A modern look at GRIN, an optimizing functional language back end

Péter Dávid Podlovics[1], Csaba Hruska[2], and Andor Pénzes[2]

[1] Eötvös Loránd University, Budapest, Hungary
peter.d.podlovics@gmail.com
[2] {csaba.hruska, andor.penzes}@gmail.com

Abstract. GRIN is short for Graph Reduction Intermediate Notation, a modern back end for lazy functional languages. Most of the currently available compilers for such languages share a common flaw: they can only optimize programs on a per-module basis. The GRIN framework allows for interprocedural whole program analysis, enabling optimizing code transformations across functions and modules as well.
Some implementations of GRIN already exist, but most of them were developed only for experimentation purposes. Thus, they either compromise on low level efficiency, or contain ad hoc modifications compared to the original specification.
Our goal is to provide a full-fledged implementation of GRIN by combining the currently available best technologies like LLVM, and evaluate the framework's effectiveness by measuring how the optimizer improves the performance of certain programs. We also present some improvements to the already existing components of the framework. Some of these improvements include a typed representation for the intermediate language and an interprocedural program optimization, the dead data elimination.

Keywords: grin · compiler · whole program optimization · intermediate representation · dead code elimination

1 Introduction

Over the last few years, the functional programming paradigm has become even more popular and prominent than it was before. More and more industrial applications emerge, the paradigm itself keeps evolving, existing functional languages are being refined day by day, and even completely new languages appear. Yet, it seems the corresponding compiler technology lacks behind a bit.

Functional languages come with a multitude of interesting features that allow us to write programs on higher abstraction levels. Some of these features include higher-order functions, laziness and sophisticated type systems based on SystemFC [1], some even supporting dependent types. Although these features make writing code more convenient, they also complicate the compilation process.

Compiler front ends usually handle these problems very well, but the back ends often struggle to produce efficient low level code. The reason for this is

that back ends have a hard time optimizing code containing *functional artifacts*. These functional artifacts are the by-products of high-level language features mentioned earlier. For example, higher-order functions can introduce unknown function calls and laziness can result in implicit value evaluation which can prove to be very hard to optimize. As a consequence, compilers generally compromise on low level efficiency for high-level language features.

Moreover, the paradigm itself also encourages a certain programming style which further complicates the situation. Functional code usually consist of many smaller functions, rather than fewer big ones. This style of coding results in more composable programs, but also presents more difficulties for compilation, since optimizing only individual functions is no longer sufficient.

In order to resolve these problems, we need a compiler back end that can optimize across functions as well as allow the optimization of laziness in some way. Also, it would be beneficial if the back end could theoretically handle any front end language.

2 Graph Reduction Intermediate Notation

GRIN is short for *Graph Reduction Intermediate Notation*. GRIN consists of an intermediate representation language (IR in the followings) as well as the entire compiler back end framework built around it. GRIN tries to resolve the issues highlighted in Section 1 by using interprocedural whole program optimization.

Interprocedural program analysis is a type of data-flow analysis that propagates information about certain program elements through function calls. Using interprocedural analyses instead of intraprocedural ones, allows for optimizations across functions. This means the framework can handle the issue of large sets of small interconnecting functions presented by the composable programming style.

Whole program analysis enables optimizations across modules. This type of data-flow analysis has all the available information about the program at once. As a consequence, it is possible to analyze and optimize global functions. With the help of whole program analysis, laziness can be made explicit. In fact, the evaluation of suspended computations in GRIN is done by an ordinary function called `eval`. This is a global function uniquely generated for each program, meaning it can be optimized just like any other function by using whole program analysis.

Finally, since the analyses and optimizations are implemented on a general intermediate representation, all other languages can benefit from the features provided by the GRIN back end. The intermediate layer of GRIN between the front end language and the low level machine code serves the purpose of eliminating functional artifacts from programs. This is achieved by using optimizing program transformations specific to the GRIN IR and functional languages in general. The simplified programs can then be optimized further by using conventional techniques already available. For example, it is possible to compile GRIN

to LLVM and take advantage of an entire compiler framework providing a huge array of very powerful tools and features.

3 Related Work

This section will introduce the reader to the state-of-the-art concerning functional language compiler technologies and whole program optimization. It will compare these systems' main goals, advantages, drawbacks and the techniques they use.

3.1 The Glasgow Haskell Compiler

GHC [2] is the de facto Haskell compiler. It is an industrial strength compiler supporting Haskell2010 with a multitude of language extensions. It has full support for multi-threading, asynchronous exception handling, incremental compilation and software transactional memory.

GHC is the most feature-rich stable Haskell compiler. However, its optimizer part is lacking in two respects. Firstly, neither of its intermediate representations (STG and Core) can express laziness explicitly, which means that the strictness analysis cannot be as optimal as it could be. Secondly, GHC only supports optimization on a per-module basis by default, and only optimizes across modules after inlining certain specific functions. This can drastically limit the information available for the optimization passes, hence decreasing their efficiency. The following sections will show alternative compilation techniques to resolve the issues presented above.

3.2 GRIN

Graph Reduction Intermediate Notation is an intermediate representation for lazy[1] functional languages. Due to its simplicity and high expressive power, it was utilized by several compiler back ends.

Boquist The original GRIN framework was developed by U. Boquist, and first described in the article [3], then in his PhD thesis [4]. This version of GRIN used the Chalmers Haskell-B Compiler [5] as its front end and RISC as its back end. The main focus of entire framework is to produce highly efficient machine code from high-level lazy functional programs through a series of optimizing code transformations. At that time, Boquist's implementation of GRIN already compared favorably to the existing Glasgow Haskell Compiler of version 4.01.

The language itself has very simple syntax and semantics, and is capable of explicitly expressing laziness. It only has very few built-in instructions (`store`, `fetch` and `update`) which can be interpreted in two ways. Firstly, they can be seen as simple heap operations; secondly, they can represent graph reduction

[1] Strict semantics can be expressed as well.

semantics [6]. For example, we can imagine `store` creating a new node, and `update` reducing those nodes.

GRIN also supports whole program optimization. Whole program optimization is a compiler optimization technique that uses information regarding the entire program instead of localizing the optimizations to functions or translation units. One of the most important whole program analyses used by the framework is the heap-points-to analysis, a variation of Andersen's pointer analysis [7].

UHC The Utrecht Haskell Compiler [8] is a completely standalone Haskell compiler with its own front end. The main idea behind UHC is to use attribute grammars to handle the ever-growing complexity of compiler construction in an easily manageable way. Mainly, the compiler is being used for education, since utilizing a custom system, the programming environment can be fine-tuned for the students, and the error messages can be made more understandable.

UHC also uses GRIN as its IR for its back-end part, however the main focus has diverted from low level efficiency, and broadened to the spectrum of the entire compiler framework. It also extended the original IR with synchronous exception handling by introducing new syntactic constructs for `try/catch` blocks [9]. Also, UHC can generate code for many different targets including LLVM [10], .Net, JVM and JavaScript.

JHC JHC [11] is another complete compiler framework for Haskell, developed by John Meacham. JHC's goal is to generate not only efficient, but also very compact code without the need of any runtime. The generated code only has to rely on certain system calls. JHC also has its own front end and back end just like UHC, but they serve different purposes.

The front end of JHC uses a very elaborate type system called the pure type system [12,13]. In theory, the pure type system can be seen as a generalization of the lambda cube [14], in practice it behaves similarly to the Glasgow Haskell Compiler's Core representation. For example, similar transformations can be implemented on them.

For its intermediate representation, JHC uses an alternate version of GRIN. Meacham made several modifications to the original specification of GRIN. Some of the most relevant additions are mutable variables, memory regions (heap and stack) and throw-only IO exceptions. JHC's exceptions are rather simple compared to those of UHC, since they can only be thrown, but never caught.

JHC generates completely portable ISO C from the intermediate GRIN code.

AJHC Originally, AJHC [15] was a fork of JHC, but later it was remerged with all of its functionalities. The main goal of AJHC was to utilize formal methods in systems programming. It was used implementing a NetBSD sound driver in high-level Haskell.

LHC The LLVM Haskell Compiler [16] is Haskell compiler made from reusable libraries using JHC-style GRIN as its intermediate representation. As its name suggests, it generates LLVM IR code from the intermediate GRIN.

3.3 Other Intermediate Representations

GRIN is not the only IR available for functional languages. In fact, it is not even the most advanced one. Other representations can either be structurally different or can have different expressive power. For example GRIN and LLVM are both structurally and expressively different representations, because GRIN has monadic structure, while LLVM uses basic blocks, and while GRIN has sum types, LLVM has vector instructions. In general, different design choices can open up different optimization opportunities.

Intel Research Compiler The Intel Labs Haskell Research Compiler [17] was a result of a long running research project focusing on functional language compilation. The project's main goal was to generate very efficient code for numerical computations utilizing whole program optimization.

The compiler reused the front end part of GHC, and worked with the external Core representation provided by it. Its optimizer part was written in MLton and was a general purpose compiler back end for strict functional languages. Differently from GRIN, it used basic blocks which can open up a whole spectrum of new optimization opportunities. Furthermore, instead of whole program defunctionalization (the generation of global `eval`), their compiler used function pointers and data-flow analysis techniques to globally analyze the program. They also supported synchronous exceptions and multi-threading.

One of their most relevant optimizations was the SIMD vectorization pass [18]. Using this optimization, they could transform sequential programs into vectorized ones. In conjunction with their other optimizations, they achieved performance metrics comparable to native C [19].

MLton MLton [20] is a widely used Standard ML compiler. It also uses whole program optimization, and focuses on efficiency.

MLton has a wide array of distinct intermediate representations, each serving a different purpose. Each IR can express a certain aspect of the language more precisely than the others, allowing for more convenient implementation of the respective analyses and transformations. They use a technique similar to defunctionalization called 0CFA, a higher-order control flow analysis. This method serves a very similar purpose to defunctionalization, but instead of following function tags, it tracks function closures. Also, 0CFA can be generalized to k-CFA, where k represents the number of different contexts the analysis distinguishes. The variant used by MLton distinguishes zero different contexts, meaning it is a *context insensitive* analysis. The main advantage of this technique is that it can be applied to higher-order languages as well.

Furthermore, MLton supports contification [21], a control-flow based transformation, which turns function calls into continuations. This can expose a lot of additional control-flow information, allowing for a broad range of optimizations such as tail recursive function call optimization.

As for its back end, MLton has its own native code generator, but it can also generate LLVM IR code [22].

4 Compiling to LLVM

LLVM is a collection of compiler technologies consisting of an intermediate representation called the LLVM IR, a modularly built compiler framework and many other tools built on these technologies. This section discusses the benefits and challenges of compiling GRIN to LLVM.

4.1 Benefits and Challenges

The main advantage LLVM has over other CISC and RISC based languages lies in its modular design and library based structure. The compiler framework built around LLVM is entirely customizable and can generate highly optimized low level machine code for most architectures. Furthermore, it offers a vast range of tools and features out of the box, such as different debugging tools or compilation to WebAssembly.

However, compiling unrefined functional code to LLVM does not yield the results one would expect. Since LLVM was mainly designed for imperative languages, functional programs may prove to be difficult to optimize. The reason for this is that functional artifacts or even just the general structuring of functional programs can render conventional optimization techniques useless.

While LLVM acts as a transitional layer between architecture independent, and architecture specific domains, GRIN serves the same purpose for the functional and imperative domains. Figure 4.1 illustrates this domain separtion. The purpose of GRIN is to eliminate functional artifacts and restructure functional programs in a way so that they can be efficiently optimized by conventional techniques.

The main challenge of compiling GRIN to LLVM has to do with the discrepancy between the respective type systems of these languages: GRIN is untyped, while LLVM has static typing. In order to make compilation to LLVM possible[1], we need a typed representation for GRIN as well. Fortunately, this problem can be circumvented by implementing a type inference algorithm for the language. To achieve this, we can extend an already existing component of the framework, the heap points-to data-flow analysis.

[1] As a matter of fact, compiling untyped GRIN to LLVM *is* possible, since only the registers are statically typed in LLVM, the memory is not. So in principle, if all variables were stored in memory, generating LLVM code from untyped GRIN would be plausible. However, this approach would prove to be very inefficient.

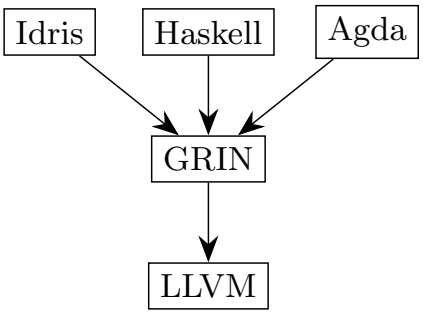


Fig. 4.1: Possible representations of different function languages

4.2 Heap points-to Analysis

Heap points-to analysis (HPT in the followings), or pointer analysis is a commonly used data-flow analysis in the context of imperative languages. The result of the analysis contains information about the possible variables or heap locations a given pointer can point to. In the context of GRIN, it is used to determine the type of data constructors (or nodes) a given variable could have been constructed with. The result is a mapping of variables and abstract heap locations to sets of data constructors.

The original version of the analysis presented in [4] and further detailed in [3] only supports node level granularity. This means, that the types of literals are not differentiated, they are unified under a common "basic value" type. Therefore, the analysis cannot be used for type inference as it is. In order to facilitate type inference, HPT has to be extended, so that it propagates type information about literals as well. This can be easily achieved by slightly adjusting the original version. Using the result of the modified algorithm, we can generate LLVM IR code from GRIN.

However, in some cases the monomorphic type inference algorithm presented above is not sufficient. For example, the Glasgow Haskell Compiler has polymorphic primitive operations. This means, that despite GRIN being a monomorphic language, certain compiler front ends can introduce external polymorphic functions to GRIN programs. To resolve this problem, we have to further extend the heap points-to analysis. The algorithm now needs a table of external functions with their respective type information. These functions *can* be polymorphic, hence they need special treatment during the analysis. When encountering external function applications, the algorithm has to determine the concrete type of the return value based on the possible types of the function arguments. Essentially, it has to fill all the type variables present in the type of the return value with concrete types. This can be achieved by unification. Fortunately, the unification algorithm can be expressed in terms of the same data-flow operations HPT already uses.

5 Dead Code Elimination

Dead code elimination is one of the most well-known compiler optimization techniques. The aim of dead code elimination is to remove certain parts of the program that neither affect its final result nor its side effects. This includes code that can never be executed, and also code which only consists of irrelevant operations on dead variables. Dead code elimination can reduce the size of the input program, as well as increase its execution speed. Furthermore, it can facilitate other optimizing transformation by restructuring the code.

5.1 Dead Code Elmination in GRIN

The original GRIN framework has three different type of dead code eliminating transformations. These are dead function elimination, dead variable elimination and dead function paramater elimination. In general, the effectiveness of most optimizations solely depends on the accuracy of the information it has about the program. The more precise information it has, the more agressive it can be. Furthermore, running the same transformation but with additional information available, can often yield more efficient code.

In the original framework, the dead code eliminating transformations were provided only a very rough approximation of the liveness of variables and function parameters. In fact, a variable was deemed dead only if it was never used in the program. As a consequence, the required analyses were really fast, but the transformations themselves were very limited.

5.2 Interprocedural Liveness Analysis

In order to improve the effectiveness of dead code elimination, we need more sophisticated data-flow analyses. Liveness analysis is a standard data-flow analysis that determines which variables are live in the program and which ones are not. It is important to note, that even if a variable is used in the program, it does not necessarily mean it is live. See Program code 5.1.

```
1 main =
2   n <- pure 5
3   y <- pure (CInt n)
4   pure 0
```

(a) Put into a data constructor

```
1 main =
2   n <- pure 5
3   foo n
4 foo x = pure 0
```

(b) Argument to a function call

Program code 5.1: Examples demonstrating that a used variable can still be dead

In the first example, we can see a program where the variable `n` is used, it is put into a `CInt` node, but despite this, it is obvious to see that `n` is still dead. Moreover, the liveness analysis can determine this fact just by examining the function body locally. It does not need to analyze any function calls. However, in the second example, we can see a very similar situation, but here `n` is an argument to a function call. To calculate the liveness of `n`, the analysis either has to assume that the arguments of `foo` are always live, or it has to analyze the body of the function. The former decision yields a faster, but less precise *intraprocedural* analysis, the latter results in a bit more costly, but also more accurate *interprocedural* analysis.

By extending the analysis with interprocedural elements, we can obtain quite a good estimate of the live variables in the program, while minimizing the cost of the algorithm. Using the information gathered by the liveness analysis, the original optimizations can remove even more dead code segments.

6 Dead Data Elimination

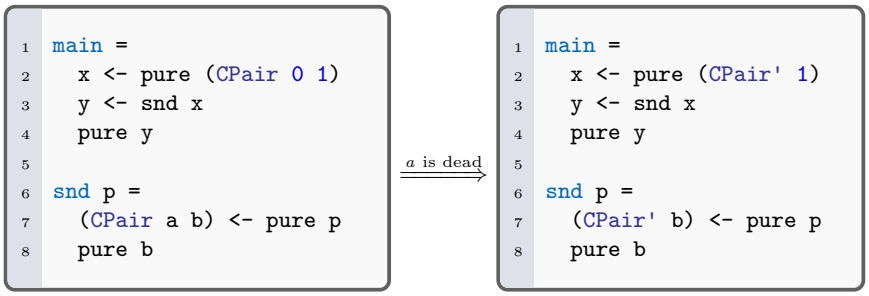
Conventional dead code eliminating optimizations usually only remove statements or expressions from programs; however, *dead data elimination* can transform the underlying data structures themselves. Essentially, it can specialize a certain data structure for a given use-site by removing or transforming unnecessary parts of it. It is a very powerful optimization technique that — given the right circumstances — can significantly decrease memory usage and reduce the number of heap operations.

Within the framework of GRIN, it was Remi Turk, who presented the initial version of dead data elimination in his master's thesis [23]. His original implementation used intraprocedural analyses and an untyped representation of GRIN. We extended the algorithm with interprocedural analyses, and improved the "dummification" process (see Sections 6.4 and 6.5). In the followings we present a high level overview of the original dead data elimination algorithm, as well as detail some of our modifications.

6.1 Dead Data Elimination in GRIN

In the context of GRIN, dead data elimination removes dead fields of data constructors (or nodes) for both definition- and use-sites. In the followings, we will refer to definition-sites as *producers* and to use-sites as *consumers*. Producers and consumers are in a *many-to-many* relationship with each other. A producer can define a variable used by many consumers, and a consumer can use a variable possibly defined by many producers. It only depends on the control-flow of the program. Program code 6.1 illustrates dead data elimination on a very simple example with a single producer and a single consumer.

As we can see, the first component of the pair is never used, so the optimization can safely eliminate the first field of the node. It is important to note, that the transformation has to remove the dead field for both the producer and the



(a) Before the transformation (b) After the transformation

Program code 6.1: A simple example for dead data elimination

consumer. Furthermore, the name of the node also has to be changed to preserve type correctness, since the transformation is specific to each producer-consumer group. This means, the data constructor `CPair` still exists, and it can be used by other parts of the program, but a new, specialized version is introduced for any optimizable producer-consumer group [1].

Dead data elimination requires a considerable amount of data-flow analyses and possibly multiple transformation passes. First of all, it has to identify potentially removable dead fields of a node. This information can be acquired by running liveness analysis on the program (see Section 5.2). After that, it has to connect producers with consumers by running the *created-by data-flow analysis*. Then it has to group producers together sharing at least one common consumer, and determine whether a given field for a given producer can be removed globally, or just dummified locally. Finally, it has to transform both the producers and the consumers.

6.2 Created-by Analysis

The created-by analysis, as its name suggests is responsible for determining the set of producers a given variable-was possibly created by. For our purposes, it is sufficient to track only node valued variables, since these are the only potential candidates for dead data elimination. Analysis example 6.1 demonstrates how the algorithm works on a simple program.

The result of the analysis is a mapping from variable names to set of producers grouped by their tags. For example, we could say that "variable `y` was created by the producer `a` given it was constructed with the `CTrue` tag". Naturally, a variable can be constructed with many different tags, and each tag can

[1] Strictly speaking, a new version is only introduced for each different set of live fields used by producer-consumer groups.

[2] For the sake of simplicity, we will assume that `xs` was constructed with the `CNil` and `CCons` tags. Also its producers are irrelevant in this example.

```
null xs =
  y <- case xs of
    (CNil) ->
      a <- pure (CTrue)
      pure a
    (CCons z zs) ->
      b <- pure (CFalse)
      pure b
  pure y
```

(a) Input program

Var	Producers
xs	$\{CNil[\dots], CCons[\dots]\}^2$
a	$\{CTrue[\mathsf{a}]\}$
b	$\{CFalse[\mathsf{b}]\}$
y	$\{CTrue[\mathsf{a}], CFalse[\mathsf{b}]\}$

(b) Anyalsis result

Analysis example 6.1: An example demonstrating the created-by analysis

have multiple producers. Also, it is important to note that some variables are their own producers. This is because producers are basically definitions-sites or bindings, identified by the name of the variable on their left-hand sides. However, not all bindings have variables on their left-hand side, and some values may not be bound to variables. Fortunately, this problem can be easily solved by a simple program transformation.

6.3 Grouping Producers

On a higher abstraction level, the result of the created-by analysis can be interpreted as a bipartite graph between producers and consumers. One group of nodes represents the producers and the other one represents the consumers. A producer is connected to a consumer if and only if the value created by the producer can be consumed by the consumer. Furthermore, each component of the graph corresponds to producer-consumer group. Each producer inside the group can only create values consumed by the consumers inside the same group, and a similar statement holds for the consumers as well.

6.4 Transforming Producers and Consumers

As mentioned earlier, the transformation applied by dead data elimination can be specific for each producer-consumer group, and both the producers and the consumers have to be transformed. Also, the transformation can not always simply remove the dead field of a producer. Take a look at Figure 6.1.

As we can see, producers P_1 and P_2 share a common consumer C_2. Let's assume, that the shared value is a `CPair` node with two fields, and neither C_1, nor C_2 uses the first field of that node. This means, the first field of the `CPair` node is locally dead for producer P_1. Also, suppose that C_3 does use the first field of that node, meaning it is live for P_2, hence it cannot be removed. In this

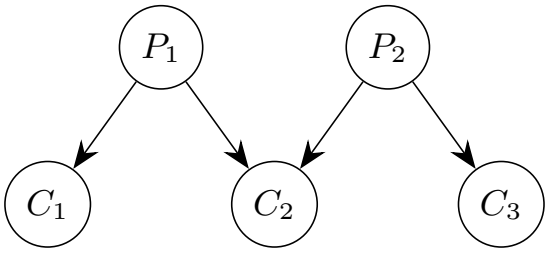


Fig. 6.1: Producer-consumer group

situation, if the transformation were to remove the locally dead field from P_1, then it would lead to a type mismatch at C_2, since C_2 would receive two `CPair` nodes with different number of arguments, with possibly different types for their first fields. In order to resolve this issue the transformation has to rename the tag at P_1 to `CPair'`, and create new patterns for `CPair'` at C_1 and C_2 by duplicating and renaming the existing ones for `CPair`. This way, we can avoid potential memory operations at the cost of code duplication.

In fact, even the code duplication can be circumvented by introducing the notion of *basic blocks* to the intermediate representation. This way, we still need to generate new alternatives (new patterns), but their right-hand sides will be simple jump instructions to the basic blocks of the original alternative's right-hand side.

6.5 The `undefined` value

Another option would be to only *dummify* the locally dead fields. In other words, instead of removing the field at the producer and restructuring the consumers, the transformation could simply introduce a dummy value for that field. The dummy value could be any placeholder with the same type as the locally dead field. For instance, it could be any literal of that type. A more sophisticated solution would be to introduce an undefined value. The `undefined` value is a placeholder as well, but it carries much more information. By marking certain values undefined instead of just introducing placeholder literals, we can facilitate other optimizations down the pipeline. However, each `undefined` value has to be explicitly type annotated for the heap points-to analysis to work correctly. Just like the other approach mentioned earlier, this alternative also solves the problem of code duplication at the cost of some modifications to the intermediate representation.

7 Idris Front End

Currently, our compiler uses the Idris compiler as its front end. The infrastructure can be divided into three components: the front end, that is responsible for generating GRIN IR from the Idris byte code; the optimizer, that applies

GRIN-to-GRIN transformations to the GRIN program, possibly improving its performance; and the back end, that compiles the optimized GRIN code into an executable.

7.1 Front end

The front end uses the bytecode produced by the Idris compiler to generate the GRIN intermediate representation. The Idris bytecode is generated without any optimizations by the Idris compiler. The code generation from Idris to GRIN is really simple, the difficult part of refining the original program is handled by the optimizer.

7.2 Optimizer

The optimization pipeline consists of three stages. In the first stage, the optimizer iteratively runs the so-called *regular optimizations*. These are the program transformations described in Urban Boquist's PhD thesis [4]. A given pipeline of these transformations are run until the code reaches a fixed-point, and cannot be optimized any further. This set of transformation are not formally proven to be confluent, so theoretically different pipelines can result in different fixed-points[1]. Furthermore, some of these transformations can work against each other, so a fixed-point may not always exist. In this case, the pipeline can be caught in a loop, where the program returns to the same state over and over again. Fortunately, these loops can be detected, and the transformation pipeline can be terminated.

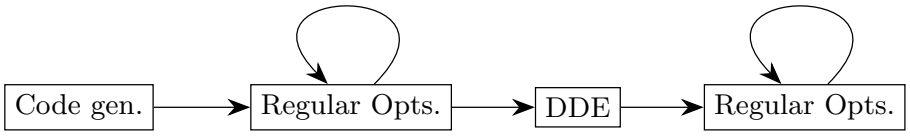


Fig. 7.1: The optimization stages

Following that, in the second stage, the optimizer runs the *dead data elimination pass*. This pass can be quite demanding on both the memory usage and the execution time due to the several data-flow analyses the transformation requires. Also, it is a rather specific transformation, which means, running it multiple times might not improve the code any further. As a consequence, the dead data elimination pass is executed only a single time during the entire optimization process. Since the dead data elimination pass can enable other optimizations, the optimizer runs the regular optimizations a second time right after the DDE pass.

[1] Although, experiments suggest that these transformations *are* confluent.

7.3 Back end

After the optimization process, the optimized GRIN code is passed onto the back end, which then generates an executable using the LLVM compiler framework. The input of the back end consists of the optimized GRIN code, the primitive operations of Idris and a minimal runtime (the latter two are both implemented in C). Currently, the runtime is only responsible for allocating heap memory for the program, and at this point it does not include a garbage collector.

The first task of the back end is to compile the GRIN code into LLVM IR code which is then optimized further by the LLVM Modular Optimizer [24]. After that, the optimized LLVM code is compiled into an object file by the LLVM Static Compiler [25]. Finally, Clang links together the object file with the C-implemented primitive operations and the runtime, and generates an executable binary.

8 Results

In this section, we present the initial results of our implementation of the GRIN framework. The measurements presented here can only be considered preliminary, given the compiler needs further work to be comparable to systems like the Glasgow Haskell Compiler or the Idris compiler [26]. Nevertheless, these statistics are still relevant, since they provide valuable information about the effectiveness of the optimizer.

8.1 Measured programs

The measurements were taken using the Idris front end and LLVM back end of the compiler. Each test program — besides "Length" — was adopted from the book *Type-driven development with Idris* [27] by Edwin Brady. These are small Idris programs demonstrating a certain aspect of the language.

"Length" is an Idris program, measuring the length of a list containing the natural numbers from 1 to 100. This example was mainly constructed to test how the dead data elimination pass can transform the inner structure of a list into a simple natural number (see Section 6).

8.2 Measured metrics

Each test program went trough the compilation pipeline described in Section 7, and measurements were taken at certain points during the compilation. The programs were subject to three different types of measurements.

- Static, compile time measurements of the GRIN code.
- Dynamic, runtime measurements of the interpreted GRIN code.
- Dynamic, runtime measurements of the executed binaries.

The compile time measurements were taken during the GRIN optimization passes, after each transformation. The measured metrics were the number of `store`s, `fetch`es and function definitions. These measurements ought to illustrate how the GRIN code becomes more and more efficient during the optimization process. The corresponding diagrams for the static measurements are Diagrams 8.1b to 8.4b. On the horizontal axis, we can see the indices of the transformations in the pipeline, and on the vertical axis, we can see the number of the corresponding syntax tree nodes. Reading these diagram from left to right, we can observe the continuous evolution of the GRIN program throughout the optimization process.

The runtime measurements of the interpreted GRIN programs were taken three points during the compilation process. First, right after the GRIN code is generated from the Idris byte code; second, after the regular optimization passes; and finally, at the end of the entire optimization pipeline. As can be seen on Figure 7.1, the regular optimizations are run a second time right after the dead data elimination pass. This is because the DDE pass can enable further optimizations. To clarify, the third runtime measurement of the interpreted GRIN program was taken after the second set of regular optimizations. The measured metrics were the number of executed function calls, case pattern matches, `store`s and `fetch`es. The goal of these measurements is to compare the GRIN programs at the beginning and at the end of the optimization pipeline, as well as to evaluate the efficiency of the dead data elimination pass. The corresponding diagrams for these measurement are Diagrams 8.1a to 8.4a.

The runtime measurements of the binaries were taken at the exact same points as the runtime measurements of the interpreted GRIN code. Their goal is similar as well, however they ought to compare the generated binaries instead of the GRIN programs. The measured metrics were the size of the binary, the number of executed user-space instructions, stores and loads. The binaries were generated by the LLVM back end described in Section 7.3 with varying opitmization levels for the LLVM Optimizer. The optimization levels are indicated in the corresponding tables: Tables 8.1 to 8.4. Where the optimization level is not specified, the default, `O0` level was used. As for the LLVM Static Compiler and Clang, the most aggressive, `O3` level was set.

8.3 Measurement setup

All the measurements were performed on a machine with `Intel(R) Core(TM) i7-4710HQ CPU @ 2.50GHz` processor and `Ubuntu 18.04 bionic` operating system with `4.15.0-46-generic` kernel. Idris compiler used by the front-end is of version 1.3.1.

The actual commands for the binary generation are detailed in Program code 8.1. That script has two parameters: `N` and `llvm-in`. `N` is the optimization level for the LLVM Optimizer, and `llvm-in` is the LLVM program generated from the optimized GRIN code.

```
opt-7 -ON <llvm-in> -o <llvm-out>
llc-7 -O3 -relocation-model=pic -filetype=obj -o <object-file>
clang-7 -O3 prim_ops.c runtime.c <object-file> -s -o <executable>
```

Program code 8.1: Commands for binary generation

As for the runtime measurements of the binary, we used the `perf` tool. The used command can be seen in Program code 8.2.

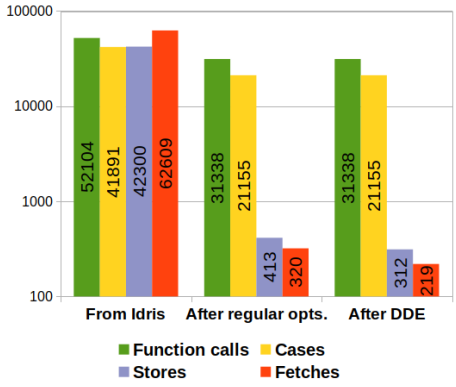
```
perf stat -e cpu/mem-stores/u -e "r81d0:u" -e instructions:u
↪  <executable>
```

Program code 8.2: Command for runtime measurements of the binary

8.4 Length

Diagram 8.1: Length - GRIN statistics

(a) Runtime



(b) Compile time

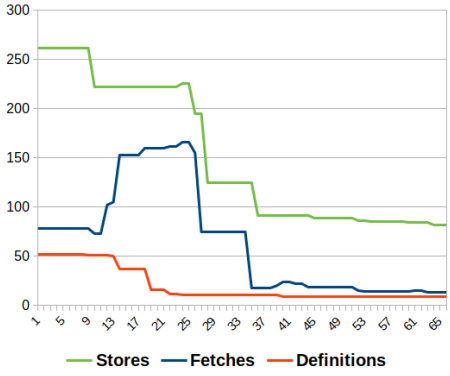


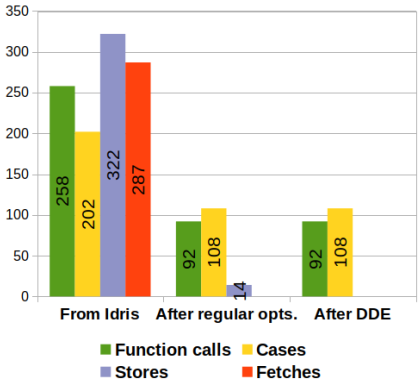
Table 8.1: Length - binary statistics

Stage	Size	Instructions	Stores	Loads
`normal-00`	23928	769588	212567	233305
`normal-03`	23928	550065	160252	170202
`regular-opt`	19832	257397	14848	45499
`dde-00`	15736	256062	14243	45083
`dde-03`	15736	284970	33929	54555

8.5 Exact length

Diagram 8.2: Exact length - GRIN statistics

(a) Runtime



(b) Compile time

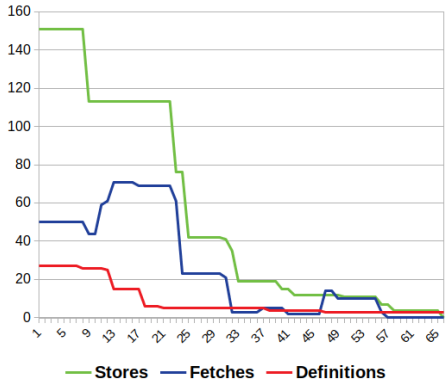


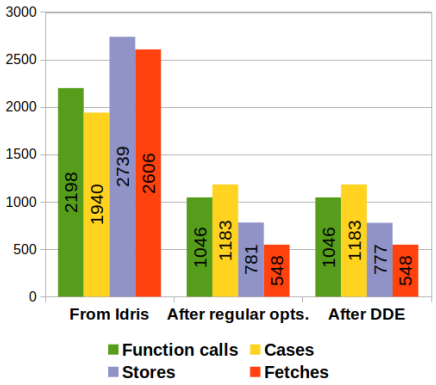
Table 8.2: Exact length - binary statistics

Stage	Size	Instructions	Stores	Loads
`normal-00`	18800	188469	14852	46566
`normal-03`	14704	187380	14621	46233
`regular-opt`	10608	183560	13462	45214
`dde-00`	10608	183413	13431	45189
`dde-03`	10608	183322	13430	44226

8.6 Type level functions

Diagram 8.3: Type level functions - GRIN statistics

(a) Runtime



(b) Compile time

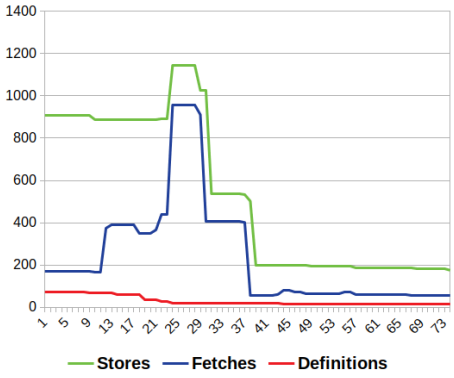


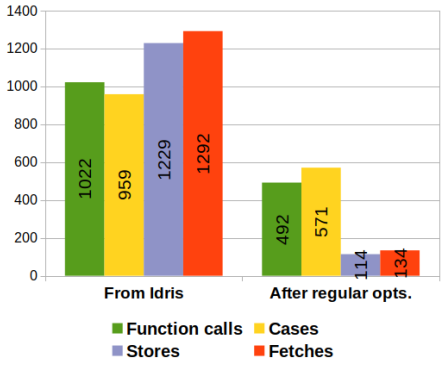
Table 8.3: Type level functions - binary statistics

Stage	Size	Instructions	Stores	Loads
`normal-O0`	65128	383012	49191	86754
`normal-O3`	69224	377165	47556	84156
`regular-opt`	36456	312122	34340	71162
`dde-O0`	32360	312075	34331	70530
`dde-O3`	28264	309822	33943	70386

8.7 Reverse

Diagram 8.4: Reverse - GRIN statistics

(a) Runtime



(b) Compile time

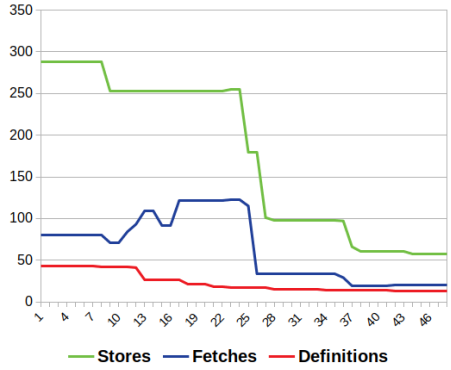


Table 8.4: Reverse - binary statistics

Stage	Size	Instructions	Stores	Loads
`normal-O0`	27112	240983	25018	58253
`normal-O3`	31208	236570	23808	56617
`regular-opt-O0`	14824	222085	19757	53125
`regular-opt-O3`	14824	220837	19599	52827

Acknowledgements

The project has been supported by the European Union, co-financed by the European Social Fund (EFOP-3.6.3-VEKOP-16-2017-00002).

References

1. Sulzmann, Martin and Chakravarty, Manuel MT and Jones, Simon Peyton and Donnelly, Kevin, "System F with type equality coercions," in *Proceedings of the 2007 ACM SIGPLAN international workshop on Types in languages design and implementation*. ACM, 2007, pp. 53–66.
2. Hall, Cordelia V. and Hammond, Kevin and Partain, Will and Peyton Jones, Simon L. and Wadler, Philip, "The Glasgow Haskell Compiler: A Retrospective," in *Proceedings of the 1992 Glasgow Workshop on Functional Programming*. London, UK: Springer-Verlag, 1993, pp. 62–71. [Online]. Available: http://dl.acm.org/citation.cfm?id=647557.729914

3. U. Boquist and T. Johnsson, "The GRIN Project: A Highly Optimising Back End for Lazy Functional Languages," in *Selected Papers from the 8th International Workshop on Implementation of Functional Languages*, ser. IFL '96. Berlin, Heidelberg: Springer-Verlag, 1997, pp. 58–84. [Online]. Available: http://dl.acm.org/citation.cfm?id=647975.743083
4. U. Boquist, "Code Optimisation Techniques for Lazy Functional Languages," Ph.D. dissertation, Chalmers University of Technology and Göteborg University, 1999.
5. Augustsson, Lennart, "Haskell B. user manual," *Programming methodology group report, Dept. of Comp. Sci, Chalmers Univ. of Technology, Göteborg, Sweden*, 1992.
6. Peyton Jones, Simon, *The Implementation of Functional Programming Languages*. Prentice Hall, January 1987, pages 185–219. [Online]. Available: https://www.microsoft.com/en-us/research/publication/the-implementation-of-functional-programming-languages/
7. Andersen, Lars Ole, "Program analysis and specialization for the C programming language," Ph.D. dissertation, University of Cophenhagen, 1994.
8. Dijkstra, Atze and Fokker, Jeroen and Swierstra, S. Doaitse, "The Architecture of the Utrecht Haskell Compiler," in *Proceedings of the 2Nd ACM SIGPLAN Symposium on Haskell*, ser. Haskell '09. New York, NY, USA: ACM, 2009, pp. 93–104. [Online]. Available: http://doi.acm.org/10.1145/1596638.1596650
9. Douma, Christof, "Exceptional GRIN," Ph.D. dissertation, Master's thesis, Utrecht University, Institute of Information and Computing, 2006.
10. C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation," in *CGO*, San Jose, CA, USA, Mar 2004, pp. 75–88.
11. John Meacham, "JHC." [Online]. Available: http://repetae.net/computer/jhc/jhc.shtml
12. Berardi, Stefano, "Towards a mathematical analysis of the Coquand-Huet calculus of constructions and the other systems in Barendregt's cube," *Technica1 report, Carnegie-Me11on University (USA) and Universita di Torino (Ita1y)*, 1988.
13. Terlouw, Jan, "Een nadere bewijstheoretische analyse van GSTT's," *Manuscript (in Dutch)*, 1989.
14. Barendregt, Henk P, "Lambda calculi with types," 1992.
15. Okabe, Kiwamu and Muranushi, Takayuki, "Systems Demonstration: Writing NetBSD Sound Drivers in Haskell," *SIGPLAN Not.*, vol. 49, no. 12, pp. 77–78, Sep. 2014. [Online]. Available: http://doi.acm.org/10.1145/2775050.2633370
16. David Himmelstrup, "LLVM Haskell Compiler." [Online]. Available: http://lhc-compiler.blogspot.com/
17. Liu, Hai and Glew, Neal and Petersen, Leaf and Anderson, Todd A., "The Intel Labs Haskell Research Compiler," *SIGPLAN Not.*, vol. 48, no. 12, pp. 105–116, Sep. 2013. [Online]. Available: http://doi.acm.org/10.1145/2578854.2503779
18. Petersen, Leaf and Orchard, Dominic and Glew, Neal, "Automatic SIMD Vectorization for Haskell," *SIGPLAN Not.*, vol. 48, no. 9, pp. 25–36, Sep. 2013. [Online]. Available: http://doi.acm.org/10.1145/2544174.2500605
19. L. Petersen, T. A. Anderson, H. Liu, and N. Glew, "Measuring the Haskell Gap," in *Proceedings of the 25th Symposium on Implementation and Application of Functional Languages*, ser. IFL '13. New York, NY, USA: ACM, 2014, pp. 61:61–61:72. [Online]. Available: http://doi.acm.org/10.1145/2620678.2620685
20. S. Weeks, "Whole-program Compilation in MLton," in *Proceedings of the 2006 Workshop on ML*, ser. ML '06. New York, NY, USA: ACM, 2006, pp. 1–1. [Online]. Available: http://doi.acm.org/10.1145/1159876.1159877

21. M. Fluet and S. Weeks, "Contification Using Dominators," *SIGPLAN Not.*, vol. 36, no. 10, pp. 2–13, Oct. 2001. [Online]. Available: http://doi.acm.org/10.1145/507669.507639
22. B. A. Leibig, "An LLVM Back-end for MLton," Department of Computer Science, B. Thomas Golisano College of Computing and Information Sciences, Tech. Rep., 2013, a Project Report Submitted in Partial Fulfillment of the Requirements for the Degree of Master of Science in Computer Science. [Online]. Available: https://www.cs.rit.edu/~mtf/student-resources/20124_leibig_msproject.pdf
23. R. Turk, "A modern back-end for a dependently typed language," Master's thesis, Universiteit van Amsterdam, 2010.
24. "Modular LLVM Analyzer and Optimizer." [Online]. Available: http://llvm.org/docs/CommandGuide/opt.html
25. "LLVM Static Compiler." [Online]. Available: https://llvm.org/docs/CommandGuide/llc.html
26. Brady, Edwin, "Idris, a general-purpose dependently typed programming language: Design and implementation," *Journal of Functional Programming*, vol. 23, no. 5, p. 552–593, 2013.
27. ——, *Type-driven development with Idris*. Manning Publications Company, 2017.