition, incorrect overloading, mutual recursive type synonyms and if the filename doesn't coincide with the module-name.

**KindInferencer** this phase is by default turned off.

**TypingStrategies** loads typing strategies from the $\{module\}.type$ file. These strategies are used by the type inferencer to give better type error messages, i.e. sibling functions like $show$ and $read$.

**TypeInferencer** collects type constraints on the AST that represents the entire front-end language. These constraints are solved by a greedy solver and if a type error occurs a type graph is used to give the user the best possible type error.

**Desugarer** desugars the surface language-AST to the core-AST. It also does a quick very conservative pass of dead code removal for unused private declarations.

**CodeGenerator** desugars the core-AST to core-assembly. Before this conversion happens a few normalization steps happen, namely coreRename, coreSaturate, coreNormalize, coreLetSort and coreLift.

A syntactic occurrence analyser is performed. With the occurrence analysis information it starts to inline certain expressions. The algorithm removes let bindings that are never used. The inliner inlines let bindings that are used only once. Trivial cases are possibly duplicated and get inlined everywhere. Flattening is performed on the applications. A special case where the $Eval$, the assembly representation of a strict let, is directly applied to its arguments.

The core-assembly further compiled to LVM-instructions. For certain schemes there are optimized instructions generated. When a variable is already in weak-head-normal-form it is not evaluated again but instead replaced by an $ATOM$. An $EVAL$ that is only used once by a $MATCH$ is passed on the stack instead of on the heap. The special case where the variable is used by a match and already in weak-head-normal-form is special cased to an $ATOM$ on the stack and directly applying the $MATCH$.

A rewrite phase takes care of optimizing the stream of instructions. Functions like id $x = x$ with up to three arguments are completely removed. The return statement is often used to return values on the stack instead of on the heap. More efficient instructions are used in order to push multiple elements around at once.

Finally it writes lvm-byte-code to the target file.

## 4.2 Library Files

The Helium compiler starts at $helium/src/commands/helium/Helium/Main.hs$. As a first step the core libraries are compiled with *coreasm*. The core to lvm assembly compiler can be found in $lvm/src/lib/Lvm/Core/Main.hs$. In the core libraries the low level primitives are defined. The core libraries, available in $helium/lib/\{lib\}.core$, are compiled:

**LvmLang** which describes the primitive instructions available to the runtime. It also explicitly forces the needed values, see Figure 4.1.

**LvmIO** which describes the low level IO operations based around file handles and channels.

**LvmException** which describes all possible exceptions and signals in the LVM runtime. This includes user thrown errors and pattern failures.

**HeliumLang** which describes how to show primitives including a lot of edge cases like adding '.0' after a float and primitives like $\$primPackedToString :: PackedString \rightarrow String$.

```
instruction primAddInt "addint" :: Int! -> Int! -> Int!

(+) :: Int -> Int -> Int!
(+) x y = let! y = y in let! x = x in primAddInt x y
```

Figure 4.1: LvmLang

**PreludePrim** which describes the handling of dictionaries used for type classes.

After the core files are compiled the $helium/lib/Prelude.hs$ is compiled. Now the module that was requested is handled. First load the imports of the module to check if a module is imported that was already compiled; this indicates a circular import and is not allowed. All imported modules are now handled first. After all imports of a certain module are resolved this module can be compiled.

## 4.3 Annotating Data Types

### 4.3.1 Data Type Representation

Before describing how data types are annotated we will first describe how they are represented in Helium core, Figure 4.2.

The data constructors have a function type and a data link. The data link indicates to which data type the constructor belongs. The function type directly shows the type of the constructor. The constructors are appointed an identifying number and the number of arguments they require.

The data type has a number indicating the kindness of the data type.

### 4.3.2 Annotation Algorithm

During the first step we collect the different constructor declarations and sort them under their respective data declarations. Any type synonyms present are used to unfold types to their most basic type.

The next step is to sort the data types topologically. This topological sort is to reduce the number of recursions during the annotation phase. The first step of the sorting phase is building a dependency graph. Once the dependency graph is built it is then used to sort the data types. Sets of mutually dependent data types are bundled together for internal recursion. Sets of data types that have no other sets of data types on which they are dependent are considered at the beginning. All sets of data types on which a set of data types is dependent are sorted to be solved first. This ensures that all dependencies are already solved.

The final step is the annotation algorithm. The algorithm starts with an empty data environment. On every next step it takes the previously calculated environment and adds the next set of mutually dependent data types to it. In order to add a mutually dependent set of data types to the existing environment local recursion is necessary to achieve the correct annotations.

In order to annotate the data types the corresponding constructors need to be typed. There are two environments: the main environment can only be added to once the fix-point iteration is finished, the current environment is the part of the environment that is updated with each run of the fix point algorithm. Annotations are used from the main and the current environment to indicate the number of fresh annotations necessary. By annotating the individual constructors, while ignoring previous annotations on this specific constructor, this process can be repeated until a fix point is reached.

After this process all the data types are annotated with annotation variables. These can be used in the constraint generation phase to state the constraints on the data types.

```
con Nothing : public [custom ''type''  ["Maybe a"]
                     ,custom ''data''  Maybe]
  = (@0,0);

con Just : public [custom ''type''  ["a -> Maybe a"]
                  ,custom ''data''  Maybe]
  = (@1,1);

custom ''data''  Maybe : public [1];

con Left : public [custom ''type''  ["a -> Either a b"]
                  ,custom ''data''  Either]
  = (@0,1);

con Right : public [custom ''type''  ["a -> Either b a"]
                   ,custom ''data''  Either]
  = (@1,1);

custom ''data''  Either : public [2];
```

Figure 4.2: Helium-Core: Data Type Representation

## 4.4 Generating Constraints

There are two type synonyms added that are not included in the modules, namely $String = [Char]$ and $PackedString = Bytes$. Functions that unpack $PackedString$ to bytes are used abundantly for runtime errors and string constants which are efficiently packed as bytes.

Before constraints can be generated, the types that are available in the core module are converted into types that support type annotations. During this conversion all types are completely expanded. We use type synonyms to construct the most basic types.

Dictionaries are expanded to the types they represent. The data-kind-dictionary is retrieved from the data-kinds collected from the module. In the case that it is not in the data-kind-dictionary it is either a $List$ or a $Tuple$, for which the kinds can be calculated from the dictionary name. Respectively the list has a kind of $1$ and the tuple has its kind available as a number in the name.

A global environment is built for the converted types. The global environment is passed down to be used in the functions declared in the module. A unique id is threaded through the constraint generator in order to provide the constraints with fresh variables.

## 4.5 Constraint Solver

Solving constraints works recursively. In order to reduce the number of recursions necessary we order the constraints so that the constraints that can be solved without further information are the first to be inspected.

A first round of the solver is run. Thereafter the type-schemes in the substitutions are simplified. The remaining and newly generated constraints are ordered. The newly ordered constraint set has another round in the solver. If the substitutions are empty and there are no remaining constraints or these are the same constraints as the previous run, the algorithm exits. Otherwise it runs again with the remaining constraints.

The rounds inside the solver solve one constraint and propagate the substitutions to the rest of the constraints. It returns additional constraints generated during the solving phase of the single constraint and the substitution that was produced is applied to the substitution returned by the rest of the constraints.

A single constraint is solved by trying to solve the equality. For types and type-schemes this means trying to unify these. Hereby future constraints need to be generated for the annotations. In order to solve generalizations the constraints first need to be solved. The substitution that is generated is applied to the type and the environment. The free variables are calculated and finally the type-scheme equality constraint is generated. The instantiation requires that the type-scheme is available before it can be processed. We instantiate this type-scheme into a fresh type and generate a type equality constraint.

Other constraints, that solve the equalities for the annotations, will need to be solved.

## 4.6 Type System Example

A solved example is shown for the Combine module (see Figure 4.3). We generate constraints for the core representation (see Figure 4.4). The constraints are solved and the types are shown inline with the core (see Figure 4.5). The framework also shows the types synthesized by the TOP framework.

Type literals don't have any constraints that need to be solved, therefore there are no constraints generated and the type is instantly known for $constant$. Type variables are stored in and recovered from an environment. Therefore only the lambda equality has to be solved for $id'$ and $cons$. Almost all constraints that are visible are used to create a type for $swap$.

```
1  module Combine where
2
3  constant :: Int
4  constant = 42
5
6  id' :: a -> a
7  id' x = x
8
9  cons :: a -> b -> a
10 cons x y = x
11
12 swap :: (a,b) -> (b,a)
13 swap (x,y) = (y,x)
```

Figure 4.3: Combine: Source

```
Lam forall{}. C. 0 == forall{}. C. 12
Let"nextClause$.0" forall(6) == forall{}. C. 33
Bind forall(6) == forall{}. C. 5
Bind 5 == 7
Ap Bytes -> 8 == Bytes -> 7
Let"u$0.1" forall(11) == forall{}. C. 14
Bind forall(11) == forall{}. C. 10
Bind 10 == 12
Match 34 == 14 -> 13
Alts Cons 34 == 15
Pat"x" forall{}. C. 20 == forall{}. C. 30
Pat"y" forall{}. C. 21 == forall{}. C. 29
Alt 15 == 16 -> 22
Pat ConTag 16 == ((,) 17) 18
Pat ConTag 20 == 17
Pat ConTag 21 == 18
Ap 23 == 30 -> 22
Ap 25 -> (26 -> (((,) 25) 26)) == 29 -> 23
Alts Cons 34 == 31
Alt 31 == 32 -> 33
-- Constraints for `cons`
Lam forall{}. C. 35 == forall{}. C. 45
-- Constraints for `id'`
Lam forall{}. C. 46 == forall{}. C. 51
-- Error message constraints
Ap (IO 54) -> 54 == 55 -> 53
Ap ([Char]) -> (IO ()) == 57 -> 55
Ap Bytes -> ([Char]) == Bytes -> 57
```

Figure 4.4: Combine: Constraints

```
module Combine where

-- Imported functions and data-types:
$primPutStrLn :: forall{}. C. ([Char]) -> (IO ())
$primPatternFailPacked :: forall{0}. C. Bytes -> 0
$primUnsafePerformIO :: forall{0}. C. (IO 0) -> 0
$primPackedToString :: forall{}. C. Bytes -> ([Char])
Data: () TypeKind: 0
Data: [] TypeKind: 1
Data: Bool TypeKind: 0
Data: Maybe TypeKind: 1
Data: Either TypeKind: 2
Data: IOMode TypeKind: 0
Data: Handle TypeKind: 0
Data: Ordering TypeKind: 0

-- Local function definitions:
-- type from the TOP-framework
constant :: forall{}. C. Int
-- type synthesized from constraints
constant :: forall{}. C. Int
constant = 42

-- type from the TOP-framework
id' :: forall{0}. C. 0 -> 0
-- type synthesized from constraints
id' :: forall{51}. C. 51 -> 51
id' = \u$0 -> (:: 51 -> 51) u$0 (:: 51)

-- type from the TOP-framework
cons :: forall{0 1}. C. 0 -> (1 -> 0)
-- type synthesized from constraints
cons :: forall{40 45}. C. 45 -> (40 -> 45)
cons = \u$0 u$1 -> (:: 45 -> (40 -> 45)) u$0 (:: 45)

-- type from the TOP-framework
swap :: forall{0 1}. C. (((,) 0) 1) -> (((,) 1) 0)
-- type synthesized from constraints
swap :: forall{29 30}. C. (((,) 30) 29) -> (((,) 29) 30)
swap = \u$0 -> (:: (((,) 30) 29) -> (((,) 29) 30))
  let nextClause$.0 :: forall{}. C. ((,) 29) 30
      nextClause$.0 = $primPatternFailPacked (:: Bytes -> (((,) 29) 30))
        "function bindings ranging from (13,1) to (13,19) in module Combine.hs"
  in let! u$0.1 :: forall{}. C. ((,) 30) 29
          u$0.1 = u$0 (:: ((,) 30) 29)
     in match u$0.1 with (:: (((,) 30) 29) -> (((,) 29) 30))
          [(@0,2) x y -> ((@0,2) (:: 29 -> (30 -> (((,) 29) 30))) y (:: 29)) x (:: 30)
          ,_ -> nextClause$.0 (:: ((,) 29) 30)]

-- Error message when this library is run with `runhelium`
-- type synthesized from constraints
main$ :: forall{}. C. ()
main$ = $primUnsafePerformIO (:: (IO ()) ->
    ()) ($primPutStrLn (:: ([Char]) ->
    (IO ())) ($primPackedToString (:: Bytes ->
    ([Char])) "No 'main' function defined in this module"))
```

Figure 4.5: Combine: Core with Types

# Chapter 5

# Normalization

The normalization chapter first explains why normalization needs to happen (see Section 5.1) and what the advantages are by reducing the number of constraints (see Section 5.2). Then we explain how the normalization is set up in the simplify routine (see Section 5.3). Different sub routines are removing renames (see Section 5.4), normalizing match statements (see Section 5.5) with a more elaborate example (see Section 5.8). The final routine removes dead code (see Section 5.6).

## 5.1 Normalized Core Representation

Normalizing core reduces the complexity of the representation. By reducing the complexity the rest of the pipeline and the runtime will have an easier job to respectively optimize and execute the program.

Before the core normalization the representation incurred a lot of unnecessary allocations. Quite a few of these unnecessary let bindings did a simple rename of the variables to the names selected by the programmer, these where removed, see Section 5.4.

## 5.2 Constraint Reduction

Normalizing and simplifying core reduces the number of constraints that are generated during constraint generation phase of the constraint based type checker for core.

While we were working on the constraint generator we noticed that more constraints where generated than we expected. Therefore the constraint solver took longer to solve these constraints. We viewed the core representation in search of an explanation for these additional constraints. We realized that the core representation of match statements was threaded. Instead of a single match statement, for each level of constructors matched, there was an entire smear of next-clauses that each matched just one constructor. We saw that when a constructor was previously matched it wasn't excluded from the next-clauses. This was done in order to match different nested patterns.

Type-checking would be a lot easier if all case-arms would be represented inside the same match statements. We analysed the core further and decided that it would be possible to combine (see Section 5.5) the case arms that were smeared out over multiple match statements inside next-clauses.

Other let bindings became unnecessary as well after the normalization of the match statements (see Section 5.6). A single pattern match failed expression would be enough to catch the cases when a fall through case is necessary.

## 5.3 Simplify

The simplifications described in Sections 5.4, 5.5 and 5.6 are performed on the expressions of the declarations inside the module. The simplify step is the last of the normalization steps performed on the core. The normalizations that already existed in the LVM module are performed first. These are the existing normalizations:

**Rename** resolves any shadowing with fresh names from the name-supply.

**Saturate** fully applies calls to external and internal functions as well as constructors.

**Normalize** ensures that there are no lambda's except directly at let bindings and applications are only performed on atomics, where atomics are variables, literals, constructors, other normalized applications and normalized let bindings.

**Let-sort** transforms recursive let bindings. A graph of the let bindings is constructed and dependencies are represented by edges. This ensures that the smallest recursive let bindings remain and the other let bindings are non recursive.

**Lift** does Johnsen style lambda lifting. After lifting each binding either has no free variables or no arguments left.

## 5.4 Remove Renames

In the core there are a lot of let bindings dedicated to the renaming of variables. In order to reduce the number of let bindings that are present in the generated core we opted for a pass that removes any renames and inlines the names that were given to the variables at a higher level in the **Expr**.

This pass preserves any strict binding in order not to lose strictness either necessary for the core to function or introduced by the programmer. For example the match statement needs its arguments in WHNF and the strictness introduces a new variable for this purpose (see Figure 5.1).

This pass removes any unnecessary renames. Take the max function (see Figure 5.2) for example. It consists of binding x and y corresponding to the first and respectively second parameter of this function (see Figure 5.3). For the core representation this is unnecessary and can be freely removed by this pass (see Figure 5.4). Thereby two let bindings are removed and the remaining code is a lot more compact.

```
1 let nextclause = exprNextclause
2 in  let! x.1 = x
3     in match x.1 of {
4         Constructor => exprConstructorMatched,
5         _           => nextclause,
6     }
```

Figure 5.1: Match Introduces Strictness

```
max :: Int -> Int -> Int
max x y = if x < y then y else x
```

Figure 5.2: Max Function: Real World Example

```
\''u$0'' ''u$1'' ->
    let ''x.515'' = ''u$0'';
    in  let ''y.516'' = ''u$1'';
        in  let! ''guard$.517'' =
                ''<'' ''x.515'' ''y.516'';
            in match ''guard$.517'' with {
                True -> ''y.516'';
                _ -> ''x.515'';}
```

Figure 5.3: Max Function: Before Rename

```
\''u$0'' ''u$1'' ->
    let! ''guard$.517'' =
        ''<'' ''u$0'' ''u$1'';
    in match ''guard$.517'' with {
        True -> ''u$1'';
        _ -> ''u$0'';}
```

Figure 5.4: Max Function: After Rename

## 5.5 Normalize Matches

### 5.5.1 Pseudo Haskell

The normalization of matches starts off with a depth-first-search of the matches that can be normalized, see lines 14-23 in Figure 5.5. Then for the deepest match, first the alternatives for that match are collected, see Figure 5.6. These alternatives are combined with the already existing alternatives, see Figure 5.7. Deeper patterns in the alternatives need to combine their expressions, see Figure 5.8.

The rest of the code can be found in Figures A.1, A.2, A.2, A.4, A.5 and A.6 in Appendix Normalize: Pseudo Haskell.

### 5.5.2 Normalization Rule

We are going to discuss an example where an hypothetical if statement is represented. The intermediate representation (see Figure 5.9) would be generated by the code generation phase. The if statement is rewritten as two independent match statements which both first force the argument to be in weak head normal form. It is even possible to fall through to a pattern match failure. We can see that this is unreachable but the compiler doesn't, therefore it is not removed.

After rewriting the expression the normalized core (see Figure 5.10) is what remains. The consolidated match statement contains all possible branches on x. Here, $True$ and $False$ are covered and the fall through still points at the pattern match failure. Another pass could be made to check that a match statement covers all possible branches completely and remove the final case, i.e. all constructors have a corresponding arm.

A real world normalization is performed on the function *unwords* (see Figure 5.11) from the prelude. Unwords takes a list of words and concatenates these words together with spaces in between. In the source code that is written by the programmer, we can distinguish 3 patterns; namely the empty list, the singleton list and the list pattern.

The generated core (see Figure 5.12) has a lot of match statements. When we look at the match statements there is something pronounced about them: there are many next-clauses.

**line number 22** The first executed match statement only matches on an empty list, namely []. If this matches this arm returns the empty string.

**line number 14** The second match statement matches the list constructor, namely :. If this matches there is a nested match:

- **line number 17** Matches on the second part of the list. If this is empty it returns the first part of the list.

**line number 10** Matches the list constructor again. If this is a match it recursively calls $unwords$ on the second part of the list, pushes a space unto the returned value and propends the first part of the match. This is subsequently returned.

**line number 5** The fall through case, this is a run-time error, gives information about the location in the source file where the pattern(s) originated that didn't match the value.

There are a few things that are noticeable: there are more matches than necessary, there are matches that match the same constructor(s), and when falling through to the next clause the constructor that was already matched isn't taken into account. This is the result of directly generating core for each individual patterns in the source-code. The patterns are written top-down in the source code are respectively visible on lines 22-24, 14-20 and 10-12.

The normalization of the core (see Figure 5.13) counteracts these direct influences from the number of patterns in the source code.

**line number 8** The first executed match statement matches on one let and decides whether it is an empty list or a list constructor. If this matches the empty list the empty string is returned. If it matches the list constructor there is a nested match:

```haskell
normalizeMatches :: Expr -> Expr
normalizeMatches expr = case expr of
    -- Otherwise is always true
    Let (Strict (Bind nameBind exprBind))
        (Match nameMatch (Alt (PatCon (ConId trueId) []) exprAlt:_))
        | nameBind == nameMatch
        && trueId == "True"
        && exprBind == "otherwise" -> exprAlt
    -- Found a match to be normalized
    Let (NonRec (Bind nameNB exprNB))
        exprN@(Let bindS@(Strict (Bind nameSB exprSB)) (Match nameM alts))
        | leadingNextClause (show nameNB)
        && nameSB == nameM ->
        let -- Depth first normalization
            exprNB' = normalizeMatches exprNB
            exprN' = normalizeMatches exprN
            expr' = Let (NonRec (Bind nameNB exprNB')) exprN'

            exprSB' = normalizeMatches exprSB
            alts' = map (\(Alt pat exprA) ->
                let exprA' = normalizeMatches exprA
                in Alt pat exprA') alts
            -- Collect alternatives for the same variable
            expr'' = case normalize exprSB' exprNB' of
                Just (alts'',bindss) ->
                    let -- Combine alternatives for the same variable
                        combAlts = combineAlts nameNB alts' alts''
                    in foldr Let (Let bindS (Match nameM combAlts)) bindss
                Nothing -> expr'
        in expr''
    -- Propagate search for matches to be normalized
    Let (Strict (Bind nameB exprB)) exprL ->
        let exprB' = normalizeMatches exprB
            exprL' = normalizeMatches exprL
        in Let (Strict (Bind nameB exprB')) exprL'
    Let (NonRec (Bind nameB exprB)) exprL ->
        let exprB' = normalizeMatches exprB
            exprL' = normalizeMatches exprL
        in Let (NonRec (Bind nameB exprB')) exprL'
    Let (Rec binds) exprL ->
        let binds' = map (\(Bind nameB exprB) ->
                let exprB' = normalizeMatches exprB
                in Bind nameB exprB') binds
            exprL' = normalizeMatches exprL
        in Let (Rec binds') exprL'
    Match name alts ->
        let alts' = map (\(Alt pat exprA) ->
                let exprA' = normalizeMatches exprA
                in Alt pat exprA') alts
        in Match name alts'
    Ap expr1 expr2 -> Ap (normalizeMatches expr1) (normalizeMatches expr2)
    Lam name expr1 -> Lam name (normalizeMatches expr1)
    Con _ -> expr
    Var _ -> expr
    Lit _ -> expr
```

Figure 5.5: Pseudo Haskell: Normalize Matches

```haskell
normalize :: Expr -> Expr -> Maybe (Alts,[Binds])
normalize exprName expr = case expr of
    Let (Strict (Bind nameSB exprSB)) (Match nameM alts)
        | nameSB == nameM
        && exprName == exprSB -> Just (alts,[])
    Let binds exprL -> case normalize exprName exprL of
        Just (alts,bindss) -> Just (alts,binds:bindss)
        Nothing -> Nothing
    Lam name exprL -> normalize exprName exprL
    _ -> Nothing
```

Figure 5.6: Pseudo Haskell: Normalize

```haskell
combineAlts :: Id -> Alts -> Alts -> Alts
combineAlts nextClause altsP altsN = case (altsP,altsN) of
    ([],altsN) -> altsN
    (altsP,[]) -> altsP
    ((Alt PatDefault _:altsP),altsN) ->
        combineAlts nextClause altsP altsN
    ((altP@(Alt patP exprP):altsP),altsN) ->
        replaceDefaults nextClause $ case findPat patP altsN of
            (altsN', Just (Nothing, exprN)) ->
                Alt patP (combineExpr nextClause exprP exprN):
                    combineAlts nextClause altsP altsN'
            (altsN', Just (Just idsN, exprN)) ->
                let PatCon contagP idsN = patP
                    idsR = idsRFromList (zip idsP idsN)
                    idsN' = map (\n -> Map.findWithDefault n n idsR) idsN
                in Alt (PatCon contagP idsN')
                    (combineExpr nextClause (updateIds idsR exprP) exprN):
                        combineAlts nextClause altsP altsN'
            (altsN', Nothing) -> altP:combineAlts nextClause altsP altsN'
    where
        findPat :: Pat -> Alts -> (Alts, Maybe (Maybe [Id], Expr))
        findPat (PatLit litP) ((Alt (PatLit litN) exprN):altsN')
            | litP == litN = (altsN', Just (Nothing, exprN))
        findPat (PatCon contagP _) ((Alt (PatCon contagN idsN) exprN):altsN')
            | contagP == contagN = (altsN', Just (Just idsN, exprN))
        findPat patP (altN':altsN') =
            let (altsN'', mAlt) = findPat patP altsN'
            in  (altN':altsN'', mAlt)
        findPat _ [] = ([],Nothing)
```

Figure 5.7: Pseudo Haskell: Combine Alternatives

```
combineExpr :: Id -> Expr -> Expr -> Expr
combineExpr nextClause exprP exprN = case (exprP, exprN) of
    (Let (Strict (Bind namePB _)) _, Let (Strict (Bind nameNB exprNB)) exprNL) ->
        let (Let (Strict (Bind _ exprPB)) exprPL) =
                updateIds (idsRSingleton namePB nameNB) exprP
        in Let (Strict (Bind nameNB (combine exprPB exprNB))) (combine exprPL exprNL)
    (Let (NonRec (Bind namePB _)) _, Let (NonRec (Bind nameNB exprNB)) exprNL) ->
        let (Let (NonRec (Bind _ exprPB)) exprPL) =
                updateIds (idsRSingleton namePB nameNB) exprP
        in Let (NonRec (Bind nameNB (combine exprPB exprNB))) (combine exprPL exprNL)
    (Let (Rec bindsP) exprPL, Let (Rec bindsN) exprNL) ->
        let binds' = map (\(Bind _ exprPB,Bind nameNB exprNB) ->
                Bind nameNB (combine exprPB exprNB)) (zip bindsP bindsN)
        in Let (Rec binds') (combine exprPL exprNL)
    (Match namePM altsP, Match nameNM altsN)
        | namePM == nameNM ->
        let alts' = combineAlts nextClause altsP altsN
        in Match nameNM alts'
    (Ap exprP1 exprP2, Ap exprN1 exprN2) ->
        Ap (combine exprP1 exprN1) (combine exprP2 exprN2)
    (Lam namePL exprPL, Lam nameNL exprNL)
        | namePL == nameNL -> Lam nameNL (combine exprPL exprNL)
    (Con conP, Con conN)
        | conP == conN -> exprN
    (Var namePV, Var nameNV)
        | namePV == nameNV -> exprN
    (Lit litP, Lit litN) | litP == litN -> exprN
    -- If there is a nextClause in exprP place exprN there
    _ | Maybe.isJust (getOcc nextClause (exprOcc exprP)) ->
        replaceNextClause nextClause exprN exprP
    _ -> internalError "PhaseNormalize" "combineExpr" "{error message}"
    where combine = combineExpr nextClause
```

Figure 5.8: Pseudo Haskell: Combine Expression

```
let nextclause1 =
    let nextclause2 = error "pattern match failure"
    in  let! x = x
        in match x of {
            False => exprFalse,
            _     => nextclause2,
        }
in  let! x = x
    in match x of {
        True => exprTrue,
        _    => nextclause1,
    }
```

Figure 5.9: Example If: Before Normalization

```
let nextclause2 = error "pattern match failure"
in  let! x = x
    in match x of {
        True  => exprTrue,
        False => exprFalse,
        _     => nextclause2}
```

Figure 5.10: Example If: After Normalization (Normalized Core)

```
unwords :: [String] -> String
unwords [] = ""
unwords [w] = w
unwords (w:ws) = w ++ ' ' : unwords ws
```

Figure 5.11: Unwords Function: Real World Example

```
\''u$0'' ->
let ''nextClause$.98'' ''u$0'' =
    let ''nextClause$.99'' ''u$0'' =
        let ''nextClause$.103'' =
              ''$primPatternFailPacked''
                "function bindings ranging
                from (482,1) to (484,39)
                in module Prelude.hs";
        in let! ''u$0.104'' = ''u$0'' ;
           in match ''u$0.104'' with {
                ''::'' w ws -> ''++'' w (''::'' (primChr 32) (unwords ws));
                _ -> ''nextClause$.103'' ; };
    in let! ''u$0.100'' = ''u$0'' ;
       in match ''u$0.100'' with {
          ''::'' ''l$0'' ''l$1'' ->
              let! ''l$1.102'' = ''l$1'' ;
              in match ''l$1.102'' with {
                 '':[]'' -> ''l$0'' ;
                 _ -> ''nextClause$.99'' ''u$0'' ; };
          _ -> ''nextClause$.99'' ''u$0'' ; };
in let! ''u$0.105'' = ''u$0'' ;
   in match ''u$0.105'' with {
      '':[]'' -> ''$primPackedToString'' "";
      _ -> ''nextClause$.98'' ''u$0'' ; }
```

Figure 5.12: Unwords Function: Before Normalization

```
\''u$0'' ->
let ''nextClause$.103'' =
    ''$primPatternFailPacked''
       "function bindings ranging
       from (482,1) to (484,39)
       in module Prelude.hs";
in let! ''u$0.105'' = ''u$0'' ;
   in match ''u$0.105'' with {
      '':[]'' -> ''$primPackedToString'' "";
      ''::'' w ws ->
          let! ''l$1.102'' = ws;
          in match ''l$1.102'' with {
             '':[]'' -> w;
             _ -> ''++'' w (''::'' (primChr 32) (unwords ws)); };
      _ -> ''nextClause$.103'' ; }
```

Figure 5.13: Unwords Function: After Normalization (Normalized Core)

**line number 12** Matches on the second part of the list. If this is empty it returns the first part of the list. Otherwise it recursively calls $unwords$ on the second part of the list, it pushes a space unto the returned value and propends the first part of the match. This is subsequently returned.

**line number 3** The fall through case, this is a run-time error, gives information about the location in the source file where the pattern(s) originated that didn't match the value.

## 5.6 Dead Code Elimination

After normalizing the match statements a few next-clauses have become unnecessary. These are eliminated with an occurrence analysis when the entire binding group does not occur in the subsequent expression. Because the let-bindings are already split on mutual recursion by let-sort (see Section 5.3) they are guaranteed to be used recursively.

We use the zip function (see Figure 5.14) from the prelude to demonstrate the dead code removal pass. The generated core (see Figure 5.15) contains two next-clauses for non-existing pattern matches. In the resulting core (see Figure 5.16) the unnecessary let-bindings are removed by the dead code elimination pass.

```
zip :: [a] -> [b] -> [(a,b)]
zip = zipWith  (\a b -> (a,b))
```

Figure 5.14: Zip Function: Real World Example

```
let ''nextClause$.395''  =
        ''$primPatternFailPacked''
            "function bindings ranging
            from (257,1) to (257,31)
            in module Prelude.hs";
in  let ''nextClause$.396''  =
            ''$primPatternFailPacked''
                "function bindings ranging
                from (257,17) to (257,30)
                in module Prelude.hs";
    in let ''.397''  ''u$0''  ''u$1''  =
            (@0,2) ''u$0''  ''u$1'' ;
      in zipWith ''.397''
```

Figure 5.15: Zip Function: Before Dead Code Elimination

```
let ''.397''  ''u$0''  ''u$1''  =
      (@0,2) ''u$0''  ''u$1'' ;
in zipWith ''.397''
```

Figure 5.16: Zip Function: After Dead Code Elimination (Normalized Core)

## 5.7 Optimized Prelude Functions

We analyzed the generated core and deduced that the normalization was effective on the following functions (see Figure 5.17) in the prelude. These functions are widely used and a lot of code actually depends on the Helium compiler to create an efficient implementation for these functions.

- readInt
- unwords
- unlines
- span
- dropWhile
- takeWhile
- splitAt
- drop
- take
- scanr1
- scanr
- foldr1
- foldr
- scanl1
- foldl1
- foldl'
- foldl
- (!!)
- filter
- map
- (++)
- init
- last
- either
- maybe
- signumFloat
- (^)
- signum
- lookup
- elem

Figure 5.17: Optimized Prelude Functions

## 5.8 Larger Normalization

In the Appendix Normalize: Rewrite we provide a complete normalization for the lookup function (see Figure B.1). The remove renames and match normalization phases have had an effect that is visible in the core that remains. The dead code elimination phase didn't have an effect on the core.

# Chapter 6

# Measurements

The this chapter we first explain the ported benchmark suite (see Section 6.1), functions from the prelude (see Section 6.2) and generated match statements (see Section 6.3) used to benchmark the runtime. We then give some information about the different measured virtual memory statistics (see Section 6.4). We discuss how we deal with the warmup phase of the runtime (see Section 6.5). Finally it presents the results that we found while benchmarking (see Section 6.6).

## 6.1 Nofib Benchmark

The *Nofib* benchmark is ported from the GHC Nofib benchmark. For these measurements the *imaginary* directory was used.

We want to know the speed of the standard in and standard out pipelines of the LVM-runtime. Because the runtime doesn't have an exact timing library we needed to use an external framework to benchmark the Helium programs. Therefore we added the following tests as a baseline to measure the speed of parsing an integer or a float and echo the value back to the framework:

**EchoInt** receives a single 0 as input over *stdin*. The function reads this line. Subsequently parses the input string to an integer. And finally returns this integer converted back to a string over *stdout*. The measurements indicate the total delay when working with an integer.

**EchoFloat** works in much the same way as EchoInt. It receives a single 0 as input, reads this line, parses the line to a floating point number and finally converts it back to an string over *stdout*. The measurements indicate the total delay when working with an floating point number.

These test are directly ported to Helium:

**Bernoulli** is executed with inputs ranging from 0 up to and including 15.

**DigitsOfE1** is executed with inputs ranging from 0 up to and including 10. Calculating the digits of Euler's number version 1.

**DigitsOfE2** is executed with inputs ranging from 0 up to and including 10. Calculating the digits of Euler's number version 2.

**Exp3_8** is executed with inputs ranging from 0 up to and including 10. In order to run the benchmarks without the simplify optimization we had to limit the range to 6. At 7 the runtime throws a *segfault* indicating that it has run out of memory. It calculates 3 to the power of its input by using a data-type representation of natural numbers (`data Nat = Z | S Nat`).

**Integrate** is executed with inputs ranging from 0 up to and including 5. These values are relatively small compared to the expected input, we therefore multiply these values by 1000, we get a range from 0 to 5000.

**Nfib** is executed with inputs ranging from 0 up to and including 25.

**Paraffins** is executed with inputs ranging from 0 up to and including 14.

**Primes** is executed with inputs ranging from 0 up to and including 15. These values are relatively small compared to the expected input, we therefore multiply these values by 100, we get a range from 0 to 1500.

**Queens** is executed with inputs ranging from 0 up to and including 9. This solves the generalized eight queens puzzle. The input is used for the number of queens and the size of the chessboard. The result is the number of solutions for the puzzle. It is possible to run Queens with larger inputs but this will take approximately 9 hours or more, for this single data-point.

**WheelSieve1** is executed with inputs ranging from 0 up to and including 5. These values are relatively small compared to the expected input, we therefore multiply these values by 1000, we get a range from 0 to 5000.

**WheelSieve2** is executed with inputs ranging from 0 up to and including 5. These values are relatively small compared to the expected input, we therefore multiply these values by 1000, we get a range from 0 to 5000.

**X2n1** is executed with inputs ranging from 0 up to and including 5. These values are relatively small compared to the expected input, we therefore multiply these values by 1000, we get a range from 0 to 5000.

These tests were ported not included during the benchmarking because they required multiple inputs and don't scale based on the inputs but vary randomly:

**Tak** uses 3 numbers *27 16 8* per line as input. Then it recursively calls itself 4 times, 3 times mixing the numbers and once with the results of the previous calls.

**GenRegexps** generate all the expansions of a generalised regular expression. An input of *[a-j][a-j][a-j]abcdefghijklmnopqrstuvwxyz* for example results in *29000* which indicate the number of characters in the output string.

## 6.2 Prelude Function

From the functions that were normalized (see Figure 5.17) the program (see Figure 6.1) was composed. The functions from the prelude that were used are: (!!), lines and unlines, unwords and words and map. The program first reads the text containing paragraphs of "lorum ipsum" through the $collect$ function. The function $execute$ reads *stdin* and returns the character at the index that is passed in. The three different arguments that are used are: 0 the start of the text, 13847 the middle of the text, and 27693 end of the text. With these different inputs a different amount of work is necessary because the text is first passed through the $lorum\_expensive$ function before the character at the index is returned.

```haskell
main :: IO ()
main = do
    line <- getLine -- Number of lines text
    let n = readInt line
    lorum <- collect n -- Lorum Ipsum text
    print $ length lorum -- Number of chars in text
    execute lorum

collect :: Int -> IO String
collect 0 = return ""
collect n = do
    l <- getLine
    ls <- collect (n-1)
    return $ l ++ '\n' : ls

-- This is measured
execute :: String -> IO ()
execute lorum = do
    line <- getLine
    if line == ""
     then return ()
     else do
        let n = readInt line -- Get the index which the character to return
        print $ (lorum_expensive lorum) !! n
        execute lorum

lorum_expensive :: String -> String
lorum_expensive lorum = unlines $ map unwords $ map words $ lines $ lorum
```

Figure 6.1: Prelude Functions Benchmark

## 6.3 Match Statements

The normalization phase specifically focussed on normalizing match statements; therefore it is necessary to include measurements focussing on certain match constructs that are tackled. Different match functions are generated. These match functions are called with a single input parameter (see Figure 6.2) and return a boolean. Three of the four match functions generate a list from this input parameter.

**Linear Width** (see Figure 6.3) already has an optimized representation of the match statements. Normalizing generated minimal changes to the core representation The code generated already follows certain aspects that are introduced by normalizing the next match expression, i.e. these match statements are already inlined. There are no overlapping case arms that can be combined.

**Linear Height** (see Figure 6.4) has more substantial changes, there exist a next-clause binding for every pattern in the source. The non-normalized code is thus folded in such a way by the match statement normalization phase that the case arms are combined in a single match statement.

**Quadratic Up** (see Figure 6.5) also has the generated next clause for every pattern in the source code. The size of the individual patterns goes up as we go to the next pattern. Every pattern is generated as a series of stand-alone matches. For every following pattern the previous pattern has discarded the match only on the empty list element at the end of the pattern. In order to match the last pattern in this statement all previous patterns have been complete matches except for the last element. The normalization folds these patterns together and this results in a linear match time with the length of the pattern that was to be matched.

**Quadratic Down** (see Figure 6.6) also has the generated next-clause for every pattern in the source code. The size of the individual patterns goes down as we go to the next pattern. Every pattern is still generated as a series of stand-alone matches. For every following pattern the previous pattern has discarded the match only after the entire list is matched but the empty list element at the end of the list. In order to match the pattern in the centre this match statement has executed a quadratic amount of matches. All previous patterns have matched the list except for the last element, the nil element, of the list. The normalization folds these patterns together and this results in a linear match time in the length of the pattern that was to be matched.

```
1 main :: IO ()
2 main = do
3     line <- getLine
4     if line == ""
5      then return ()
6      else do
7         let n = readInt line
8         print $ match_test n
9         main
```

Figure 6.2: Calling Match Function

```
10  match_test :: Int -> Bool
11  match_test n = case [0..n] of
12      [0, ··· ,size] -> (is_even size)
13      _ -> False
```

Figure 6.3: Match Function: Linear Width

```
10  match_test :: Int -> Bool
11  match_test n = case n of
12      0 -> True
13      ⋮
14      size -> (is_even size)
15      _ -> False
```

Figure 6.4: Match Function: Linear Height

```
10  match_test :: Int -> Bool
11  match_test n = case [0..n] of
12      [0] -> True
13      ⋮       ⋱
14      [0, ··· ,size] -> (is_even size)
15      _ -> False
```

Figure 6.5: Match Function: Quadratic Up

```
10  match_test :: Int -> Bool
11  match_test n = case [0..n] of
12      [0, ··· ,size] -> (is_even size)
13      ⋮       ⋰
14      [0] -> True
15      _ -> False
```

Figure 6.6: Match Function: Quadratic Down

## 6.4 Virtual Memory

Just before benchmarking the generated code the Helium runtime is spun up, the Helium runtime process is started with the benchmark program as its argument. A new process is spun up for each individual input that is benchmarked. When the process is settled the memory consumption of the runtime is recorded. After benchmark is completed the memory measurements are taken again. Because we want the process to start in the same state for all the inputs we teared the process down after a single input is tested.

The measurements are taken from the OS **proc**{**process_id**}**status**, all the measurements which relate to **virtual** memory are recorded **Vm**$*$, as described in the manpage[1] and in this blog post[2].

**VmPeak** this marks the peak virtual memory usage.

**VmSize** this marks the current virtual memory usage.

**VmLck** the locked memory size, pages that can't be swapped out of memory but are allowed to move.

**VmPin** the pinned memory size, pages that can't be moved because of direct access to physical memory location.

**VmHWM** this marks the maximum resident memory usage. HWM is the acronym for High Water Mark.

**VmRSS** this marks the current resident memory usage. RSS is the acronym for Resident Set Size.

**VmData** this marks the size of the data segment.

**VmStk** this marks the size of the stack segment.

**VmExe** this marks the size of the text segment.

**VmLib** this marks the size of shared libraries.

**VmPTE** this marks the size of page table entries. PTE is the acronym for Page Table Entries.

**VmSwap** this marks the swapped-out memory.

## 6.5 Benchmarking

### 6.5.1 Hardware

The system on which the test are run is a Dell XPS 9550 (see Table 6.1). Before running a benchmark we rebooted the system and let it settle with no additional processes started and approximately 15GiB[3] of free memory available.

### 6.5.2 Imaginary

Criterion[4] is used to run the benchmark. There is a warmup phase of at least 3 seconds. During the warmup phase it is estimated how many iterations should be performed. During the benchmark there is a minimum runtime for each individual test of at least 5 seconds. There also exists a minimum number of iterations, namely 5050 iterations need to be executed. Criterion handles the measurements and the averaging of the execution times taken with each iteration.

[1] `http://man7.org/linux/man-pages/man5/proc.5.html`
[2] `https://ewx.livejournal.com/579283.html`
[3] GiB equals $1024^3$ bytes, GB equals $1000^3$ bytes
[4] `https://github.com/bheisler/criterion.rs`

| Spec | Value |
|---|---|
| System | Dell XPS 9550 |
| CPU | Core i7-6700HQ (2.6GHz x 8) |
| RAM | 16GB DDR4 2133MHz |
| OS | Ubuntu 18.04.2 LTS |

Table 6.1: Hardware

### 6.5.3 Prelude

For measurements of the prelude functions we executed the test program with and without the normalization 120 times. Each time we did 16 iterations for warmup and 16 iterations that measured the time it took. The total times per execution are divided by 16 and there are 120 measurements for both the normalization enabled and disabled.

### 6.5.4 Match Statements

For measuring the match statements we generated the necessary test programs. We generated the quadratically increasing test programs for each match test. We did this until we hit the limit of either the runtime or the compiler. For the linear width test we generated until 1024 items were matched in succession. For the linear height test we generated 4096 different literal patterns. For the quadratic test cases we reached 256 items ordered by increasing width, up, and by decreasing width, down.

For measuring the match statements we executed the generated test program with and without the normalization 120 times. Each time we did a warmup, walking all possible match arms, we measured how long it took to take every match arm specifically.

## 6.6 Results

### 6.6.1 Execution Times: Imaginary

The imaginary part of the nofib benchmark was ported in order to review performance differences before and after the normalization changes. A short overview of the timing data is in Table 6.2. All the functions are displayed with their highest input values. The higher the input value the longer the average runtime is. For the complete set of timing data we refer to the appendix (see Table C.1).

The first thing that jumps out is that Exp3_8 ran into a stack overflow, 4 inputs lower than was tested with the normalizations, this resulted in a segmentation fault (see Section 6.6.4). Running into a segmentation fault can indicate that the memory usage was out of control, therefore we measured the memory consumption (see Section 6.6.5).

In the overview about the collected timing data (see Table 6.2) we displayed some basic stats. The total number of invocations is 142, of those 138 didn't segfault. A generous set of 124 programs had their speed increased, with a maximum improvement of 21% and an average improvement of 3.66%. Only a subset of 4 invocations were slower than the baseline, with a worst degradation of 8% and an average degradation of 4.50%. A total of 10 functions performed the same as the baseline, the average improvement measured 3.16% better than the baseline. This of course excludes the 4 segfaults that have an infinitely better performance with the normalization.

### 6.6.2 Execution Times: Prelude Functions

We want to be able to state that our implementation did improve the performance. We have the following hypothesis:

$H_{null}$ There is no performance difference.

| invocation | | disabled simplify | | enabled simplify | |
|---|---|---|---|---|---|
| function | input | relative | actual | relative | actual |
| EchoInt | 0 | 1.03 | $13.1 \pm 6.85\mu s$ | 1.00 | $12.7 \pm 6.46\mu s$ |
| EchoFloat | 0 | 1.02 | $36.6 \pm 16.88\mu s$ | 1.00 | $35.8 \pm 16.03\mu s$ |
| Bernoulli | 15 | 1.07 | $41.4 \pm 22.04\mu s$ | 1.00 | $38.6 \pm 17.76\mu s$ |
| DigitsOfE1 | 10 | 1.04 | $71.5 \pm 33.75\mu s$ | 1.00 | $68.9 \pm 30.20\mu s$ |
| DigitsOfE2 | 10 | 1.03 | $44.8 \pm 20.36\mu s$ | 1.00 | $43.5 \pm 19.05\mu s$ |
| Exp3_8 | 10 | stack overflow | | 1.00 | $57.5 \pm 4.74ms$ |
| Integrate | 5 | 1.05 | $21.4 \pm 10.38\mu s$ | 1.00 | $20.3 \pm 8.58\mu s$ |
| Nfib | 25 | 1.00 | $45.7 \pm 1.08ms$ | 1.00 | $45.7 \pm 1.29ms$ |
| Paraffins | 14 | 1.01 | $149.5 \pm 1.42ms$ | 1.00 | $147.5 \pm 1.63ms$ |
| Primes | 15 | 1.01 | $17.3 \pm 38.26ms$ | 1.00 | $17.2 \pm 37.99ms$ |
| Queens | 9 | 1.02 | $124.1 \pm 1.18ms$ | 1.00 | $121.7 \pm 0.99ms$ |
| WheelSieve1 | 5 | 1.00 | $16.9 \pm 7.92\mu s$ | 1.00 | $16.9 \pm 8.73\mu s$ |
| WheelSieve2 | 5 | 1.00 | $16.8 \pm 8.33\mu s$ | 1.00 | $16.8 \pm 8.42\mu s$ |
| X2n1 | 5 | 1.00 | $36.7 \pm 0.74ms$ | 1.00 | $36.6 \pm 1.06ms$ |

. . . 128 more invocations

| | Invocations | Highest | Mean |
|---|---|---|---|
| Improvement | 124 | 21% | 3.66% |
| Degradation | 4 | 8% | 4.50% |
| Average Improvement | 138 | | 3.16% |
| Stack overflow prevented | 4 | | |

Table 6.2: Overview Of Timing Data

| invocation | | disabled simplify | | enabled simplify | |
|---|---|---|---|---|---|
| function | input | relative | actual | relative | actual |
| Prelude | 0 | 1.00 | $13.6 \pm 0.27\mu s$ | 1.00 | $13.6 \pm 0.35\mu s$ |
| Prelude | 13847 | 1.02 | $38.7 \pm 0.41ms$ | 1.00 | $37.9 \pm 2.10ms$ |
| Prelude | 27693 | 1.04 | $79.1 \pm 3.95ms$ | 1.00 | $76.0 \pm 0.75ms$ |

Table 6.3: Prelude Function Stats

$H_1$ There is a performance gain.

Here the performance is specified as the time it takes to execute the program.

We need to perform T-Tests to give accurate answers to the hypothesis. We taken an alpha of 0.05 to be able to state the following facts with 95% certainty. The p-value is a value between 0 and 1. If the p-value is lower than our alpha we have a significant difference. In order to select the T-Test we are allowed to use on our data we have to know whether or not the variances are significantly different between the two runtimes.

#### F-Test

F-Test indicate whether the variances are significantly different from each other, therefore we perform F-Tests on our data. We assume that there exist no difference between the run with the normalization enabled and the normalization disabled. We have the following hypothesis:

$H_{null}$ The variance remains the same with the simplification enabled and disabled.

Using the F-Test for input 0, Table D.1, we have determined that simplify enabled (mu = 13608, variance = 120168) is significantly different from simplify disabled (mu = 13554, variance = 71388), p = 2.4111E-3. Therefore we reject the null hypothesis for input 0.

Using the F-Test for input 13847, Table D.2, we have determined that simplify enabled (mu = 37874645, variance = 4.4093E+12) is significantly different from simplify disabled (mu = 38696236, variance = 1.6657E+11), p = 7.0025E-53. Therefore we reject the null hypothesis for input 13847.

Using the F-Test for input 27693, Table D.3, we have determined that simplify enabled (mu = 75952407, variance = 5.6222E+11) is significantly different from simplify disabled (mu = 79139739, variance = 1.5587E+13), p = 5.4286E-54. Therefore we reject the null hypothesis for input 27693.

The variances of the two implementations differ across all the tested inputs. Therefore we need to execute the two sample T-Test assuming unequal variances. We also note that the variance of the run with the simplification disabled is only higher than the simplification enabled at Table D.3.

#### T-Test Assuming Unequal Variance.

The T-Test indicates whether the means of the two executions are significantly different from each other. We can therefore make statements about whether or not an implementation is faster or slower.

First we need to reject the $H_{null}$ hypothesis for the inputs to be able to state that there is indeed a difference. Then we can test our alternative hypothesis $H_1$.

Using the T-Test for input 0, Table 6.4, we have determined that simplify enabled (mu = 13608, variance = 120168) is not significantly different from simplify disabled (mu = 13554, variance = 71388), p = 1.7908E-1. Therefore we cannot reject the null hypothesis for input 0.

Using the T-Test for input 13847, Table 6.5, we have determined that simplify enabled (mu = 37874645, variance = 4.4093E+12) is significantly different from simplify disabled (mu = 38696236, variance = 1.6657E+11), p = 4.8238E-5. Therefore we reject the null hypothesis for input 13847.

Using the T-Test for input 27693, Table 6.6, we have determined that simplify enabled (mu = 75952407, variance = 5.6222E+11) is significantly different from simplify disabled (mu = 79139739, variance = 1.5587E+13), p = 1.4859E-14. Therefore we reject the null hypothesis for input 27693.

Both cases that use a higher input number 13847 and 27693 had means that significantly differed from the baseline test.

| T-Test Two-Sample Assuming Unequal Variance | enabled simplify | disabled simplify |
|---|---|---|
| Mean | 13608 | 13554 |
| Variance | 120168 | 71388 |
| Observations | 120 | 120 |
| df | 224 | |
| $T_{stat}$ | 1.3478 | |
| $P(F \leq f)$ one-tail | 8.9541E-2 | |
| $T_{crit}$ one-tail | 1.6517 | |
| $P(F \leq f)$ two-tail | 1.7908E-1 | |
| $T_{crit}$ two-tail | 1.9706 | |

Table 6.4: T-Test Input: 0

| T-Test Two-Sample Assuming Unequal Variance | enabled simplify | disabled simplify |
|---|---|---|
| Mean | 37874645 | 38696236 |
| Variance | 4.4093E+12 | 1.6657E+11 |
| Observations | 120 | 120 |
| df | 128 | |
| $T_{stat}$ | -4.2073 | |
| $P(F \leq f)$ one-tail | 2.4119E-5 | |
| $T_{crit}$ one-tail | 1.6568 | |
| $P(F \leq f)$ two-tail | 4.8238E-5 | |
| $T_{crit}$ two-tail | 1.9787 | |

Table 6.5: T-Test Input: 13847

Secondly finding out whether the performance has indeed improved. We can test our hypothesis $H_1$ for the program with inputs 13847 and 27693. If the test show that $T_{stat} < -T_{crit}$ one-tail, we accept hypothesis $H_1$. If the hypothesis is accepted we can state that we have gained performance!

Using the T-Test for input 13847, Table 6.5, we have determined that simplify enabled (mu = 37874645, variance = 4.4093E+12) is significantly faster then simplify disabled (mu = 38696236, variance = 1.6657E+11), $T_{stat} = -4.2073 < -T_{crit} = -1.6568$. Therefore we accept hypothesis $H_1$ for input 13847.

Using the T-Test for input 27693, Table 6.6, we have determined that simplify enabled (mu = 75952407, variance = 5.6222E+11) is significantly different from simplify disabled (mu = 79139739, variance = 1.5587E+13), $T_{stat} = -8.6885 < -T_{crit} = 1.6568$. Therefore we accept hypothesis $H_1$ for input 27693.

The observed difference between the sample means is in both cases significant. Therefore we can state that the performance has increased.

What does this tell us about the case where the input is 0? In that case there are a fewer fall through match statements than there were in the hot path without the simplification. The increased speed comes from the deeply nested match branches that get inlined into higher match clauses. If these deeply nested matches are not used the speed increase will be negligible.

### 6.6.3 Execution Times: Match Statements

The Figures 6.7, 6.8, 6.9, 6.10, 6.11 and 6.12 use a logarithmic scale to display the differences between the runtimes in nanoseconds. The graphs start around $1 \cdot 10^4 ns$ or $10 \mu s$ and ends around $1 \cdot 10^6 ns$ or $1000 \mu s$ or $1 ms$. The color soft red is used when the normalization is disabled, while cyan is used for the data-points where the normalization is enabled. The vertical lines indicate the maximum and minimum runtime of the program, where the dot indicates the median of these values. We fitted a smooth blue line through the data-points, the line is dotted when used for

| T-Test Two-Sample Assuming Unequal Variance | enabled simplify | disabled simplify |
|---|---|---|
| Mean | 75952407 | 79139739 |
| Variance | 5.6222E+11 | 1.5587E+13 |
| Observations | 120 | 120 |
| df | 128 | |
| $T_{stat}$ | -8.6885 | |
| $P(F \leq f)$ one-tail | 7.4294E-15 | |
| $T_{crit}$ one-tail | 1.6568 | |
| $P(F \leq f)$ two-tail | 1.4859E-14 | |
| $T_{crit}$ two-tail | 1.9787 | |

Table 6.6: T-Test Input: 27693

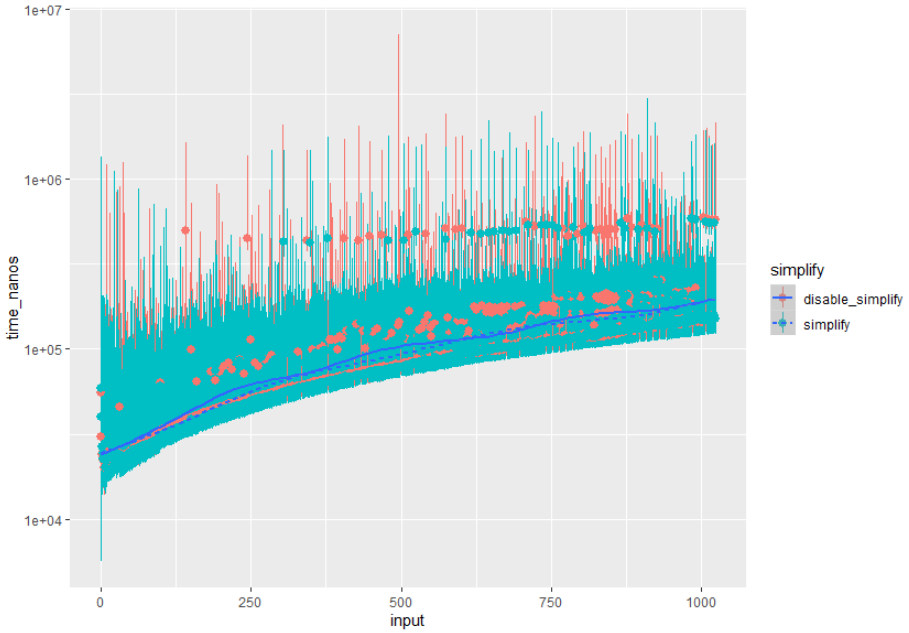


Figure 6.7: Linear Width: Code in Figure 6.3

the normalized data-points. For the quadratic functions we split the data-points on the different program sizes. The dots are replaced by symbols for the respective sizes. The lines are fitted on each subset of data-points.

From Figure 6.7 it is clear that there where no changes in the execution times. The normalization pass did not have anything that could be put in the same match under different case statements. Therefore we did not expect to see any difference between the two runtimes.

From Figure 6.8 it is clear that the runtimes diverge. The tight loop created by combining the case arms into one match statement clearly resulted in an increase in speed.

In Figure 6.9 we split the different tested match sizes out. The left corner is not entirely visible therefore we zoomed in to produce Figure 6.10. It is clear that after an input of around 10 the difference between the two implementations becomes increasingly visible. Where the extra time for lower inputs is coming from is unclear but is visible in both implementations and thus not caused by the normalization.

In Figure 6.11 we split the different tested match sizes out. This time the different sizes clearly show different effects. The left corner is again a bit cramped therefore the zoomed in to produce Figure 6.12. The input 0 has to traverse all the next clauses in the original implementation. It is visible that this has an effect on even the smallest of inputs.


Figure 6.8: Linear Height: Code in Figure 6.4

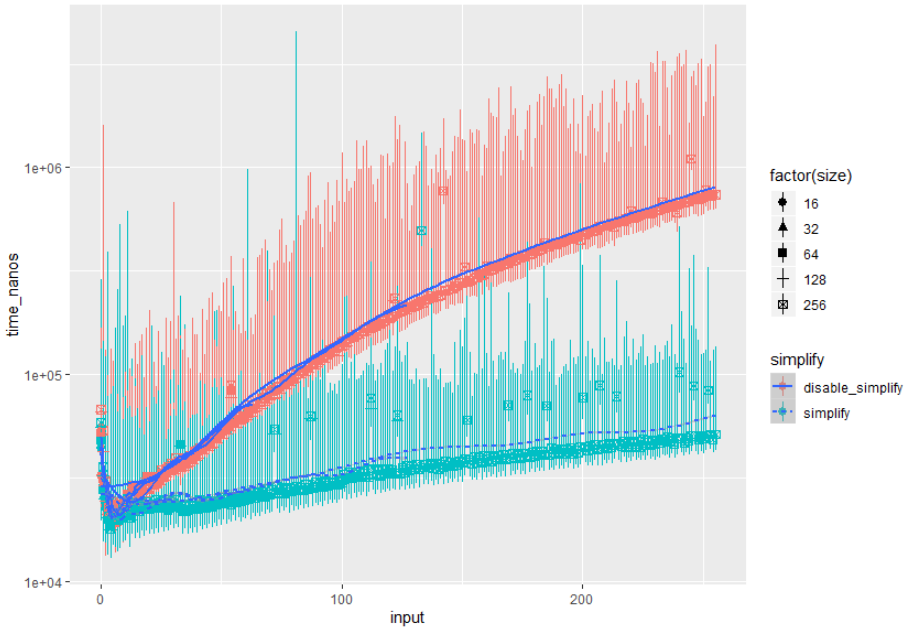


Figure 6.9: Quadratic Up: Code in Figure 6.5

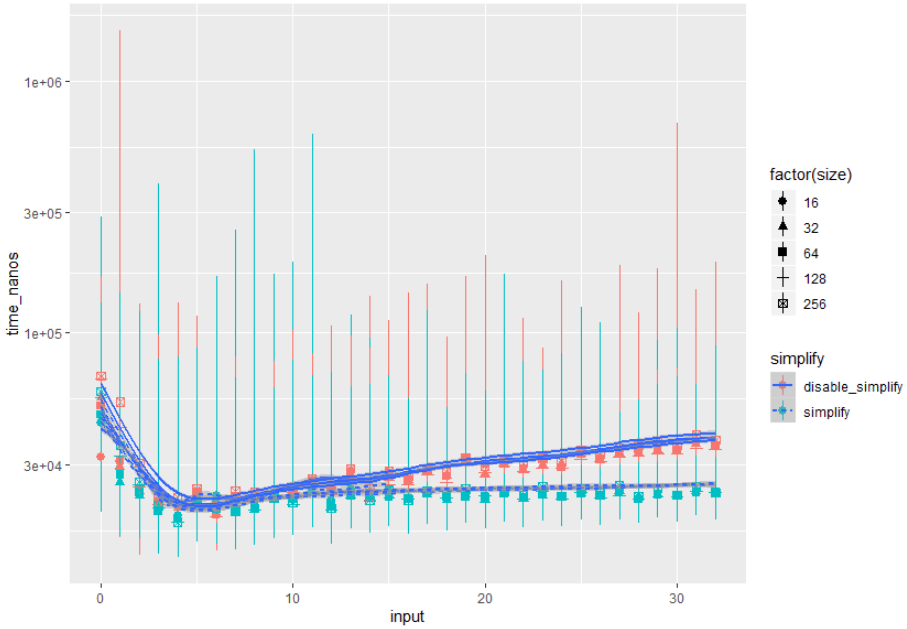


Figure 6.10: Quadratic Up: Zoom 0-32

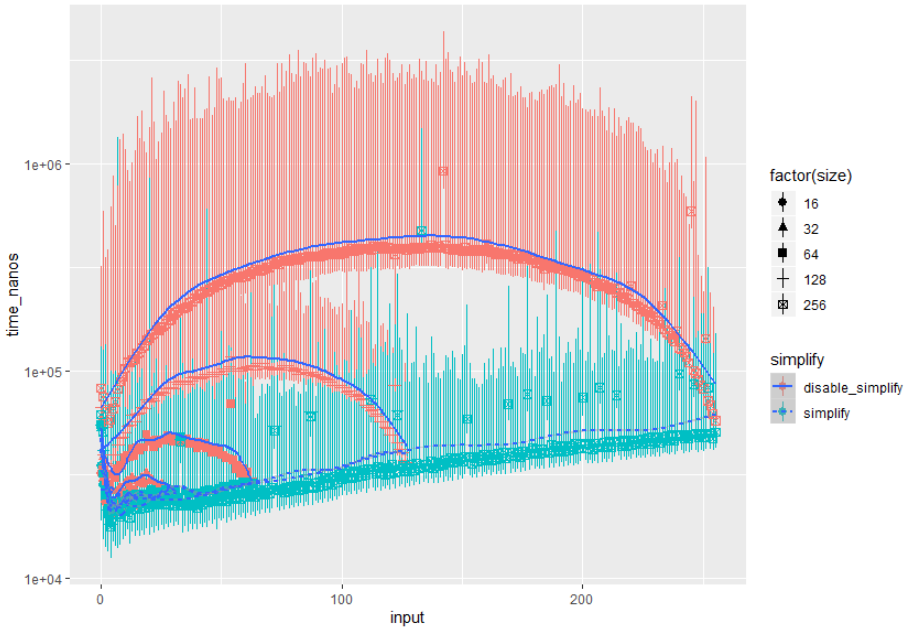


Figure 6.11: Quadratic Down: Code in Figure 6.6

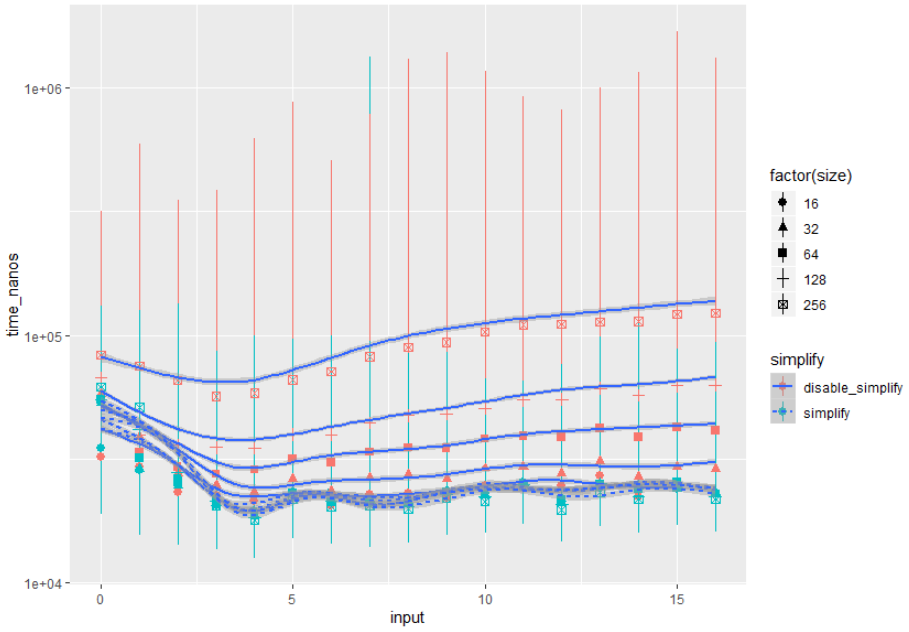


Figure 6.12: Quadratic Down: Zoom 0-16

The main difference with enabling normalization is that all matches that are executed now result in progress towards the actual pattern that is to be matched. For nested patterns, that have overlap with other patterns, progress can now be shared. Shorter sub-patterns can now be implemented as an early-out if the longer pattern is not matched.

### 6.6.4 Segmentation Faults

While building the benchmarking harness we encountered multiple segmentation faults. These segmentation faults were happening when the test input was reached the higher numbers from the range for certain programs under benchmarking pressure. The generated byte code of these programs strained the runtime more than it was able to cope with. This result led us to believe that the runtime would use the most memory for these kinds of programs. Therefore the benchmarking harness was extended to include memory usage measurements. In order to run the benchmarks without the simplification the test input had to be reduced.

### 6.6.5 Memory Consumption

We analysed the data for memory and it indicated something peculiar; the memory consumption did not depend on either the program we were running nor on the input we were using. The program *Exp3_8*, which segfaults from certain inputs onwards without our simplification step, does not increase its memory usage before the segfault is encountered. This indicates that the runtime reserves a predetermined amount of memory and doesn't request more memory from the OS while running.

In the data we have found one outlier in the Virtual memory High Water Mark, *VmHWM*, this is decreased by 75400kB during benching. As this is a high water mark it should always be increasing. Therefore this is an unexpected result and can only originate from a previous process with the same process id. We can thus assume that we read the memory usage of a previous process with the same process id that used 75400kB more memory than we ended up using. This was the reason to search for other outliers that had a higher value for *VmHWM* when they started than when they stopped. We found out that all Helium runtime processes start with 8kB of memory. There exist two data points where the value is different from 8kB before running,

starting with a high water mark of 79576kB and 428kB. We have excluded these data-points from our memory-data.

We have recorded all the statistics that refer to virtual memory usage. From the different statistics a lot are just 0, e.g. we do not lock our pages to physical memory which is indicated by *VmLck* and we do not pin our memory to a single physical memory location *VmPin*. Other statistics have a single value independent from the program that is loaded. These depend on the *runhelium* executable, e.g. *VmData*, *VmStk*, *VmExe* and *VmLib*. We do not end up in swap memory therefore *VmSwap* stays 0. The *VmPeak* and *VmSize* are always the same value, 1073768576kB, this is the virtual memory that is requested by the *runhelium* executable.

There are a few statistics available that show some variance: those are *VmHWM*, *VmRSS* and *VmPTE*. At the start of the runtime *VmHWM* and *VmRSS* are 8kB. We calculated the mean and the variance after running. The mean and the variance are very close to each other indicating that we probably are not measuring what we were hoping for.

We executed a t-test on the invocation with the largest difference in runtime (*DigitsOfE2* with input $5$). Our null hypothesis was that there was a observable difference in the maximum memory usage, *VmHWM*, between enabling and disabling the normalization. The result was that even the execution with the largest timing increase didn't have any significant difference in memory consumption. The calculated $T_{stat}$ is $0.8817$ and the two-tailed p-value is $0.3789$, with a threshold of $5\%$, a p-value was larger than $0.05$, we must reject the hypothesis. There was no significant difference between the two samples.

The memory usage of the program did not depend on the input.

### 6.6.6 Storage Use

We have measured the file size of the prelude file, its source and with and without the normalization. We have chosen the prelude as it is an example of a larger library. Libraries are shared either as source code or as lvm-byte-code file, when the programmer doesn't want to share his source code.

We also measured the sizes of the different lvm files for the generated match statements. We colored the first and last place in file size green and red. For an overview of the files see Table 6.7.

When we view these results we can clearly see that we made great strides towards reducing the byte-code files. The lvm-byte-code file will be faster to send over the internet as well as faster to run in the LVM runtime. Especially when patterns in the match statement overlap the generated files are much smaller. The maximum difference is a whopping 140 times reduction in file size!

| File | Source | Enabled | Disabled | Improvement |
|---|---|---|---|---|
| Prelude | 14.8 kB | 62.4 kB | 68.2 kB | 0.1 X |
| Complex | 1.8 kB | 7.4 kB | 7.4 kB | 0 X |
| Debug | 148 bytes | 2.9 kB | 2.9 kB | 0 X |
| Ratio | 791 bytes | 7.7 kB | 7.7 kB | 0 X |
| | | | | |
| EchoInt | 153 bytes | 3.3 kB | 3.3 kB | 0 X |
| EchoFloat | 154 bytes | 3.3 kB | 3.3 kB | 0 X |
| Bernoulli | 1.7 kB | 9.2 kB | 9.2 kB | 0 X |
| DigitsOfE1 | 716 bytes | 7.5 kB | 7.5 kB | 0 X |
| DigitsOfE2 | 842 bytes | 6.8 kB | 6.8 kB | 0 X |
| Exp3_8 | 2.5 kB | 7.1 kB | 7.1 kB | 0 X |
| Integrate | 1.3 kB | 8.6 kB | 8.6 kB | 0 X |
| Nfib | 338 bytes | 4.2 kB | 4.2 kB | 0 X |
| Paraffins | 3.3 kB | 19.0 kB | 19.0 kB | 0 X |
| Primes | 473 bytes | 5.6 kB | 5.6 kB | 0 X |
| Queens | 598 bytes | 6.0 kB | 6.0 kB | 0 X |
| WheelSieve1 | 1.1 kB | 10.1 kB | 10.1 kB | 0 X |
| WheelSieve2 | 1.4 kB | 10.8 kB | 10.8 kB | 0 X |
| X2n1 | 1.2 kB | 6.8 kB | 6.8 kB | 0 X |
| | | | | |
| linear_width_1024 | 4.3 kB | 78.0 kB | 110.8 kB | 0.4 X |
| linear_width_512 | 2.2 kB | 41.1 kB | 57.5 kB | 0.4 X |
| linear_width_256 | 1.2 kB | 22.7 kB | 30.9 kB | 0.4 X |
| linear_width_128 | 658 bytes | 13.5 kB | 17.6 kB | 0.3 X |
| linear_width_64 | 437 bytes | 8.8 kB | 10.9 kB | 0.2 X |
| linear_width_32 | 341 bytes | 6.5 kB | 7.6 kB | 0.2 X |
| linear_width_16 | 293 bytes | 5.4 kB | 5.9 kB | 0.1 X |
| linear_width_8 | 270 bytes | 4.8 kB | 5.1 kB | 0.1 X |
| | | | | |
| linear_height_4096 | 70.8 kB | 69.2 kB | 575.1 kB | 8.3 X |
| linear_height_2048 | 35.0 kB | 36.5 kB | 288.4 kB | 7.9 X |
| linear_height_1024 | 17.1 kB | 20.1 kB | 145.1 kB | 7.2 X |
| linear_height_512 | 8.6 kB | 11.9 kB | 73.4 kB | 6.2 X |
| linear_height_256 | 4.4 kB | 7.8 kB | 38.5 kB | 4.9 X |
| linear_height_128 | 2.3 kB | 5.7 kB | 21.1 kB | 3.7 X |
| linear_height_64 | 1.2 kB | 4.7 kB | 12.4 kB | 2.6 X |
| linear_height_32 | 734 bytes | 4.2 kB | 8.0 kB | 1.9 X |
| linear_height_16 | 486 bytes | 3.9 kB | 5.9 kB | 1.5 X |
| | | | | |
| quadratic_down_256 | 113.3 kB | 26.8 kB | 3.8 MB | 141.8 X |
| quadratic_down_128 | 26.5 kB | 15.5 kB | 960.2 kB | 62.0 X |
| quadratic_down_64 | 7.0 kB | 9.9 kB | 252.8 kB | 25.5 X |
| quadratic_down_32 | 2.1 kB | 7.1 kB | 71.1 kB | 10.0 X |
| quadratic_down_16 | 818 bytes | 5.6 kB | 23.3 kB | 4.2 X |
| quadratic_down_8 | 458 bytes | 4.9 kB | 10.2 kB | 2.1 X |
| quadratic_down_4 | 340 bytes | 4.6 kB | 6.4 kB | 1.4 X |
| quadratic_down_2 | 293 bytes | 4.4 kB | 5.1 kB | 1.2 X |
| quadratic_down_1 | 273 bytes | 4.3 kB | 4.6 kB | 1.1 X |
| | | | | |
| quadratic_up_256 | 113.3 kB | 26.8 kB | 3.7 MB | 138.1 X |
| quadratic_up_128 | 26.5 kB | 15.5 kB | 959.0 kB | 61.9 X |
| quadratic_up_64 | 7.0 kB | 9.9 kB | 252.2 kB | 25.5 X |
| quadratic_up_32 | 2.1 kB | 7.1 kB | 70.8 kB | 10.0 X |
| quadratic_up_16 | 818 bytes | 5.6 kB | 23.2 kB | 4.1 X |
| quadratic_up_8 | 458 bytes | 4.9 kB | 10.2 kB | 2.1 X |
| quadratic_up_4 | 340 bytes | 4.6 kB | 6.3 kB | 1.4 X |
| quadratic_up_2 | 293 bytes | 4.4 kB | 5.1 kB | 1.2 X |
| quadratic_up_1 | 273 bytes | 4.3 kB | 4.6 kB | 1.1 X |

Table 6.7: File Sizes

# Chapter 7

# Reflection And Discussion

We ran into some problems while trying to execute all the different subtasks we took on. The original research questions that we asked ourselves is described in Section 7.1.

We started with working on using the GHC compiler as the backend for the Helium compiler. This way interoperability between Haskell code compiled with the different compilers could be achieved. For this we researched the differences between the core representations of different compilers, namely Helium, UHC and GHC, see Section 7.2. It turned out that this was not possible any longer, see Section 7.3.

After this we started working on Heap Recycling, see Section 7.4. Heap Recycling was a bigger task then anticipated and contained a lot of subtasks. One of the subtasks was parsing the heap recycling annotation the programmer wrote, which is implemented.

The next step was to implement a constraint based type-and-effect system. Annotated data-types are necessary to generate annotation constraints. The data-type annotation algorithm can be found in Section 4.3. For the type-and-effect system we needed a constraint generation and constraint solving to be implemented, see Sections 4.4 and 4.5. The constraint based type checker is now able to type-check the prelude. While working on the type checker we had some programs that could not be typed by our type checker. The Top framework returned a type for these programs otherwise the compiler would have thrown error messages and our type checker for core would not even have run. These type-bugs were introduced by the code generation phase, see Section 4.1. The problem was that Chars would be equal to Int. This resulted in a String being equal to an $[Int]$. We solved this by using the correct transform functions available in the *LVM.core* files during code generation. Other more general types where introduced with the code generation stripping the match statements of their context. The context for a match statement is completely gone when all the case arms are hidden in different let bindings. In order to overcome this hurdle we implemented a normalizing phase for core, see Chapter 5.

## 7.1 Original Research Questions

What we originally wanted to know is how much effect analysis (see Section 1.4) can have on the performance of programs written in Haskell. In order to understand the advantages these optimizations are able to unleash, we would have needed a infrastructure within the Helium compiler to test our hypothesis: *Can we turn the Helium compiler into an optimizing compiler?*. The following sub-questions needed to be answered accordingly:

- Can we implement a type-and-effect infrastructure in Helium?
- Can we instantiate the type-and-effect system for cardinality analysis?
- Can we use our analysis to generate better performing code compared to the current Helium compiler?

We overextended because in addition to this infrastructure for code analysis we including the hypothesis: *Can we use GHC-Core as the intermediate representation and GHC as our back-end?* The following sub-questions arose:

- Can we generate a GHC-Core file from Helium an compile this with GHC?
- How can we represent analysis results in GHC-Core?
- What if we can not use our analysis result in the typed part of GHC-Core?

Interoperability between code compiled with Helium and code compiled with GHC is possible because GHC can process GHC-Core files. The first step would be to compile the GHC-Core generated with Helium with the GHC compiler. It turns out that interoperability is not possible because the part of GHC that read the GHC-Core files is not available anymore, see Section 7.3.

We found out that we could add notes to GHC-Core. Analysis results that can not be represented in GHC-Core would have been represented with annotations. Strictness for example can be represented with unpacked types in the core representation. Uniqueness on the other hand can not be shown in the types of the core and would be represented as notes. This limits the usefulness of analyzing core at this level because a lot of the analysis we wanted to do would not be compatible with what GHC-Core can represent.

Then finally we stated that we wanted to use an analysis for testing our framework that didn't have many implementations. We wanted to test the framework with the sharing analysis. The reason for doing so was because a lot of research has been done with the implementation of strictness analysis in functional languages.

## 7.2 Compare Cores

This section compares different cores from different compilers. The interoperability between Helium and GHC is evaluated. The interoperability between Helium and GHC is needed to use Haskell files precompiled with GHC.

In order to understand and compare the internal representation of different compilers the different core representations used are GHC[1], Helium[2] [16] and UHC[3] [4]. The individual core languages are described in the following subsections.

### 7.2.1 UHC-Core

UHC uses multiple intermediate representations, core is one of a number of intermediate representations; the others are Grin and Silly. Silly is the lowest representation, it is imperative and is eventually translated into C or LLVM. C or LLVM are compiled to native executables. Grin is a small and strict functional language; it is at a lower level than core. Core is an untyped simplification of Haskell. Core effectively represents the untyped $\lambda$-calculus where GHC would use the typed $\lambda$-calculus described by Dijkstra et al. [4].

A quick overview of the internals is given in Figure 7.1 of the compiler. In the paper that provided this quick overview they updated the runtime garbage collector for LLVM from Boehm GC to an accurate garbage collector which compiles with the help of a shadow-stack described by van der Ende [33].

UHC-Core in Figure 7.2 is represented in memory using the data-structure given. All types are erased in this representation; that is the main difference with GHC-Core.

[1] `https://www.haskell.org/ghc`
[2] `http://foswiki.cs.uu.nl/foswiki/Helium`
[3] `http://foswiki.cs.uu.nl/foswiki/UHC`

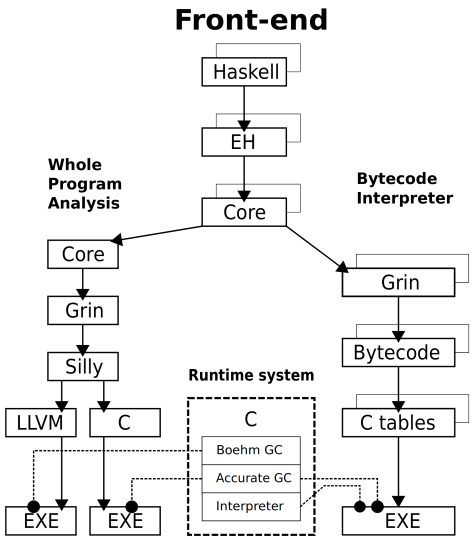


Figure 7.1: UHC internals

```
data CModule
    = Mod nm :: Name expr :: CExpr

data CExpr
    = Int int :: Int
    | Char char :: Char
    | String str :: String
    | Var name :: Name
    | Tup tag :: Tag
    | Lam arg :: Name body :: CExpr
    | App func :: CExpr arg :: CExpr
    | Case expr :: CExpr alts :: [CAlt] dflt :: CExpr
    | Let categ :: Categ binds :: [CBind] body :: CExpr

data CAlt
    = Alt pat : CPat expr :: CExpr

data CBind
    = Bind name : Name expr :: CExpr
    | FFI name : Name imp :: String ty :: Ty

data CPat
    = Var name :: Name
    | Con name :: Name tag :: Tag binds :: [CPatBind]
    | BoolExpr name :: Name cexpr :: CExpr

data CPatBind
    = Bind offset :: Int pat :: CPat
```

Figure 7.2: UHC-Core

```
type Program = [Bind Var]

data Bind b
    = NonRec b (Expr b)
    | Rec [(b, (Expr b))]

data Expr b
    = Var   Id
    | Lit   Literal
    | App   (Expr b) (Arg b)
    | Lam   b (Expr b)
    | Let   (Bind b) (Expr b)
    | Case  (Expr b) b Type [Alt b]
    | Cast  (Expr b) Coercion
    | Tick  (Tickish Id) (Expr b)
    | Type  Type
    | Coercion Coercion

type Arg b = Expr b

type Alt b = (AltCon, [b], Expr b)

data AltCon
    = DataAlt DataCon
    | LitAlt  Literal
    | DEFAULT
```

Figure 7.3: GHC-Core

### 7.2.2 GHC

GHC compiles Haskell to assembly code and links this with the precompiled runtime, that was written in C, which generates an executable. Haskell code is parsed and desugared into core. After the desugar phase the core is optimized with multiple optimization phases that are run on the core representation and it is translated into *STG*-language a language for the Spineless Tagless G-machine. The third intermediate representation is *Cmm*. This language is closely related to C, which has been used as portable assembly in the past. There are numerous C-compilers available so this way GHC could easily be ported to new platforms. From Cmm they generated a subset of C that is cross compiler compatible. These days the GHC-compiler can output assembly code directly without a C-compiler. Another option to generate an executable is with the LLVM-back-end. The LLVM-back-end compiles much slower and the generated code runs on average just as fast, but if the code depends on numeric and array heavy code the result might be much faster.

GHC-Core in Figure 7.3 is a form of typed $\lambda$-calculus called System F which they extended to System FC in Figure 7.4. System F allows universal quantifications over types. System FC extended that with type equality constraints and coercions. Type equalities are a proof that types can be coerced, thus casts are type safe between the types for which these proofs can be constructed. GHC-Core is used for the optimization passes that happen inside GHC described by Tolmach and Chevalier [32] and Eisenberg [6].

### 7.2.3 Helium

Helium uses its own runtime, the *LVM*, the Lazy Virtual Machine, which executes the LVM-assembly. The LVM-runtime looks a lot like the *JVM*-runtime, the Java Virtual Machine. You

| | | | | |
|---|---|---|---|---|
| Program | *Prog* | → | $Bind_1$ ; ... ; $Bind_n$ | $n \geq 1$ |
| Binding | *Bind* | → | *var* = *Expr* | Non-recursive |
| | | \| | rec $var_1$ = $Expr_1$ ; ... ; $var_n$ = $Expr_n$ | Recursive $n \geq 1$ |
| Expression | *Expr* | → | *Expr Atom* | Application |
| | | \| | *Expr ty* | Type application |
| | | \| | \ $var_1 \ldots var_n$ -> *Expr* | Lambda abstraction |
| | | \| | /\ $tyvar_1 \ldots tyvar_n$ -> *Expr* | Type abstraction |
| | | \| | case *Expr* of { *Alts* } | Case expression |
| | | \| | let *Bind* in *Expr* | Local definition |
| | | \| | *con* $var_1 \ldots var_n$ | Constructor $n \geq 0$ |
| | | \| | *prim* $var_1 \ldots var_n$ | Primitive $n \geq 0$ |
| | | \| | *Atom* | |
| Atoms | *Atom* | → | *var* | Variable |
| | | \| | *Literal* | Unboxed Object |
| Literals | *Literal* | → | *integer* \| *float* \| ... | |
| Alternatives | *Alts* | → | $Calt_1; \ldots; Calt_n; Default$ | $n \geq 0$ |
| | | \| | $Lalt_1; \ldots; Lalt_n; Default$ | $n \geq 0$ |
| Constr. alt | *Calt* | → | *con* $var_1 \ldots var_n$ -> *Expr* | $n \geq 0$ |
| Literal alt | *Lalt* | → | *Literal* -> *Expr* | |
| Default alt | *Default* | → | NoDefault | |
| | | \| | *var* -> *Expr* | |

Figure 7.4: System FC

need the JVM on your system to run Java code. LVM is a general back-end for (lazy) functional languages. LVM-byte-code is generated from Helium-Core (see Figure 7.5) which is an untyped $\lambda$-calculus, just like UHC-Core. The $\lambda$-calculus used by Helium is closely related to the internals of GHC-Core. GHC-Core is a typed $\lambda$-calculus, so in order to interoperate with GHC-Core we have to have type information in our Helium-Core. Communicating with GHC would give a huge benefit on multiple fronts. Libraries that can not be compiled with Helium yet could be compiled with GHC and integrated in projects built in Helium. In a best case scenario the optimizations that are performed by GHC can be incorporated into Helium.

### 7.2.4 Conclusion

We can either generate GHC-Core directly as the intermediate representation in Helium or we can define a pass from Helium-Core, which is untyped, to GHC-Core, which is typed. The second option has the downside that the type inference has to happen again.

```
-----------------------------------------------------------------
-- Modules
-----------------------------------------------------------------
type CoreModule = Module Expr
type CoreDecl   = Decl Expr

-----------------------------------------------------------------
-- Core expressions:
-----------------------------------------------------------------
data Expr       = Let       !Binds Expr
                | Match     !Id Alts
                | Ap        Expr Expr
                | Lam       !Id Expr
                | Con       !(Con Expr)
                | Var       !Id
                | Lit       !Literal

data Binds      = Rec       ![Bind]
                | Strict    !Bind
                | NonRec    !Bind

data Bind       = Bind      !Id Expr

type Alts       = [Alt]
data Alt        = Alt       !Pat Expr

data Pat        = PatCon    !(Con Tag) ![Id]
                | PatLit    !Literal
                | PatDefault

data Literal    = LitInt    !Int
                | LitDouble !Double
                | LitBytes  !Bytes

data Con tag    = ConId  !Id
                | ConTag tag !Arity
```

Figure 7.5: Helium-Core

## 7.3 Interoperability With GHC

The GHC-Compiler was able to write GHC-Core to external files. We were not able to reproduce the functionality described by Tolmach and Chevalier [32]. The interoperability code to emit and read external core was removed from the git repository[4] 5 years ago. This was the main reason that the interoperability part stranded even though we thought it to be possible because of the paper by Tolmach and Chevalier [32]. In that paper they also describe applications inside the GHC repository that can check the GHC-Core language for type errors.

The functionality is described in *Generating and compiling External Core Files*[5]. The suffix `.hcr` was used as the file extension of external-core. The ability to read external-core files is disabled since *GHC 6.8.2* and was later removed.

It was possible to add notes[6] to external-core. This could have been used by Helium to write additional information into this representation.

## 7.4 Heap Recycling

Helium is a high level core representation that makes it possible to analyse functions. During our work we did a normalization phase on the the core representation. Normalization resulted in some speed improvements, but most importantly the core representation became smaller. The smaller representation of the functions that we want to analyse results in fewer constraint being generated and solved, with a shorter runtime for our analysis as a result. The core representation has not yet an optimization phase. When a heap recycling analysis is implemented, Helium will have a real optimization phase for core in the compiler.

In order to limit the scope of Heap Recycling Analysis we wanted the programmer to annotate locations where Heap Recycling would be possible. In order to achieve this we needed a way for the programmer to annotate the programs. Therefore it was decided that the Heap Recycling Annotation needed to be added to the Helium compiler.

The outset of this research was to implement Heap Recycling in Helium. This requires a few things adding the heap recycling annotation to the syntax of the Helium compiler. It was the first milestone in the project to get heap recycling working. The annotation made by the programmer can be used when future work progresses and the analysis is implemented. The second part would be adding a type and effects system to annotate programs. These annotations would indicate how many times a certain element would be used. Lastly these usage annotations would be queried for the functions that request specialization through the annotation that are made by the programmer. If there exists a case where the function is used with non-shared argument(s) then the memory of those arguments can be reused. This can also result in partial specialization of these functions.

We want to explain which optimizations should happen after a heap recycling analysis is used to specialize the function. This can be seen as the gold standard for the heap recycling optimization phase. We will note the expected outcome of the specialization. By writing test functions and note what we expect to happen we can track that the specializations happen when expected.

In the related work in Chapter 3 the map function is already shown in Figure 3.1. Now we annotate the map function with heap recycling annotations and name it mapr as the map-recycle function in Figure 7.6. If the annotated mapr function is called with an argument that is shared, Figure 7.7, then we expect the code that would be be generated for the mapr function to be the same as for the normal map function. Whereas if the passed argument is not shared, as in Figure 7.8, then we expect that the generated code is a specialized version that reuses the memory that is provided as its second argument. This reuse of memory that would otherwise be freed results in the expected performance gain.

[4] `https://gitlab.haskell.org/ghc/ghc/commit/5bf22f06ef71f61094de7564dee770f136d5481a`
[5] `https://downloads.haskell.org/~ghc/7.4.1/docs/html/users_guide/ext-core.html`
[6] `https://downloads.haskell.org/~ghc/7.4.1/docs/html/users_guide/rewrite-rules.html`

```
mapr :: (a -> b) -> [a] -> [b]
mapr _ r@[]     = r@[]
mapr f r@(x:xs) = r@(f x : mapr f xs)
```

Figure 7.6: Map Function: with Recycling Annotations

```
main = do
    let xs = [1..10]
    let ys = mapr (1+) xs
    print xs
    print ys
```

Figure 7.7: Use Case: map with reuse annotation, no specialization possible

Other functions that are described in the Heap Recycling paper will also be tested in much the same way. Reverse in Figure 1.1 was already shown in the Introduction. Other examples are filter in Figure 7.9 and quicksort in Figure 7.12.

Filter can reuse elements when the predicate returns true, and the spine of the data is not used again as shown in Figure 7.11. It cannot reuse the data if that data is shared, this is visible in Figure 7.10. When the predicate returns false the element becomes garbage as would have been the case without the heap recycling optimization.

Quicksort in Figure 7.12 has an additional function the ## operator, it is much the same as the ++ operator in the prelude. Implementing annotated functions in the prelude that can reuse the provided memory would result in an overall speed increase for compiled programs. Use cases for reverse are in Figures 7.15 and 7.16 and for quicksort in Figures 7.13 and 7.14.

Swap2 demonstrates how there can be multiple heap recycling annotations, see Figure 7.17. This nested recycling may not be possible in the first prototypes of the optimization.

An open research question is: "What to do when a function is used with unique arguments from one location and with shared values from a second locations?" Cascading specialization could result in enormous bloat of the final binary. The worst case bloat would result in a two times increase in number of functions a specialized and non specialized version for every function. Maybe this bloat will be worth the speed and memory usage improved by the specializations. However if partial specialization is implemented any combination of reusable arguments can become its own specialized function. Maybe a limit will be necessary if this is the case, i.e. if the level is 2 then one additional function will be allowed: the non specialized implementation and the most specialized or the most used specialization will be generated.

When only one implementation per function is allowed all call sites need to be able to recycle their arguments. In Figure 7.18 internal cannot be specialized to accept reusable arguments.

```
main = do
    let xs = [1..10]
    let ys = mapr (1+) xs
    print ys
```

Figure 7.8: Use Case: map with reuse annotation, specialization possible

```
filter :: (a -> Bool) -> [a] -> [a]
filter p r@[] = r
filter p r@(x:xs)
    | p x = r@(x:filter p xs)
    | otherwise = filter p xs
```

Figure 7.9: Filter Function: with Recycling Annotations

```
main = do
    let xs = [1..10]
    let ys = filter odd xs
    print xs
    print ys
```

Figure 7.10: Use Case: filter with reuse annotation, no specialization possible

```
main = do
    let xs = [1..10]
    let ys = filter odd xs
    print ys
```

Figure 7.11: Use Case: filter with reuse annotation, specialization possible

```
qsort :: Ord a => [a] -> [a]
qsort r@[] = r
qsort r@(x:xs) = (qsort left) ## r@(x:qsort right)
    where
    (left,right) = split x xs
split :: Ord a => a -> [a] -> ([a],[a])
split k l = pivot l [] []
    where
    pivot :: [a] -> [a] -> [a] -> ([a],[a])
    pivot [] accl accr = (accl,accr)
    pivot r@(x:xs) accl accr
        | x < k = pivot xs r@(x:accl) accr
        | otherwise = pivot xs accl r@(x:accr)
(##) :: [a] -> [a] -> [a]
(##) [] rs = rs
(##) r@(l:ls) rs = r@(l:ls ## rs)
```

Figure 7.12: Quicksort Function: with Recycling Annotations

```
main = do
    let xs = [6,24,36,39,66,88,53,54,31,87] -- randomly generated sequence
    let ys = quicksort xs
    print xs
    print ys
```

Figure 7.13: Use Case: quicksort with reuse annotation, no specialization possible

```
main = do
    let xs = [6,24,36,39,66,88,53,54,31,87] -- randomly generated sequence
    let ys = quicksort xs
    print ys
```

Figure 7.14: Use Case: quicksort with reuse annotation, specialization possible

```
main = do
    let xs = [1..10]
    let ys = reverse xs
    print xs
    print ys
```

Figure 7.15: Use Case: reverse with reuse annotation, no specialization possible

```
main = do
    let xs = [1..10]
    let ys = reverse xs
    print ys
```

Figure 7.16: Use Case: reverse with reuse annotation, specialization possible

```
swap2 :: [a] -> [a]
swap2 r1@(x1:r2@(x2:xs)) = r1@(x2:r2@(x1:xs)) -- first two elements swapped in place
swap2 xs = xs -- empty and singleton list
```

Figure 7.17: Swap2 Function: with Multiple Reuse Annotations

```
main = do
    let xs = [1..10]
    let ls = internal xs -- reuse possible if internal becomes specialized
    print ls

    let ys = [1..10]
    let zs = internal ys -- reuse not possible, internal not specialized
    print ys
    print zs

internal :: [a] -> [a]
internal = mapr id
```

Figure 7.18: internal wraps a function that can reuse its arguments

## 7.5 Discussion

For the future we advise against taking on two quite different projects in one thesis. Instead focus on one project and keep everything else as plain as possible. When it turns out that the project is ready for some more variance you can expand on the foundation that has been set.

For this project that would have been limit the scope to the type-and-effect system. Implementing Counting Analysis in a real world compiler would have been enough, there have not been many studies on the subject and using the analysis result to perform a strictness optimization, which has had a lot of research already.

If it turns out that we had a working analysis we could have extended the number of optimization based on the results that were returned. Now we could be doing a fancy analysis such as Heap Recycling to increase the depth of the work. For the heap recycling the backend would have needed some modification namely the option to overwrite memory. The compiled code must know for sure when these byte code instructions are generated that there are no other references as this can not be checked by the runtime. The performance testing foundation would have been made earlier and results are generated over night.

We ran into unforeseen problems with the core of Helium. There were type inconsistencies because the code generator did not produce type correct code. Because the core was untyped before, the type incorrect core was accepted. It was untyped which is now resolved, thereby also updating the code generation. Another function of code generation is generating core from the patterns. The generated core was highly distorted with many let bindings for even the simplest patterns. The core after normalization is much easier to work with. The types of match statements can now be collected because the different cases for the same variable are now solidified. A benchmarking framework was built to benchmark programs written in Helium. The Imaginary directory of the Nofib benchmark suite has been ported to Helium.

Interoperability with GHC was based on dated research results even though they were only a few years old, maybe we could have caught this earlier.

# Chapter 8

# Conclusion

This research attempted to improve the Helium core by adding a normalizing phase to create an optimizing compiler. In this conclusion will quickly recap what was discussed and look at the research questions (see Section 8.1) and summarize the results (see Section 8.2).

In Chapter 4 we first explain the pipeline (see Section 4.1) and the available library files (see Section 4.2). Then we explain how the data-type annotation algorithm work (see Section 4.3). And lastly we discuss the constraints that are generated (see Section 4.4) and solved (see Section 4.5)

In Chapter 5 we explain why normalization needs to happen (see Section 5.1) and what the advantages are of reducing the number of constraints (see Section 5.2). The normalization is set up in the simplify routine (see Section 5.3). There are three different sub-routines; First removing renames (see Section 5.4), second normalizing match statements (see Section 5.5) with a complete example (see Section 5.8), and finally how to removes dead code (see Section 5.6).

In Chapter 6 we explain which benchmark suite we ported (see Section 6.1), the functions we selected to be executed from the prelude (see Section 6.2) and the generated match statements (see Section 6.3). It presents information about the measured virtual memory statistics (see Section 6.4). We discuss how we warm the runtime up and do the measurements (see Section 6.5). Finally it presents and disects the results that where found (see Section 6.6).

## 8.1 Answering Research Questions

In order to answer the main hypothesis we will first answer all the research questions. After these answers we will look at the main hypothesis and see if this is answered.

### What is a good baseline measurement for the Helium compiler?

During further research into the Helium compilers internals we noticed that the Helium compiler already is an optimizing compiler. It has quite a few tricks up its sleave (see Section 4.1). The code that was compiled by Helium with the already existing optimizations is used as the baseline to measure further improvements.

### What is a good benchmarking suite for the Helium compiler?

In this work we provide three ways to test the performance of Helium compiled programs. First, we ported the Imaginary directory of the Nofib benchmark suite from GHC(see Section 6.1). Second, we created a program that stresses Prelude functions that were affected by our normalization (see Section 6.2). Last, we generated match statements (see Section 6.3). These match statements were used to observe the effects of different match pattern combinations.

### Which normalizations will compact the core representation?

We researched different core representations (see Section 7.2) and how the core representation of Helium is used in the compiler. First, we found that the representation of match statements wasn't used to its full potential, namely the case arms are split over different let-bound next-clauses. Second, the variable names introduced by the programmer are represented

as renaming let-bindings in the core representation. Last, there are let-binding groups that are not used. In order to note the improvements that were possible with these normalizations we wrote normalization phases that takes these representations into account (see Chapter 5).

### How to ensure no type differences are introduced by normalizing core?

We have introduced a constraint based type-system (see Sections 4.4 and 4.5) for the Helium-core. The types that are reported by this type-system can be compared to the types generated by the TOP typing framework. With this type-system we discovered type inconsistencies introduced in the code generation.

### Does how we represent Helium programs in core affect the performance?

Yes we can confirm that the core representation of programs can largely effect the performance of these programs. The performance results from Section 6.6 are summarized in Section 8.2.

## 8.2 Summary of Results

We have measured performance differences on a wide variety of different benchmarks. From memory usage on disk, memory usage in RAM and the time it took to execute the test programs.

Let's start with storage: to be able to quickly start an application we have to ensure that the byte-code files are as small as possible. The normalizations help in reducing the code size that needs to be loaded, see Section 6.7. The size of the byte-code after normalization is almost the same as the size of the byte-code before normalization. An entirely different ratio is seen when the patterns in the match clauses become larger and longer. Here a size explosions up to 3.8 MB is seen that can be represented in just 26.8kB. Especially when doing a lot of matching the code size exploded. We observed this now because it was never stress-tested in the past.

Second up is the memory consumption, here no differences are visible. We are measuring the memory consumption of the runtime. The LVM runtime start off with a fixed amount of memory and never changes the allocated memory. The memory that is used internally can greatly differ, this could not be measured. When we were testing the Nofib benchmark which has a function that creates very deep recursive structures, namely $Exp3\_8$, we had a segmentation fault. This segmentation fault only appeared in the non-normalized code. This could indicate that the preallocated memory was enough for our implementation but resulted in an out of memory error for the non normalized code. Therefore we point out that the non-normalized code stresses the runtime more than the normalized code.

Lastly the execution times here we measured 3 different test sets. Improvements are seen across the three test sets.

The first set of test programs are from the Nofib benchmark suite. The average improvement for the Nofib benchmark suite is 3.16%. The worst case is averted, namely the stack overflow exception. The worst degradation that didn't turn in a stack overflow was 4.50% slower. The best improvement was 21% faster.

The second test tests some function that are optimized from the prelude. Here we wanted to test that we created a significant improvement in the prelude. The prelude is used in all programs that are written in Haskell. We have proved that there is no degradation in the performance with the changed core. As soon as the program executes deeper pattern matches there is an improvement.

The last test generates extreme match statement cases and views how these hold up with the new analysis. We see no difference when the pattern becomes wider. As there are more patterns in a match statement the implementation starts to increase the performance of the code. However when we reach really complicated overlapping patterns in the match statement we see the real difference. Our implementation reduces the complexity of the resulting case statements.

## 8.3 Future Work

We have made a solid core for future research in the Helium compiler. Future work on the normalization pass could still yield valuable performance. Removing the catch all clause of the match statements that match all possible constructors would allow the dead code removal phase to remove more code, namely pattern match failure cases.

On the solidified core and the constraint based type system research can continue. Adding an optimization phase to the Helium compiler would increase the performance. From the performance benchmark suite Nofib the Imaginary directory has been ported and this can be used to test the optimization. Open research questions related to this research are:

- Can we implement a type-and-effect infrastructure in Helium?
- Can we instantiate the type-and-effect system for cardinality analysis?
- Can we use our analysis to generate better performing code compared to the current Helium compiler?

After this research has been done different optimizations that can be based on the analysis result. The most notable being the Heap Recycling Optimization. We would love to see heap recycling being used in the wild. It can be used to generate highly optimized in-place algorithms without resorting to an imperative style of programming.

Another direction that could be taken is inlining of code; this is especially important when an actual application is compiled. The compilation of an application could be indicated to the compiler. All imports are known at this moment and there are no further exports. The program can load all library functions used into its own module. Now a full inline phase for the entire program could happen.

Of all functions their complete usage can now be seen. A least general type can be used to have more type information available while generating byte-code.

Functions can be specialized with respect to their predicates. This in combination with an inliner could result in inlining the $addInt$ or $addFloat$ instructions for example. These instructions are now hidden in a function that is dynamically dispatched through a dictionary.

# Bibliography


[1] Maximilian C. Bolingbroke and Simon L. Peyton Jones. Types are calling conventions. In *Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell, Haskell 2009, Edinburgh, Scotland, UK, 3 September 2009*, pages 1–12, 2009. doi: 10.1145/1596638.1596640. URL http://doi.acm.org/10.1145/1596638.1596640.

[2] Chris Clack, AJT Davie, and K Hammond. Implementation of functional languages. In *9th International Workshop, Selected Papers*, volume 1467, 1998.

[3] Mario Coppo, Ferruccio Damiani, and Paola Giannini. Strictness, totality, and non-standard-type inference. *Theoretical Computer Science*, 272(1-2):69–112, 2002.

[4] Atze Dijkstra, Jeroen Fokker, and S Swierstra. The architecture of the utrecht haskell compiler. In *Haskell'09 - Proceedings of the 2009 ACM SIGPLAN Haskell Symposium*, pages 93–104, 01 2009.

[5] Paul Downen, Luke Maurer, Zena M. Ariola, and Simon Peyton Jones. Sequent calculus as a compiler intermediate language. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*, pages 74–88, 2016. doi: 10.1145/2951913.2951931. URL http://doi.acm.org/10.1145/2951913.2951931.

[6] Richard A Eisenberg. System fc, as implemented in ghc. 2015.

[7] Andrew John Gill. *Cheap deforestation for non-strict functional languages*. PhD thesis, University of Glasgow, 1996.

[8] Kevin Glynn, Peter J. Stuckey, and Martin Sulzmann. Effective strictness analysis with horn constraints. In *Proceedings of the 8th International Symposium on Static Analysis*, SAS '01, pages 73–92, London, UK, UK, 2001. Springer-Verlag. ISBN 3-540-42314-1. URL http://dl.acm.org/citation.cfm?id=647170.718284.

[9] Jörgen Gustavsson. *A type based sharing analysis for update avoidance and optimisation*, volume 34. ACM, 1998.

[10] Jurriaan Hage and Bastiaan Heeren. Heuristics for type error discovery and recovery. In *Symposium on Implementation and Application of Functional Languages*, pages 199–216. Springer, 2006.

[11] Jurriaan Hage and Bastiaan Heeren. Strategies for solving constraints in type and effect systems. *Electronic Notes in Theoretical Computer Science*, 236:163–183, 2009.

[12] Jurriaan Hage and Stefan Holdermans. Heap recycling for lazy languages. In *Proceedings of the 2008 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, PEPM '08, pages 189–197, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-977-7. doi: 10.1145/1328408.1328436. URL http://doi.acm.org/10.1145/1328408.1328436.

[13] Jurriaan Hage, Stefan Holdermans, and Arie Middelkoop. A generic usage analysis with subeffect qualifiers. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*, ICFP '07, pages 235–246, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-815-2. doi: 10.1145/1291151.1291189. URL http://doi.acm.org/10.1145/1291151.1291189.

[14] Bastiaan Heeren, Jurriaan Hage, and S Doaitse Swierstra. Constraint based type inferencing in helium. *Immediate Applications of Constraint Programming (ACP)*, pages 57–78, 2003.

[15] Bastiaan Heeren, Jurriaan Hage, and S Doaitse Swierstra. Scripting the type inference process. In *ACM SIGPLAN Notices*, volume 38, pages 3–13. ACM, 2003.

[16] Bastiaan Heeren, Daan Leijen, and Arjan van IJzendoorn. Helium, for learning haskell. In *Proceedings of the ACM SIGPLAN Haskell Workshop (Haskell'03), Uppsala, Sweden*, page 62 – 71. ACM SIGPLAN, August 2003. URL https://www.microsoft.com/en-us/research/publication/helium-for-learning-haskell/.

[17] Bastiaan Heeren, Daan Leijen, and Arjan van IJzendoorn. Helium, for learning haskell. In *Proceedings of the 2003 ACM SIGPLAN Workshop on Haskell*, Haskell '03, pages 62–71, New York, NY, USA, 2003. ACM. ISBN 1-58113-758-3. doi: 10.1145/871895.871902. URL http://doi.acm.org/10.1145/871895.871902.

[18] Bastiaan Johannes Heeren. Top quality type error messages. 2005.

[19] Stefan Holdermans and Jurriaan Hage. Making "stricterness" more relevant. In *Proceedings of the 2010 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, PEPM '10, pages 121–130, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-727-1. doi: 10.1145/1706356.1706379. URL http://doi.acm.org/10.1145/1706356.1706379.

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

[21] Ruud Koot and Jurriaan Hage. Type-based exception analysis for non-strict higher-order functional languages with imprecise exception semantics. In *Proceedings of the 2015 Workshop on Partial Evaluation and Program Manipulation*, PEPM '15, pages 127–138, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3297-2. doi: 10.1145/2678015.2682542. URL http://doi.acm.org/10.1145/2678015.2682542.

[22] Andreas Krall. Efficient javavm just-in-time compilation. In *Parallel Architectures and Compilation Techniques, 1998. Proceedings. 1998 International Conference on*, pages 205–212. IEEE, 1998.

[23] John Launchbury, Andy Gill, John Hughes, Simon Marlow, Simon Peyton Jones, and Philip Wadler. Avoiding unnecessary updates. In *Functional Programming, Glasgow 1992*, pages 144–153. Springer, 1993.

[24] Daniel Johannes Pieter Leijen. The $\lambda$ abroad. 2003.

[25] Nicholas D Matsakis and Felix S Klock II. The rust language. In *ACM SIGAda Ada Letters*, volume 34, pages 103–104. ACM, 2014.

[26] Luke Maurer, Paul Downen, Zena M. Ariola, and Simon L. Peyton Jones. Compiling without continuations. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, pages 482–494, 2017. doi: 10.1145/3062341.3062380. URL http://doi.acm.org/10.1145/3062341.3062380.

[27] Dorit Nuzman, Sergei Dyshel, Erven Rohou, Ira Rosen, Kevin Williams, David Yuste, Albert Cohen, and Ayal Zaks. Vapor simd: Auto-vectorize once, run everywhere. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 151–160. IEEE Computer Society, 2011.

[28] Hyeong-Seok Oh, Ji Hwan Yeo, and Soo-Mook Moon. Bytecode-to-c ahead-of-time compilation for android dalvik virtual machine. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, pages 1048–1053. EDA Consortium, 2015.

[29] Rinus Plasmeijer and MCJD van Eekelen. Concurrent clean language report-version 1.3. 1998.

[30] Ilya Sergey, Dimitrios Vytiniotis, Simon L. Peyton Jones, and Joachim Breitner. Modular, higher order cardinality analysis in theory and practice. *J. Funct. Program.*, 27: e11, 2017. doi: 10.1017/S0956796817000016. URL `https://doi.org/10.1017/S0956796817000016`.

[31] Ilya Sergey, Dimitrios Vytiniotis, Simon L Peyton Jones, and Joachim Breitner. Modular, higher order cardinality analysis in theory and practice. *Journal of Functional Programming*, 27, 2017.

[32] Andrew Tolmach and Tim Chevalier. An external representation for the ghc core language (for ghc 6.10), 2009.

[33] Paul van der Ende. Extending the uhc llvm back-end: Adding support for accurate garbage collection. 2010.

[34] Hidde Verstoep and Jurriaan Hage. Polyvariant cardinality analysis for non-strict higher-order functional languages: Brief announcement. In *Proceedings of the 2015 Workshop on Partial Evaluation and Program Manipulation*, pages 139–142. ACM, 2015.

[35] HL Verstoep. Counting analyses. Master's thesis, 2013.

[36] Keith Wansbrough. Simple polymorphic usage analysis. Technical report, University of Cambridge, Computer Laboratory, 2005.

[37] Keith Wansbrough and Simon Peyton Jones. Once upon a polymorphic type. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 15–28. ACM, 1999.

[38] Andrew K Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and computation*, 115(1):38–94, 1994.

# Appendix A

# Normalize: Pseudo Haskell

```
127 altsRemovePatDefault :: Alts -> Alts
128 altsRemovePatDefault [] = []
129 altsRemovePatDefault (Alt PatDefault _:alts) = alts
130 altsRemovePatDefault (alt:alts) = alt:altsRemovePatDefault alts
```

Figure A.1: Pseudo Haskell: Remove Default Pattern From Alternatives

```haskell
idsRSingleton :: Id -> Id -> Map Id Id
idsRSingleton x y
    | "\"_\"" == show y = Map.singleton y x
    | otherwise = Map.singleton x y
```

Figure A.2: Pseudo Haskell: Ids Singleton

```haskell
idsRFromList :: [(Id,Id)] -> Map Id Id
idsRFromList [] = Map.empty
idsRFromList ((x,y):xys) = Map.union (idsRSingleton x y) (idsRFromList xys)
```

Figure A.3: Pseudo Haskell: Ids From List

```haskell
updateIds :: Map Id Id -> Expr -> Expr
updateIds idsR expr = case expr of
    Let (Strict (Bind nameB exprB)) exprL ->
        Let (Strict (Bind (replace nameB) (update exprB))) (update exprL)
    Let (NonRec (Bind nameB exprB)) exprL ->
        Let (NonRec (Bind (replace nameB) (update exprB))) (update exprL)
    Let (Rec binds) exprL ->
        let binds' = map (\(Bind nameB exprB) ->
              Bind (replace nameB) (update exprB)) binds
        in Let (Rec binds') (update exprL)
    Match nameM alts ->
        let alts' = map (\(Alt pat exprA) -> Alt pat (update exprA)) alts
        in Match (replace nameM) alts'
    Ap expr1 expr2 -> Ap (update expr1) (update expr2)
    Lam nameL exprL -> Lam (replace nameL) (update exprL)
    Con _ -> expr
    Var nameV -> Var (replace nameV)
    Lit _ -> expr
    where update expr' = updateIds idsR expr'
          replace name = Map.findWithDefault name name idsR
```

Figure A.4: Pseudo Haskell: Ids Update

```haskell
replaceDefaults :: Id -> Alts -> Alts
replaceDefaults nextClause alts =
    if Maybe.isJust findDefault
     then map (\(Alt pat exprP) -> Alt pat $
          replaceNextClause nextClause defaultExpr exprP) alts
     else alts
    where defaultExpr = Maybe.fromJust findDefault
          findDefault = findDefault' alts
          findDefault' [] = Nothing
          findDefault' (Alt PatDefault exprA:_) = Just exprA
          findDefault' (_:alts') = findDefault' alts'
```

Figure A.5: Pseudo Haskell: Replace Default Pattern In Alternatives

```
169 replaceNextClause :: Id -> Expr -> Expr -> Expr
170 replaceNextClause nextClause exprN exprP = case exprP of
171     Let (Strict (Bind nameB exprB)) exprL ->
172         Let (Strict (Bind nameB (replace exprB))) (replace exprL)
173     Let (NonRec (Bind nameB exprB)) exprL ->
174         Let (NonRec (Bind nameB (replace exprB))) (replace exprL)
175     Let (Rec binds) exprL ->
176         let binds' = map (\(Bind nameB exprB) -> Bind nameB (replace exprB)) binds
177         in Let (Rec binds') (replace exprL)
178     Match nameM alts ->
179         let alts' = map (\(Alt pat exprA) -> Alt pat (replace exprA)) alts
180         in Match nameM alts'
181     Ap expr1 _ | leftMostAp expr1 == Var nextClause -> exprN
182     Ap expr1 expr2 -> Ap (replace expr1) (replace expr2)
183     Lam nameL exprL -> Lam nameL (replace exprL)
184     Con _ -> exprP
185     Var nameV | nameV == nextClause -> exprN
186     Var _ -> exprP
187     Lit _ -> exprP
188     where replace = replaceNextClause nextClause exprN
189           leftMostAp (Ap expr1 _) = leftMostAp expr1
190           leftMostAp expr1 = expr1
```

Figure A.6: Pseudo Haskell: Replace NextClause In Expression

# Appendix B

# Normalize: Rewrite

```
lookup :: Eq a => a -> [(a,b)] -> Maybe b
lookup _ []     = Nothing
lookup k ((x,y):xys)
    | k == x    = Just y
    | otherwise = lookup k xys
```

Figure B.1: Lookup Function: Real World Example

```
\''$dictEqv87'' ''u$0'' ''u$1'' ->
let ''nextClause$.518'' ''u$0'' ''u$1'' ''$dictEqv87'' =
  let ''nextClause$.519'' =
        ''$primPatternFailPacked''
          "function bindings ranging
          from (58,1) to (61,33)
          in module Prelude.hs";
  in let ''k.520'' = ''u$0'';
     in let! ''u$1.521'' = ''u$1'';
        in match ''u$1.521'' with {
            ''::'' ''l$0'' ''l$1'' ->
              let! ''l$0.522'' = ''l$0'';
              in match ''l$0.522'' with {
                  (@0,2) x y ->
                    let ''xys.523'' = ''l$1'';
                    in let! ''guard$.524'' = ''==='' ''$dictEqv87'' ''k.520'';
                       in match ''guard$.524'' with {
                            True -> Just y;
                            _ ->
                              let! ''guard$.525'' = otherwise;
                              in match ''guard$.525'' with {
                                  True ->
                                    lookup ''$dictEqv87'' ''k.520'' ''xys.523'';
                                  _ -> ''nextClause$.519'';};};
                _ -> ''nextClause$.519'';};
            _ -> ''nextClause$.519'';};
in let! ''u$1.526'' = ''u$1'';
   in match ''u$1.526'' with {
        '':[]'' -> Nothing;
        _ -> ''nextClause$.518'' ''u$0'' ''u$1'' ''$dictEqv87'';}
```

Figure B.2: Lookup Function: Generated Core

## B.1 Remove Renames

```
let ''k.520'' = ''u$0'';
in let! ''u$1.521'' = ''u$1'';
   in match ''u$1.521'' with {
        ''::'' ''l$0'' ''l$1'' ->
          let! ''l$0.522'' = ''l$0'';
          in match ''l$0.522'' with {
              (@0,2) x y ->
                let ''xys.523'' = ''l$1'';
                in let! ''guard$.524'' = ''==='' ''$dictEqv87'' ''k.520'';
                   in match ''guard$.524'' with {
                        True -> Just y;
                        _ ->
                          let! ''guard$.525'' = otherwise;
                          in match ''guard$.525'' with {
                              True ->
                                lookup ''$dictEqv87'' ''k.520'' ''xys.523'';
                              _ -> ''nextClause$.519'';};};
              _ -> ''nextClause$.519'';};
        _ -> ''nextClause$.519'';}
```

Figure B.3: Lookup Function: Before Renaming k.520

```
let ''xys.523'' = ''l$1'';
in let! ''guard$.524'' =
          ''==='' ''$dictEqv87'' ''k.520'';
   in match ''guard$.524'' with {
        True -> Just y;
        _ ->
          let! ''guard$.525'' = otherwise;
          in match ''guard$.525'' with {
              True -> lookup ''$dictEqv87'' ''k.520'' ''xys.523'';
              _ -> ''nextClause$.519'';};}
```

Figure B.4: Lookup Function: Before Renaming xys.523

```
lookup ''$dictEqv87'' ''k.520'' ''xys.523'';
--------------------------------------------------------------------------------
lookup ''$dictEqv87'' ''u$0'' ''l$1'';
```

Figure B.5: Lookup Function: First Renaming

```
''==='' ''$dictEqv87'' ''k.520'';
--------------------------------------------------------------------------------
''==='' ''$dictEqv87'' ''u$0'';
```

Figure B.6: Lookup Function: Second Renaming

```
let! ''guard$.524'' = ''==='' ''$dictEqv87'' ''u$0'';
in match ''guard$.524'' with {
    True -> Just y;
    _ ->
      let! ''guard$.525'' = otherwise;
      in match ''guard$.525'' with {
          True -> lookup ''$dictEqv87'' ''u$0'' ''l$1'';
          _ -> ''nextClause$.519'';};}
```

Figure B.7: Lookup Function: After Renaming xys.523

```
let! ''u$1.521'' = ''u$1'';
in match ''u$1.521'' with {
    ''::'' ''l$0'' ''l$1'' ->
      let! ''l$0.522'' = ''l$0'';
      in match ''l$0.522'' with {
          (@0,2) x y ->
            let! ''guard$.524'' = ''==='' ''$dictEqv87'' ''u$0'';
            in match ''guard$.524'' with {
                True -> Just y;
                _ ->
                  let! ''guard$.525'' = otherwise;
                  in match ''guard$.525'' with {
                      True -> lookup ''$dictEqv87'' ''u$0'' ''l$1'';
                      _ -> ''nextClause$.519'';};};
          _ -> ''nextClause$.519'';};
    _ -> ''nextClause$.519'';}
```

Figure B.8: Lookup Function: After Renaming k.520

```
\''$dictEqv87'' ''u$0'' ''u$1'' ->
let ''nextClause$.518'' ''u$0'' ''u$1'' ''$dictEqv87'' =
  let ''nextClause$.519'' =
        ''$primPatternFailPacked''
          "function bindings ranging
          from (58,1) to (61,33)
          in module Prelude.hs";
  in let! ''u$1.521'' = ''u$1'';
     in match ''u$1.521'' with {
          ''::'' ''l$0'' ''l$1'' ->
            let! ''l$0.522'' = ''l$0'';
            in match ''l$0.522'' with {
                (@0,2) x y ->
                  let! ''guard$.524'' = ''==='' ''$dictEqv87'' ''u$0'';
                  in match ''guard$.524'' with {
                      True -> Just y;
                      _ ->
                        let! ''guard$.525'' = otherwise;
                        in match ''guard$.525'' with {
                            True -> lookup ''$dictEqv87'' ''u$0'' ''l$1'';
                            _ -> ''nextClause$.519'';};};
                      _ -> ''nextClause$.519'';};
          _ -> ''nextClause$.519'';};
in let! ''u$1.526'' = ''u$1'';
   in match ''u$1.526'' with {
        '':[]'' -> Nothing;
        _ -> ''nextClause$.518'' ''u$0'' ''u$1'' ''$dictEqv87'';}
```

Figure B.9: Lookup Function: Renames Removed

## B.2 Normalize Matches

```
\''$dictEqv87'' ''u$0'' ''u$1'' ->
let ''nextClause$.518'' ''u$0'' ''u$1'' ''$dictEqv87'' =
  let ''nextClause$.519'' =
        ''$primPatternFailPacked'=
          "function bindings ranging
          from (58,1) to (61,33)
          in module Prelude.hs";
  in let! ''u$1.521'' = ''u$1'';
     in match ''u$1.521'' with {
          ''::'' ''l$0'' ''l$1'' ->
            let! ''l$0.522'' = ''l$0'';
            in match ''l$0.522'' with {
                (@0,2) x y ->
                  let! ''guard$.524'' = ''==='' ''$dictEqv87'' ''u$0'';
                  in match ''guard$.524'' with {
                      True -> Just y;
                      _ ->
                        let! ''guard$.525'' = otherwise;
                        in match ''guard$.525'' with {
                            True -> lookup ''$dictEqv87'' ''u$0'' ''l$1'';
                            _ -> ''nextClause$.519'';};};
                    _ -> ''nextClause$.519'';};
              _ -> ''nextClause$.519'';};
in let! ''u$1.526'' = ''u$1'';
   in match ''u$1.526'' with {
        '':[]'' -> Nothing;
        _ -> ''nextClause$.518'' ''u$0'' ''u$1'' ''$dictEqv87'';}
```

Figure B.10: Lookup Function: Before Rewrite

```
let ''nextClause$.519'' =
      ''$primPatternFailPacked'=
        "function bindings ranging
        from (58,1) to (61,33)
        in module Prelude.hs";
in let! ''u$1.521'' = ''u$1'';
   in match ''u$1.521'' with {
        ''::'' ''l$0'' ''l$1'' ->
          let! ''l$0.522'' = ''l$0'';
          in match ''l$0.522'' with {
              (@0,2) x y ->
                let! ''guard$.524'' = ''==='' ''$dictEqv87'' ''u$0'';
                in match ''guard$.524'' with {
                    True -> Just y;
                    _ -> lookup ''$dictEqv87'' ''u$0'' ''l$1'';};
            _ -> ''nextClause$.519'';};
        _ -> ''nextClause$.519'';}
```

Figure B.11: Lookup Function: Otherwise Removed

```
''[]'' -> Nothing;
_ -> ''nextClause$.518'' ''u$0'' ''u$1'' ''$dictEqv87'';
------------------------------------------------------------------------------
''::'' ''l$0'' ''l$1'' ->
  let! ''l$0.522'' = ''l$0'';
  in match ''l$0.522'' with {
      (@0,2) x y ->
        let! ''guard$.524'' = ''==='' ''$dictEqv87'' ''u$0'';
        in match ''guard$.524'' with {
            True -> Just y;
            _ -> lookup ''$dictEqv87'' ''u$0'' ''l$1'';};
      _ -> ''nextClause$.519'';};
_ -> ''nextClause$.519'';
------------------------------------------------------------------------------
''[]'' -> Nothing;
''::'' ''l$0'' ''l$1'' ->
  let! ''l$0.522'' = ''l$0'';
  in match ''l$0.522'' with {
      (@0,2) x y ->
        let! ''guard$.524'' = ''==='' ''$dictEqv87'' ''u$0'';
        in match ''guard$.524'' with {
            True -> Just y;
            _ -> lookup ''$dictEqv87'' ''u$0'' ''l$1'';};
      _ -> ''nextClause$.519'';};
_ -> ''nextClause$.519'';
```

Figure B.12: Lookup Function: Combining Alternatives

```
\''$dictEqv87'' ''u$0'' ''u$1'' ->
let ''nextClause$.519'' =
      ''$primPatternFailPacked''
          "function bindings ranging
          from (58,1) to (61,33)
          in module Prelude.hs";
in let! ''u$1.526'' = ''u$1'';
   in match ''u$1.526'' with {
        ''[]'' -> Nothing;
        ''::'' ''l$0'' ''l$1'' ->
          let! ''l$0.522'' = ''l$0'';
          in match ''l$0.522'' with {
              (@0,2) x y ->
                let! ''guard$.524'' = ''==='' ''$dictEqv87'' ''u$0'';
                in match ''guard$.524'' with {
                    True -> Just y;
                    _ -> lookup ''$dictEqv87'' ''u$0'' ''l$1'';};
              _ -> ''nextClause$.519'';};
        _ -> ''nextClause$.519'';}
```

Figure B.13: Lookup Function: After Rewrite

## B.3 Dead Code Elimination

There is no dead code to be eliminated for the lookup function.

# Appendix C

# Timing

Table C.1: All Timing Data

| invocations | | disabled simplify | | enabled simplify | |
|---|---|---|---|---|---|
| function | input | relative | actual | relative | actual |
| EchoInt | 0 | 1.03 | $13.1 \pm 6.85\mu s$ | 1.00 | $12.7 \pm 6.46\mu s$ |
| EchoFloat | 0 | 1.02 | $36.6 \pm 16.88\mu s$ | 1.00 | $35.8 \pm 16.03\mu s$ |
| Bernoulli | 0 | 1.06 | $38.9 \pm 20.28\mu s$ | 1.00 | $36.7 \pm 17.17\mu s$ |
| Bernoulli | 1 | 1.04 | $47.5 \pm 24.22\mu s$ | 1.00 | $45.9 \pm 21.29\mu s$ |
| Bernoulli | 2 | 1.03 | $51.7 \pm 27.50\mu s$ | 1.00 | $50.2 \pm 24.91\mu s$ |
| Bernoulli | 3 | 1.05 | $38.7 \pm 18.58\mu s$ | 1.00 | $36.9 \pm 16.70\mu s$ |
| Bernoulli | 4 | 1.07 | $74.2 \pm 38.16\mu s$ | 1.00 | $69.3 \pm 32.84\mu s$ |
| Bernoulli | 5 | 1.03 | $38.9 \pm 19.89\mu s$ | 1.00 | $37.6 \pm 18.38\mu s$ |
| Bernoulli | 6 | 1.08 | $78.0 \pm 43.16\mu s$ | 1.00 | $72.2 \pm 32.82\mu s$ |
| Bernoulli | 7 | 1.04 | $38.9 \pm 19.97\mu s$ | 1.00 | $37.2 \pm 17.28\mu s$ |
| Bernoulli | 8 | 1.06 | $99.7 \pm 51.74\mu s$ | 1.00 | $94.0 \pm 45.12\mu s$ |
| Bernoulli | 9 | 1.06 | $38.8 \pm 20.36\mu s$ | 1.00 | $36.7 \pm 16.93\mu s$ |
| Bernoulli | 10 | 1.08 | $109.8 \pm 57.11\mu s$ | 1.00 | $101.5 \pm 46.36\mu s$ |
| Bernoulli | 11 | 1.07 | $41.4 \pm 21.68\mu s$ | 1.00 | $38.9 \pm 17.16\mu s$ |
| Bernoulli | 12 | 1.07 | $149.1 \pm 76.14\mu s$ | 1.00 | $139.2 \pm 61.61\mu s$ |
| Bernoulli | 13 | 1.04 | $41.5 \pm 20.82\mu s$ | 1.00 | $39.8 \pm 19.84\mu s$ |
| Bernoulli | 14 | 1.06 | $137.2 \pm 68.49\mu s$ | 1.00 | $129.1 \pm 63.02\mu s$ |
| Bernoulli | 15 | 1.07 | $41.4 \pm 22.04\mu s$ | 1.00 | $38.6 \pm 17.76\mu s$ |
| DigitsOfE1 | 0 | 1.03 | $16.2 \pm 8.23\mu s$ | 1.00 | $15.7 \pm 7.50\mu s$ |
| DigitsOfE1 | 1 | 1.05 | $19.5 \pm 9.63\mu s$ | 1.00 | $18.6 \pm 7.99\mu s$ |
| DigitsOfE1 | 2 | 1.04 | $24.6 \pm 11.07\mu s$ | 1.00 | $23.8 \pm 10.72\mu s$ |
| DigitsOfE1 | 3 | 1.05 | $30.6 \pm 14.06\mu s$ | 1.00 | $29.1 \pm 11.66\mu s$ |
| DigitsOfE1 | 4 | 1.03 | $35.7 \pm 16.85\mu s$ | 1.00 | $34.6 \pm 15.16\mu s$ |
| DigitsOfE1 | 5 | 1.03 | $40.5 \pm 17.69\mu s$ | 1.00 | $39.3 \pm 16.83\mu s$ |
| DigitsOfE1 | 6 | 1.03 | $46.5 \pm 20.58\mu s$ | 1.00 | $45.4 \pm 19.82\mu s$ |
| DigitsOfE1 | 7 | 1.03 | $52.0 \pm 24.15\mu s$ | 1.00 | $50.6 \pm 20.95\mu s$ |
| DigitsOfE1 | 8 | 1.01 | $57.0 \pm 26.21\mu s$ | 1.00 | $56.4 \pm 26.24\mu s$ |
| DigitsOfE1 | 9 | 1.04 | $63.9 \pm 30.17\mu s$ | 1.00 | $61.6 \pm 27.28\mu s$ |
| DigitsOfE1 | 10 | 1.04 | $71.5 \pm 33.75\mu s$ | 1.00 | $68.9 \pm 30.20\mu s$ |
| DigitsOfE2 | 0 | 1.04 | $16.0 \pm 8.22\mu s$ | 1.00 | $15.5 \pm 7.07\mu s$ |
| DigitsOfE2 | 1 | 1.05 | $18.0 \pm 8.86\mu s$ | 1.00 | $17.2 \pm 7.32\mu s$ |
| DigitsOfE2 | 2 | 1.03 | $22.7 \pm 11.03\mu s$ | 1.00 | $22.1 \pm 9.56\mu s$ |

continued...

. . . continued

| invocations | | disabled simplify | | enabled simplify | |
|---|---|---|---|---|---|
| function | input | relative | actual | relative | actual |
| DigitsOfE2 | 3 | 1.05 | $23.9 \pm 11.01\mu s$ | 1.00 | $22.7 \pm 9.39\mu s$ |
| DigitsOfE2 | 4 | 1.04 | $26.9 \pm 11.60\mu s$ | 1.00 | $25.7 \pm 10.78\mu s$ |
| DigitsOfE2 | 5 | 1.21 | $35.0 \pm 12.96\mu s$ | 1.00 | $28.8 \pm 13.49\mu s$ |
| DigitsOfE2 | 6 | 1.02 | $32.1 \pm 15.91\mu s$ | 1.00 | $31.4 \pm 13.74\mu s$ |
| DigitsOfE2 | 7 | 1.00 | $33.8 \pm 14.89\mu s$ | 1.02 | $34.3 \pm 14.90\mu s$ |
| DigitsOfE2 | 8 | 1.00 | $36.9 \pm 15.76\mu s$ | 1.00 | $36.8 \pm 15.76\mu s$ |
| DigitsOfE2 | 9 | 1.02 | $39.2 \pm 17.53\mu s$ | 1.00 | $38.4 \pm 15.84\mu s$ |
| DigitsOfE2 | 10 | 1.03 | $44.8 \pm 20.36\mu s$ | 1.00 | $43.5 \pm 19.05\mu s$ |
| Exp3_8 | 0 | 1.01 | $13.3 \pm 6.54\mu s$ | 1.00 | $13.2 \pm 7.09\mu s$ |
| Exp3_8 | 1 | 1.05 | $16.6 \pm 8.42\mu s$ | 1.00 | $15.8 \pm 7.30\mu s$ |
| Exp3_8 | 2 | 1.05 | $19.0 \pm 8.77\mu s$ | 1.00 | $18.0 \pm 9.04\mu s$ |
| Exp3_8 | 3 | 1.06 | $28.7 \pm 13.49\mu s$ | 1.00 | $27.1 \pm 11.43\mu s$ |
| Exp3_8 | 4 | 1.04 | $49.5 \pm 23.85\mu s$ | 1.00 | $47.5 \pm 20.02\mu s$ |
| Exp3_8 | 5 | 1.08 | $122.7 \pm 53.39\mu s$ | 1.00 | $113.8 \pm 51.24\mu s$ |
| Exp3_8 | 6 | 1.05 | $337.2 \pm 123.31\mu s$ | 1.00 | $321.3 \pm 116.85\mu s$ |
| Exp3_8 | 7 | | stack overflow | 1.00 | $1034.6 \pm 222.90\mu s$ |
| Exp3_8 | 8 | | stack overflow | 1.00 | $3.3 \pm 0.56ms$ |
| Exp3_8 | 9 | | stack overflow | 1.00 | $12.4 \pm 0.76ms$ |
| Exp3_8 | 10 | | stack overflow | 1.00 | $57.5 \pm 4.74ms$ |
| Integrate | 0 | 1.03 | $18.2 \pm 8.75\mu s$ | 1.00 | $17.6 \pm 8.43\mu s$ |
| Integrate | 1 | 1.01 | $19.2 \pm 8.56\mu s$ | 1.00 | $19.1 \pm 8.88\mu s$ |
| Integrate | 2 | 1.03 | $19.9 \pm 10.36\mu s$ | 1.00 | $19.3 \pm 8.22\mu s$ |
| Integrate | 3 | 1.03 | $20.4 \pm 9.30\mu s$ | 1.00 | $19.8 \pm 8.71\mu s$ |
| Integrate | 4 | 1.04 | $20.9 \pm 10.41\mu s$ | 1.00 | $20.1 \pm 10.02\mu s$ |
| Integrate | 5 | 1.05 | $21.4 \pm 10.38\mu s$ | 1.00 | $20.3 \pm 8.58\mu s$ |
| Nfib | 0 | 1.10 | $14.1 \pm 12.77\mu s$ | 1.00 | $12.8 \pm 5.94\mu s$ |
| Nfib | 1 | 1.02 | $13.0 \pm 6.46\mu s$ | 1.00 | $12.7 \pm 6.13\mu s$ |
| Nfib | 2 | 1.04 | $13.7 \pm 7.40\mu s$ | 1.00 | $13.2 \pm 6.03\mu s$ |
| Nfib | 3 | 1.03 | $14.5 \pm 7.08\mu s$ | 1.00 | $14.1 \pm 6.81\mu s$ |
| Nfib | 4 | 1.02 | $16.2 \pm 7.95\mu s$ | 1.00 | $15.8 \pm 8.51\mu s$ |
| Nfib | 5 | 1.02 | $19.0 \pm 8.49\mu s$ | 1.00 | $18.7 \pm 7.99\mu s$ |
| Nfib | 6 | 1.02 | $23.4 \pm 11.85\mu s$ | 1.00 | $22.9 \pm 11.43\mu s$ |
| Nfib | 7 | 1.02 | $25.0 \pm 11.44\mu s$ | 1.00 | $24.4 \pm 10.33\mu s$ |
| Nfib | 8 | 1.02 | $33.9 \pm 16.29\mu s$ | 1.00 | $33.1 \pm 13.85\mu s$ |
| Nfib | 9 | 1.02 | $48.3 \pm 22.58\mu s$ | 1.00 | $47.4 \pm 20.11\mu s$ |
| Nfib | 10 | 1.02 | $66.5 \pm 33.10\mu s$ | 1.00 | $65.1 \pm 26.82\mu s$ |
| Nfib | 11 | 1.01 | $95.6 \pm 42.50\mu s$ | 1.00 | $94.9 \pm 40.48\mu s$ |
| Nfib | 12 | 1.01 | $139.2 \pm 56.19\mu s$ | 1.00 | $138.3 \pm 53.24\mu s$ |
| Nfib | 13 | 1.02 | $216.9 \pm 94.63\mu s$ | 1.00 | $212.3 \pm 77.86\mu s$ |
| Nfib | 14 | 1.01 | $336.1 \pm 133.35\mu s$ | 1.00 | $331.3 \pm 124.42\mu s$ |
| Nfib | 15 | 1.01 | $506.5 \pm 168.67\mu s$ | 1.00 | $501.2 \pm 160.56\mu s$ |
| Nfib | 16 | 1.01 | $791.4 \pm 258.84\mu s$ | 1.00 | $783.8 \pm 245.57\mu s$ |
| Nfib | 17 | 1.00 | $1271.2 \pm 484.60\mu s$ | 1.00 | $1265.1 \pm 476.56\mu s$ |
| Nfib | 18 | 1.00 | $1874.9 \pm 517.05\mu s$ | 1.00 | $1867.5 \pm 468.65\mu s$ |
| Nfib | 19 | 1.01 | $2.9 \pm 0.49ms$ | 1.00 | $2.8 \pm 0.48ms$ |
| Nfib | 20 | 1.00 | $4.4 \pm 0.50ms$ | 1.00 | $4.4 \pm 0.50ms$ |
| Nfib | 21 | 1.00 | $7.0 \pm 0.57ms$ | 1.00 | $6.9 \pm 0.52ms$ |
| Nfib | 22 | 1.01 | $11.1 \pm 0.64ms$ | 1.00 | $11.0 \pm 0.53ms$ |
| Nfib | 23 | 1.00 | $17.6 \pm 0.68ms$ | 1.00 | $17.6 \pm 0.63ms$ |

continued. . .

. . . continued

| invocations | | disabled simplify | | enabled simplify | |
|---|---|---|---|---|---|
| function | input | relative | actual | relative | actual |
| Nfib | 24 | 1.01 | $28.5 \pm 0.90ms$ | 1.00 | $28.4 \pm 0.90ms$ |
| Nfib | 25 | 1.00 | $45.7 \pm 1.08ms$ | 1.00 | $45.7 \pm 1.29ms$ |
| Paraffins | 0 | 1.05 | $24.8 \pm 13.41\mu s$ | 1.00 | $23.7 \pm 11.46\mu s$ |
| Paraffins | 1 | 1.07 | $58.3 \pm 30.22\mu s$ | 1.00 | $54.5 \pm 25.81\mu s$ |
| Paraffins | 2 | 1.05 | $94.6 \pm 49.04\mu s$ | 1.00 | $90.3 \pm 40.35\mu s$ |
| Paraffins | 3 | 1.03 | $143.1 \pm 65.46\mu s$ | 1.00 | $139.3 \pm 62.42\mu s$ |
| Paraffins | 4 | 1.06 | $248.1 \pm 119.03\mu s$ | 1.00 | $234.7 \pm 86.77\mu s$ |
| Paraffins | 5 | 1.10 | $426.6 \pm 189.56\mu s$ | 1.00 | $388.7 \pm 130.19\mu s$ |
| Paraffins | 6 | 1.03 | $727.2 \pm 280.84\mu s$ | 1.00 | $707.0 \pm 251.68\mu s$ |
| Paraffins | 7 | 1.03 | $1330.5 \pm 567.63\mu s$ | 1.00 | $1290.2 \pm 507.59\mu s$ |
| Paraffins | 8 | 1.01 | $2.2 \pm 0.54ms$ | 1.00 | $2.2 \pm 0.51ms$ |
| Paraffins | 9 | 1.02 | $4.1 \pm 0.53ms$ | 1.00 | $4.0 \pm 0.51ms$ |
| Paraffins | 10 | 1.03 | $8.0 \pm 0.60ms$ | 1.00 | $7.8 \pm 0.56ms$ |
| Paraffins | 11 | 1.01 | $16.6 \pm 0.63ms$ | 1.00 | $16.4 \pm 0.51ms$ |
| Paraffins | 12 | 1.01 | $35.7 \pm 0.67ms$ | 1.00 | $35.5 \pm 0.81ms$ |
| Paraffins | 13 | 1.01 | $80.3 \pm 0.99ms$ | 1.00 | $79.4 \pm 1.08ms$ |
| Paraffins | 14 | 1.01 | $149.5 \pm 1.42ms$ | 1.00 | $147.5 \pm 1.63ms$ |
| Primes | 0 | 1.01 | $13.2 \pm 6.00\mu s$ | 1.00 | $13.1 \pm 5.68\mu s$ |
| Primes | 1 | 1.06 | $50.2 \pm 27.77\mu s$ | 1.00 | $47.6 \pm 24.41\mu s$ |
| Primes | 2 | 1.06 | $89.6 \pm 73.54\mu s$ | 1.00 | $84.8 \pm 68.76\mu s$ |
| Primes | 3 | 1.12 | $171.6 \pm 202.03\mu s$ | 1.00 | $152.9 \pm 171.18\mu s$ |
| Primes | 4 | 1.02 | $272.2 \pm 378.73\mu s$ | 1.00 | $266.1 \pm 374.00\mu s$ |
| Primes | 5 | 1.15 | $441.9 \pm 718.38\mu s$ | 1.00 | $385.7 \pm 610.49\mu s$ |
| Primes | 6 | 1.19 | $723.1 \pm 1307.21\mu s$ | 1.00 | $609.8 \pm 1061.36\mu s$ |
| Primes | 7 | 1.02 | $901.8 \pm 1661.60\mu s$ | 1.00 | $886.9 \pm 1643.72\mu s$ |
| Primes | 8 | 1.01 | $1368.3 \pm 2650.09\mu s$ | 1.00 | $1351.9 \pm 2645.40\mu s$ |
| Primes | 9 | 1.00 | $2.2 \pm 4.37ms$ | 1.00 | $2.2 \pm 4.40ms$ |
| Primes | 10 | 1.01 | $2.7 \pm 5.59ms$ | 1.00 | $2.6 \pm 5.54ms$ |
| Primes | 11 | 1.01 | $4.7 \pm 10.08ms$ | 1.00 | $4.6 \pm 9.88ms$ |
| Primes | 12 | 1.00 | $10.9 \pm 23.96ms$ | 1.01 | $11.1 \pm 24.55ms$ |
| Primes | 13 | 1.01 | $13.0 \pm 28.74ms$ | 1.00 | $12.9 \pm 28.41ms$ |
| Primes | 14 | 1.00 | $15.0 \pm 33.13ms$ | 1.00 | $15.0 \pm 33.63ms$ |
| Primes | 15 | 1.01 | $17.3 \pm 38.26ms$ | 1.00 | $17.2 \pm 37.99ms$ |
| Queens | 0 | 1.03 | $13.3 \pm 6.39\mu s$ | 1.00 | $13.0 \pm 5.81\mu s$ |
| Queens | 1 | 1.06 | $15.6 \pm 8.76\mu s$ | 1.00 | $14.7 \pm 6.91\mu s$ |
| Queens | 2 | 1.01 | $19.3 \pm 9.32\mu s$ | 1.00 | $19.1 \pm 9.12\mu s$ |
| Queens | 3 | 1.04 | $33.1 \pm 16.79\mu s$ | 1.00 | $31.9 \pm 14.16\mu s$ |
| Queens | 4 | 1.02 | $80.6 \pm 34.51\mu s$ | 1.00 | $79.0 \pm 34.12\mu s$ |
| Queens | 5 | 1.02 | $331.2 \pm 130.23\mu s$ | 1.00 | $324.0 \pm 122.30\mu s$ |
| Queens | 6 | 1.02 | $1392.6 \pm 509.77\mu s$ | 1.00 | $1370.1 \pm 502.53\mu s$ |
| Queens | 7 | 1.01 | $5.3 \pm 0.52ms$ | 1.00 | $5.2 \pm 0.50ms$ |
| Queens | 8 | 1.02 | $24.9 \pm 0.63ms$ | 1.00 | $24.4 \pm 0.61ms$ |
| Queens | 9 | 1.02 | $124.1 \pm 1.18ms$ | 1.00 | $121.7 \pm 0.99ms$ |
| WheelSieve1 | 0 | 1.01 | $13.1 \pm 6.97\mu s$ | 1.00 | $12.9 \pm 5.93\mu s$ |
| WheelSieve1 | 1 | 1.02 | $13.6 \pm 7.13\mu s$ | 1.00 | $13.3 \pm 6.23\mu s$ |
| WheelSieve1 | 2 | 1.04 | $14.0 \pm 7.64\mu s$ | 1.00 | $13.5 \pm 6.56\mu s$ |
| WheelSieve1 | 3 | 1.04 | $14.6 \pm 7.06\mu s$ | 1.00 | $14.1 \pm 7.46\mu s$ |
| WheelSieve1 | 4 | 1.03 | $17.0 \pm 8.30\mu s$ | 1.00 | $16.6 \pm 8.13\mu s$ |
| WheelSieve1 | 5 | 1.00 | $16.9 \pm 7.92\mu s$ | 1.00 | $16.9 \pm 8.73\mu s$ |

continued. . .

. . . continued

| invocations | | disabled simplify | | enabled simplify | |
|---|---|---|---|---|---|
| function | input | relative | actual | relative | actual |
| WheelSieve2 | 0 | 1.02 | $13.3 \pm 7.32\mu s$ | 1.00 | $13.0 \pm 5.74\mu s$ |
| WheelSieve2 | 1 | 1.02 | $13.5 \pm 6.41\mu s$ | 1.00 | $13.3 \pm 6.34\mu s$ |
| WheelSieve2 | 2 | 1.02 | $13.8 \pm 6.41\mu s$ | 1.00 | $13.5 \pm 7.14\mu s$ |
| WheelSieve2 | 3 | 1.05 | $14.5 \pm 7.41\mu s$ | 1.00 | $13.8 \pm 6.23\mu s$ |
| WheelSieve2 | 4 | 1.01 | $16.9 \pm 8.01\mu s$ | 1.00 | $16.7 \pm 8.14\mu s$ |
| WheelSieve2 | 5 | 1.00 | $16.8 \pm 8.33\mu s$ | 1.00 | $16.8 \pm 8.42\mu s$ |
| X2n1 | 0 | 1.04 | $14.1 \pm 7.83\mu s$ | 1.00 | $13.5 \pm 6.32\mu s$ |
| X2n1 | 1 | 1.00 | $5.5 \pm 0.66ms$ | 1.07 | $5.9 \pm 1.05ms$ |
| X2n1 | 2 | 1.00 | $12.2 \pm 0.59ms$ | 1.08 | $13.2 \pm 4.00ms$ |
| X2n1 | 3 | 1.00 | $19.9 \pm 0.69ms$ | 1.00 | $19.8 \pm 0.86ms$ |
| X2n1 | 4 | 1.00 | $27.4 \pm 0.93ms$ | 1.00 | $27.3 \pm 1.34ms$ |
| X2n1 | 5 | 1.00 | $36.7 \pm 0.74ms$ | 1.00 | $36.6 \pm 1.06ms$ |

# Appendix D

# F-Tests

| F-Test Two-Sample for variance | enabled simplify | disabled simplify |
|---|---|---|
| Mean | 13608 | 13554 |
| Variance | 120168 | 71388 |
| Observations | 120 | 120 |
| df | 119 | 119 |
| $F_{stat}$ | 1.6833 | |
| $P(F \leq f)$ one-tail | 2.4111E-3 | |
| $F_{crit}$ one-tail | 1.3536 | |

Table D.1: F-Test Input: 0

| **F-Test Two-Sample for variance** | **enabled simplify** | **disabled simplify** |
|---|---|---|
| Mean | 37874645 | 38696236 |
| Variance | 4.4093E+12 | 1.6657E+11 |
| Observations | 120 | 120 |
| df | 119 | 119 |
| $F_{stat}$ | 26.472 | |
| $P(F \leq f)$ one-tail | 7.0025E-53 | |
| $F_{crit}$ one-tail | 1.3536 | |

Table D.2: F-Test Input: 13847

| **F-Test Two-Sample for variance** | **enabled simplify** | **disabled simplify** |
|---|---|---|
| Mean | 75952407 | 79139739 |
| Variance | 5.6222E+11 | 1.5587E+13 |
| Observations | 120 | 120 |
| df | 119 | 119 |
| $F_{stat}$ | 27.724 | |
| $P(F \leq f)$ one-tail | 5.4286E-54 | |
| $F_{crit}$ one-tail | 1.3536 | |

Table D.3: F-Test Input: 27693