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 [**?**], some even supporting dependent types. Although these features make writing code more convenient, they also complicate the compilation process.

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

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 consists 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 individual functions only 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 suitable front end language.

In this paper we present a modern look at the GRIN framework. We explain some of its core concepts, and also provide a number of improvements to it. The results are demonstrated through a modernized implementation of the framework[1]. The main contributions presented in the paper are the following.

1. Extension of the heap points-to analysis with more accurate basic value tracking
2. Specification of a type inference algorithm for GRIN using the extended heap points-to analysis
3. Implementation of an LLVM back end for the GRIN framework
4. Extension of the dead data elimination transformation with typed dummification and an overview of an alternative transformation for producer-consumer groups
5. Implementation of an Idris front end for the GRIN framework

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.

2.1 General overview

Interprocedural program analysis is a type of data-flow analysis that propagates information about certain program elements through function calls. Using in-

[1] Almost the entire GRIN framework has been reimplemented. The only exceptions are the simplfyifing transformations which are no longer needed by the new code generator that uses LLVM as its target language.

terprocedural 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. Furthermore, 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, many 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 such as closures, higher-order functions and even laziness. 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.

2.2 A small example

As a brief introduction to the GRIN language, we will show how a small functional program can be encoded in GRIN. We will use the following example program: `(add 1) (add 2 3)`. The `add` function simply takes two integers, and adds them together. This means, that the program only makes sense in a language that supports partial function application, due to `add` being applied only to a single argument. We will also assume, that the language has lazy semantics. We can see the GRIN code generated from the above program in Program code 2.1.

The first thing we can notice is that GRIN has a monadic structure, and syntactically it is very similar to low-level Haskell. The second one, is that it has data constructors (`CInt`, `Fadd`, etc). We will refer to them as *nodes*. Thirdly, we can see four function definitions: `grinMain`, the main entry point of our program; `add`, the function adding two integers together; and two other functions called `eval` and `apply`. Lastly, we can see `_prim_int_add` and the `store`, `fetch` and `update` operations, which do not have definitions. The first one is a primitive operation, and the last three are intrinsic operations responsible for graph reduction. We can also view `store`, `fetch` and `update` as simple heap operations: `store` puts values onto the heap, `fetch` reads values from the heap, and `update` modifies values on the heap.

```
grinMain =
  a <- store (CInt 1)
  b <- store (CInt 2)
  c <- store (CInt 3)

  r <- store (Fadd b c)
  suc <- pure (P1_add a)
  apply suc r

add x y =
  (CInt x1) <- eval x
  (CInt y1) <- eval y
  r <- _prim_int_add x1 y1
  pure (CInt r)
```

```
eval p =
  v <- fetch p
  case v of
    (CInt _n)   -> pure v
    (P2_add)    -> pure v
    (P1_add _x) -> pure v
    (Fadd x2 y2) ->
      r_add <- add x2 y2
      update p r_add
      pure r_add

apply f u =
  case f of
    (P2_add) ->
      pure (P1_add u)
    (P1_add z) -> add z u
```

Program code 2.1: GRIN code generated from `(add 1) (add 2 3)`

The GRIN program is always a first order, strict, defunctionalized version of the original program, where laziness and partial application are expressed explicitly by `eval` and `apply`. A lazy function call can be expressed by wrapping its arguments into an `F` node. As can be seen, the `add 2 3` expression is compiled into the `Fadd 2 3` node. Whenever a lazy value needs to be evaluated, the GRIN program will call the `eval` function, which will force the given computation and update the stored value (so that it is not computed twice), or it will just return the value if it is already in weak head normal form. For a partial function call, the GRIN program will construct a `P` node, and call the `apply` function. The number in the `P` node's tag indicates how many arguments are still missing to the given function call. The `apply` function will take a partially applied function (a `P` node), and will apply it to a given argument. The result can be either another partially applied function, or the result of a saturated function call.

The definitions of `eval` and `apply` are uniquely generated for each program by the GRIN back end. As we can see, they are just ordinary GRIN functions, which means the compiler can analyze and optimize them. For a more detailed description, the reader can refer to [**?**, **?**].

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 [**?**] 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 using the syntax of the language. This means, in order to generate optimal machine code, the code generator cannot use only the AST of the program, but also has to rely on the previously calculated strictness analysis result. This makes the code generation phase more complicated. 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 Clean compiler

The Clean compiler [**?**] is also an industrial-grade compiler, supporting concurrency and a multitude of platforms. It uses the abstract ABC machine as it's evaluation model. The ABC machine is a stack machine which uses three different stacks: the Argument stack, the Basic value stack and the Code stack. The Clean compiler performs no optimizations on the ABC machine level, since defining code transformations on a stack-based representation would be quite inconvenient. Instead, the driving design principle behind the ABC machine is that it should be easy to generate native machine code from it. In the present days, this task is often accomplished by LLVM, which not only guarantees performance, but also provides a higher level intermediate representation. Nonetheless, the Clean compiler generates performant code for most major platforms.

The main difference between Clean and Haskell lies in the type systems. Clean uses uniqueness typing, a concept similar to linear typing. A function argument can be marked unique, which means that it will be used only a single time in the function definition. This allows the compiler to generate destructive updates on that argument after it has been used. The efficiency of Clean programs is largely not attributed to code optimizations, but rather to the fact that the programmer writes mutable code to begin with. Uniqueness typing introduces controlled mutability which can highly increase the efficiency of Clean programs.

3.3 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.

[1] Strict semantics can be expressed as well.

Boquist The original GRIN framework was developed by U. Boquist, and first described in the article [**?**], then in his PhD thesis [**?**]. This version of GRIN used the Chalmers Haskell-B Compiler [**?**] as its front end and RISC as its back end. The main focus of the 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 semantics [**?**]. 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 [**?**].

UHC The Utrecht Haskell Compiler [**?**] 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 [**?**]. Also, UHC can generate code for many different targets including LLVM [**?**], .Net, JVM and JavaScript.

JHC JHC [**?**] 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 [**?**, **?**]. In theory, the pure type system can be seen as a generalization of the lambda cube [**?**], 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, which for instance was used to implement a NetBSD sound driver in high-level Haskell [**?**].

LHC The LLVM Haskell Compiler [**?**] is a 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.

3.4 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.

MLton MLton [**?**] 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 [**?**], 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 [**?**].

Intel Research Compiler The Intel Labs Haskell Research Compiler [**?**] was a result of a long running research project of Intel 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 [**?**]. 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 [**?**].

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 separation. 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

[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.

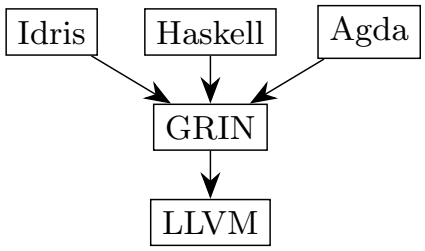


Fig. 4.1: Possible representations of different functional languages

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.

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 [**?**] and further detailed in [**?**] 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 defining primitive types for the literal values. 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[1]. Es-

[1]This concrete type always exists, since all inputs to the program have concrete types (which are propagated through the program), and we know the entire program at compile time.

sentially, 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.

4.3 Type Information from the Surface Language

Another option would be to use type information provided by the surface language. This approach might seem convenient, but it has three major disadvantages. The first one is that this solution would need to address each front end language separately, since they might have different type systems. Secondly, requiring type information from the front end would rule out dynamically typed languages. Lastly, the surface language's type system tells us about the *semantics* of the program, however we need information about the *data representation* to efficiently analyze, optimize, and generate machine code from GRIN programs. The two concepts might seem familiar at first, but the type-based control flow analysis yields a lot less precise result than the heap-points-to analysis (slightly modified 0-CFA) [**?**].

In object oriented languages, type-based control flow analysis is sometimes used to make the general pointer analysis more precise. In certain cases, type information can help to filter out impossible cases calculated by the pointer analysis (e.g.: when using interfaces). For functional languages, this approach only works for strict data structures. For example, if we have a strict list, we know that it has been constructed with either `Nil` or `Cons`. However, if the list is lazy, it still might be a thunk referring to any function that returns a list. This means, that in the defunctionalized GRIN program, the list can not only have a `CNil` or a `CCons` tag, but also any `F` tag belonging to a function that returns a list. Consequently, the set of possible tags for a given lazy type would have to include all those `F` tags as well. This would hinder the type-based analysis considerably inaccurate.

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.

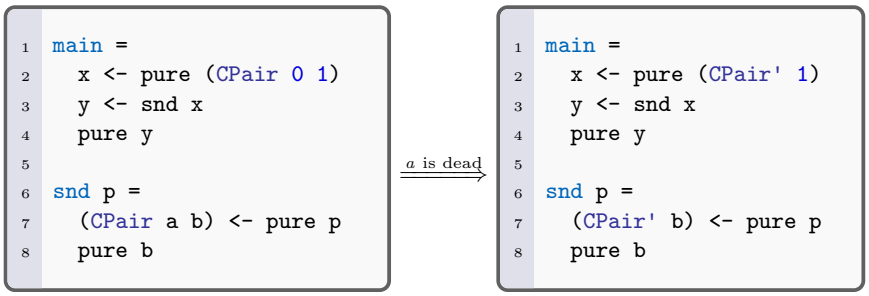
6 Dead Data Elimination

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 powerful optimization technique that — given the right circumstances — can significantly decrease memory usage and reduce the number of executed 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 [**?**]. 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 following 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.



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

Program code 6.1: A simple example for dead data elimination

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 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.

```
1 null xs =
2   y <- case xs of
3     (CNil) ->
4       a <- pure (CTrue)
5       pure a
6     (CCons z zs) ->
7       b <- pure (CFalse)
8       pure b
9   pure y
```

(a) Input program

Var	Producers
xs	$\{CNil[\ldots], CCons[\ldots]\}^1$
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

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

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

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 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 directed 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 one 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.

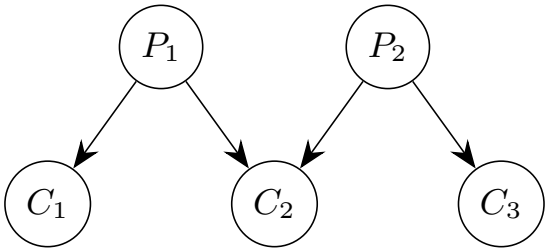


Fig. 6.1: Producer-consumer group

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

[1] 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.

field of that node, meaning it is live for P_2, hence it cannot be removed. In this 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. Basic blocks allow us to transfer control between different code segments meanwhile maintaining the same local environment (local variables). This means, we can share code between the different alternatives of a case expression. 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. Previously we needed structural extensions facilitating code sharing (basic blocks), now we had to introduce a new basic value (typed `undefined`).

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

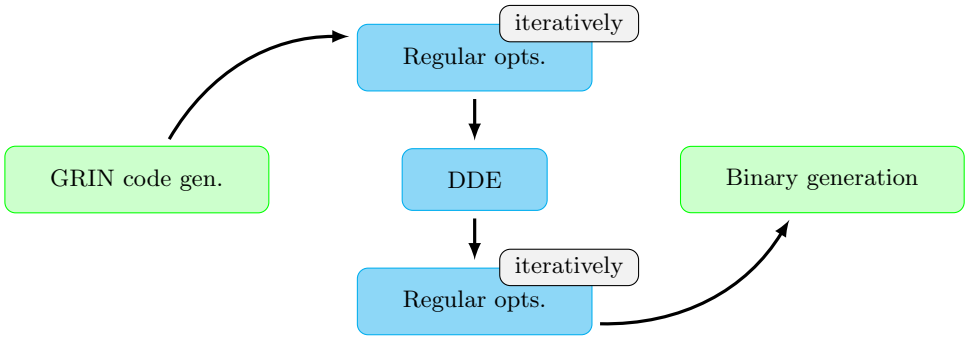


Fig. 7.1: Idris compilation pipeline

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, as can be seen in Figure 7.1. 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 [**?**]. 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.

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 it requires, and the unrefined implementation. 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. This also means, that the liveness analysis could collect more precise information about certain variables, which implies that another pass of DDE could optimize the GRIN program even further. However, in order to run the DDE pass multiple times its implementation has to be improved (see section 9).

[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 [**?**]. Currently, the back end uses the default LLVM optimization pipeline. After that, the optimized LLVM code is compiled into an object file by the LLVM Static Compiler [**?**]. 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 [**?**]. 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* [**?**] by Edwin Brady. These are small Idris programs demonstrating a certain aspect of the language.

"Length" is an Idris program, calculating 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 through 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 at 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, loads, total heap memory usage (in bytes) and execution speed (in milliseconds)[1]. The binaries were generated by the LLVM back end described in Section 7.3 with varying optimization 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 for all the measurements.

There are also measurements for the binaries generated by the Idris compiler. These were compiled using the highest (`O3`) optimization level and the C back end. For these executables, the size is not included, because Idris compiles a full-fledged runtime system into the binary. Since our Idris back end only has a mnimal runtime yet, the sizes of the binaries are not comparable. However, all other metrics are, because during these measurements, Idris' garbage collector was never triggered. This can be accomplished by configuring the initial size of the heap memory through the runtime system of Idris. This allows us to compare Idris and GRIN binaries despite the *yet* non-implemented garbage collector for GRIN.

[1]The execution speed was measured by averaging the result of 1000 measurements.

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. The Idris compiler used by the front-end is of version 1.3.1, and the LLVM used by the back end is of version 7.

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 runtime of Idris and the minimal runtime of GRIN. The `perf` command can be seen in Program code 8.2 which was used to count the number of executed user space instructions, stores, loads and to measure the execution speeds. The runtimes were used to determine the memory usage, and to make sure that Idris' garbage collector is never triggered.

```
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

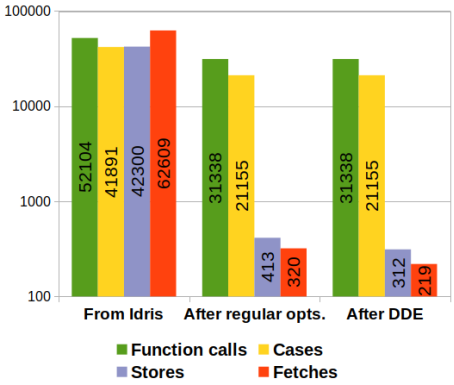
The first thing we can notice on the runtime statistics of the GRIN code, is that the GRIN optimizer significantly reduced the number of heap operations, as well as the number of function calls and case pattern matches. Moreover, the DDE pass could further improve the program's performance by removing additional heap operations.

The compile time statistics demonstrate an interesting phenomena. The number of `store`s and function definitions continuously keep decreasing, but at a certain point, the number of `fetch`es suddenly increase by a relatively huge margin. This is due to the fact that the optimizer usually performs some preliminary transformations on the GRIN program *before* inlining function definitions. This explains the sudden rise in the number of `fetch`es during the early stages

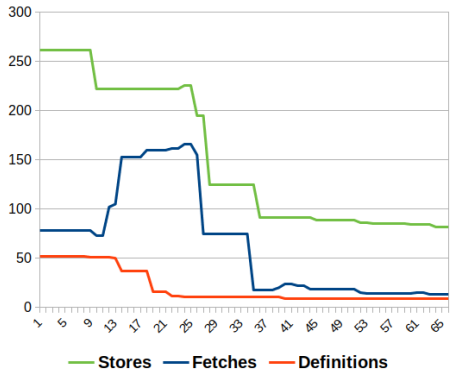
of the optimization process. Following that spike, the number of heap operations and function definitions gradually decrease until the program cannot be optimized any further.

Diagram 8.1: Length - GRIN statistics

(a) Runtime



(b) Compile time



The runtime statistics for the executed binary are particularly interesting. First, observing the `O0` statistics, we can see that the regular optimizations substantially reduced the number of executed instructions and memory operations, just as we saw with the interpreted GRIN code. Also, it is interesting to see that the DDE optimized binary did not perform any better than the regularly optimized one; however, its size decreased by more than 20%.

We can also notice the huge memory usage difference between the Idris program and the GRIN programs that were only optimized by LLVM but not by GRIN. This because the rather simple code generation scheme of the Idris front end as discussed in 7.1. However, after running the optimizations, the optimized GRIN programs consume considerably less memory, and have better execution times as well.

It is worth noting that the Idris binary executed significantly more instructions, and performed a lot more stores and loads than the unoptimized GRIN binary, yet it had a better execution time. The excessive number of memeory operations can be explained by Idris' calling convention. The function arguments are always psuhed onto the stack by the caller, and popped by the callee. This results in a lot of stack memory stores and loads which are reflected in the measurements. However, since the stack memory operations are quite fast, they have no significant impact on the execution times.

As for the high number of executed instructions, we can only hypothesize that it's caused by the Idris runtime system. Idris uses the runtime system to allocate memory through multiple function calls. In GRIN, the memory operations are

kind of "inlined" into the generated LLVM code. This might mean that the binaries generated by the Idris compiler could execute a lot more instructions for every memory operation.

Table 8.1: Length - CPU binary statistics

Stage	Size	Instructions	Stores	Loads	Memory	Time
`idris`	-	2822725	366880	1064977	9440	0.838
`normal-O0`	23928	769588	212567	233305	674080	1.993
`normal-O3`	23928	550065	160252	170202	674080	1.056
`regular-opt`	19832	257397	14848	45499	8200	0.463
`dde-O0`	15736	256062	14243	45083	5776	0.525
`dde-O3`	15736	284970	33929	54555	5776	0.461

Also, it should be pointed out that the aggressively optimized DDE binary performed much worse than the `O0` version. This is because the default optimization pipeline of LLVM is designed for the C and C++ languages. As a consequence, in certain scenarios it may perform poorly for other languages. In the future, we plan to construct a better LLVM optimization pipeline for GRIN.

8.5 Exact length

For the GRIN statistics of "Exact length", we can draw very similar conclusions as for "Length". However, closely observing the statistics, we can see, that the DDE pass completely eliminated *all* heap operations from the program. In principle, this means, that all the variables can be put into registers during the execution of the program. In practice, some variables will be spilled onto stack, but the heap will never be used.

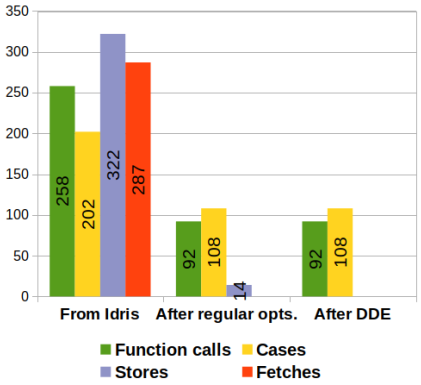
The binary statistics show that the optimized GRIN programs really do not use any heap memory. As for the other measured metrics, we do not see any major improvements.

Table 8.2: Exact length - CPU binary statistics

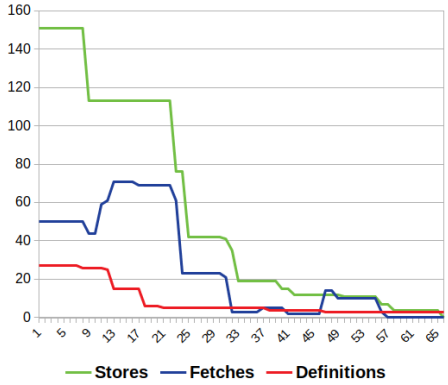
Stage	Size	Instructions	Stores	Loads	Memory	Time
`idris`	-	260393	23320	68334	1888	0.516
`normal-O0`	18800	188469	14852	46566	4112	0.464
`normal-O3`	14704	187380	14621	46233	4112	0.455
`regular-opt`	10608	183560	13462	45214	112	0.451
`dde-O0`	10608	183413	13431	45189	0	0.453
`dde-O3`	10608	183322	13430	44226	0	0.448

Diagram 8.2: Exact length - GRIN statistics

(a) Runtime



(b) Compile time

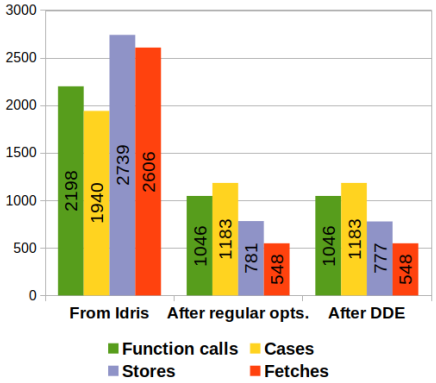


8.6 Type level functions

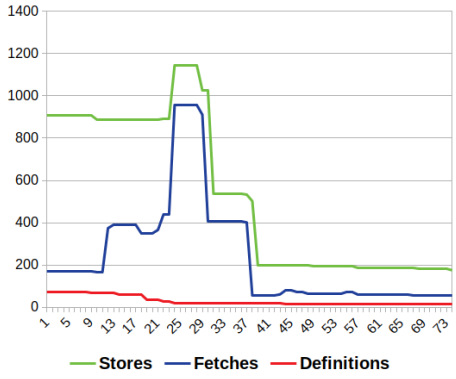
The GRIN statistics for this program may not be particularly interesting, but they demonstrate that the GRIN optimizations work for programs with many type level computations as well.

Diagram 8.3: Type level functions - GRIN statistics

(a) Runtime



(b) Compile time



The binary statistics look promising for “Type level functions”. Almost all measured performance metrics are strictly decreasing, which suggests that even the default LLVM optimization pipeline can work for GRIN. Also, the optimized GRIN programs use almost half as much memory as the Idris program.

Table 8.3: Type level functions - CPU binary statistics

Stage	Size	Instructions	Stores	Loads	Memory	Time
`idris`	-	525596	70841	158363	29816	0.637
`normal-O0`	65128	383012	49191	86754	44212	0.581
`normal-O3`	69224	377165	47556	84156	44212	0.536
`regular-opt`	36456	312122	34340	71162	15412	0.516
`dde-O0`	32360	312075	34331	70530	15236	0.532
`dde-O3`	28264	309822	33943	70386	15236	0.513

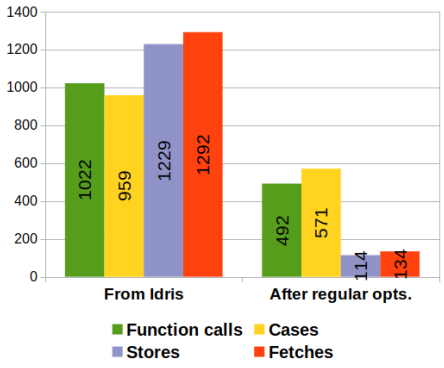
8.7 Reverse

Unlike, the previous programs, “Reverse” could not have been optimized by the dead data elimination pass. The pass had no effect on it. Fortunately, the regular optimizations alone could considerably improve both the runtime and compile time metrics of the GRIN code.

The binary statistics are rather promising. The binary size decreased by a substantial margin and the number of executed memory operations has also been reduced by quite a lot. Furthermore, the optimized GRIN programs use less than one third of the memory that the Idris program uses.

Diagram 8.4: Reverse - GRIN statistics

(a) Runtime



(b) Compile time

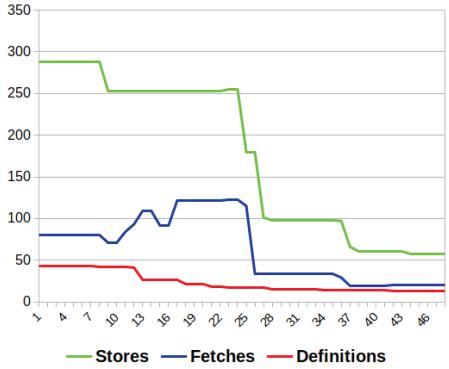


Table 8.4: Reverse - CPU binary statistics

Stage	Size	Instructions	Stores	Loads	Memory	Speed
`idris`	-	350215	37893	101040	7656	0.576
`normal-O0`	27112	240983	25018	58253	18640	0.498
`normal-O3`	31208	236570	23808	56617	18640	0.481
`regular-opt-O0`	14824	222085	19757	53125	2384	0.467
`regular-opt-O3`	14824	220837	19599	52827	2384	0.454

8.8 General conclusions

In general, the measurements demonstrate that the GRIN optimizer can considerably improve the performance metrics of a given GRIN program. The regular optimizations themselves can usually produce highly efficient programs, however, in certain cases the dead data elimination pass can facilitate additional optimizations, and can further improve the performance.

The results of the binary measurements indicate that the GRIN optimizer performs optimizations orthogonal to the LLVM optimizations. This supports the motivation behind the framework, which is to transform functional programs into a more manageable format for LLVM by eliminating the functional artifacts. This is backed up by the fact, that none of the fully optimized `normal` programs could perform as well as the regularly or DDE optimized ones. Also, it is interesting to see, that there is not much difference between the `O0` and `O3` default LLVM optimization pipelines for GRIN. This motivates further research to find an optimal pipeline for GRIN.

Finally, it is rather surprising to see, that the dead data elimination pass did not really impact the performance metrics of the executed binaries, but it significantly reduced their size. Firstly, it might be unorthodox to expect speedup from dead code elimination; however, dead data elimination does not only remove unused code, but it transforms the underlying data representations that the program uses. For instance, it could reduce the size of nodes such that they fit into fewer registers, which could help the register allocator, and thus improve the performance of the program. Also, it could remove the elements of a list, leaving only its spine, thus reducing the initial number of heap operations required to allocate the list. Finally, it could help the garbage collector by not allocating unused heap objects as well as reducing the size of the memory map it has to traverse.

Not seeing any performance gains can be explained by the fact, that most of these programs are quite simple, and do not contain any compound data structures. Dead data elimination can shine when a data structure is used in a specific way, so that it can be locally restructured for each use site. However, when applying it to simple programs, we can obtain sub par results.

Nevertheless, the binary size reduction is still notable, and demonstrates that even for simple programs, dead data elimination can still have a significant impact.

9 Future Work

Currently, the framework only supports the compilation of Idris, but we are working on supporting Haskell by integrating the Glasgow Haskell Compiler as a new front end. As of right now, the framework *can* generate GRIN IR code from GHC's STG representation, but the generated programs still contain unimplemented primitive operations. The main challenge is to somehow handle these primitive operations. In fact, there is only a small set of primitive operations that cannot be trivially incorporated into the framework, but these might even require extending the GRIN IR with additional built-in instructions.

Besides the addition of built-in instructions, the GRIN intermediate representation can be improved further by introducing the notion of function pointers and basic blocks. Firstly, the original specification of GRIN does not support modular compilation. However, extending the IR with function pointers can help to achieve incremental compilation. Each module could be compiled separately with indirect calls to other modules through function pointers, then by using different data-flow analyses and program transformations, all modules could be optimized together incrementally. In theory, if the entire program is available for analysis at compile time, incremental compilation could produce the same result as whole program compilation. In practice, the LLVM compiler already uses link-time optimizations which implement a very similar idea.

Secondly, the original GRIN IR has a monadic structure which can make it difficult to analyze and transform the control flow of the program. In certain cases it would be beneficial to explicitly transfer control from one program point to another. There two main use cases for this: code sharing (see section 6.4) and explicit tail recursion. Fortunately, replacing the monadic structure of GRIN with basic blocks can resolve both of these issues.

Whole program analysis is a powerful tool for optimizing compilers, but it can be quite demanding on execution time. This being said, there are certain techniques to speed up these analyses. The core of the GRIN optimizer is the heap points-to analysis, an Andersen-style inclusion based pointer analysis [**?**]. This type of data-flow analysis is very well researched, and there are several ways to improve the algorithm's performance. Firstly, cyclic references could be detected and eliminated between data-flow nodes at runtime. This optimization allows the algorithm to analyze millions of lines of code within seconds [**?**]. Secondly, the algorithm itself could be parallelized for both CPU and GPU [**?**], achieving considerable speedups. Furthermore, some alternative algorithms could also be considered. For example, Steengaard's unification based algorithm [**?**] is a less precise analysis, but it runs in almost linear time. It could be used as a preliminary analysis for some simple transformations at the beginning of the pipeline. Finally, Shapiro's algorithm [**?**] could act as a compromise between Steengaard's and Andersen's algorithm. In a way, Shapiro's analysis lies somewhere between the other two analyses. It is slower than Steengaard's, but also much more precise; and it is less precise than Andersen's, but also much faster.

Another way to improve on the execution time of the analyses is to drastically improve their implementations. Currently, the analyses are implemented

manually as abstract interpretations, and are not optimized further in any way. However, they could reimplemented in well-established, industrial-strength program analysis frameworks. One option would be the Soufflé Datalog compiler [**?**]. It uses Datalog to define logic-based program analyses, then compiles them to highly-parallelized C++ code. Soufflé facilitates implementing highly scalable data-flow analyses for whole program compilation.

10 Conclusions

In this paper we presented a modern look at GRIN, an optimizing functional language back end originally published by Urban Bouquist.

We gave an overview of the GRIN framework, and introduced the reader to the related research on compilers utilizing GRIN and whole program optimization. Then we gave an extension for the heap points-to analysis with more accurate basic value tracking. This allowed for defining a type inference algorithm for the GRIN intermediate representation, which then was used in the implementation of the LLVM back end. Following that, we detailed the dead data elimination pass and the required data-flow analyses, originally published by Remi Turk. We also presented an extension of the dummification transformation which is compatible with the typed representation of GRIN by extending the IR with the `undefined` value. Furthermore, we gave an alternative method for transforming producer-consumer groups by using basic blocks. Our last contribution was the implementation of the Idris front end.

We evaluated our implementation of GRIN using simple Idris programs taken from the book *Type-driven development with Idris* [**?**] by Edwin Brady. We measured the optimized GRIN programs, as well as the generated binaries. It is important to note, that the measurements presented in this paper can only be considered preliminary, given the compiler needs further work to be comparable to other systems. Nevertheless, these statistics are still relevant, since they provide valuable information about the effectiveness of the optimizer. The results demonstrate that the GRIN optimizer can significantly improve the performance of GRIN programs. Furthermore, they indicate that the GRIN optimizer performs optimizations orthogonal to the LLVM optimizations, which supports the motivation behind the framework. As for dead data elimination, we found that it can facilitate other transformations during the optimization pipeline, and that it can considerably reduce the size of the generated binaries.

All things considered, the current implementation of GRIN brought adequate results. However, there are still many promising ideas left to research.