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

Péter Dávid Podlovics, Csaba Hruska

February 1, 2019

Abstract

GRIN is short for Graph Reduction Intermediate Notation [1], 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 [2], and measure the framework's effectiveness compared to some of the most well-known functional language compilers such as the Glasgow Haskell Compiler [3] and the Idris compiler [4]. 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.

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 very sophisticated type systems. 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 Related Work

2.1 GRIN

The original GRIN framework was developed by U. Boquist, and first described in an article [5], then in his PhD thesis [1]. This version of GRIN used the Chalmers Haskell-B Compiler as its front end and RISC as its back end. At that time, his implementation of GRIN already compared favorably to the existing Glasgow Haskell Compiler of version 4.01.

2.2 Adaptations of GRIN

Other compilers also use GRIN as their back end. Probably the most notable one is the Utrecht Haskell Compiler [6]. UHC is a completely standalone Haskell compiler with its own front end. The main idea behind UHC is to use attribute grammars handle the ever-growing complexity of compiler construction in an easily manageable way.

2.3 Other Intermediate Representations

GRIN is not the only IR available for functional languages. In fact, it is not even the most advanced one. The Haskell Research Compiler [7] and the MLton [8] Standard ML compiler both use IR languages very similar to GRIN. However, these IRs are built from basic blocks instead of monadic bindings. This approach opens up a whole spectrum of new optimization opportunities. For example, the Haskell Research Compiler uses SIMD vectorization passes in its optimization pipeline, and achieves performance metrics comparable to native C [9].

2.4 Compilers with LLVM Back Ends

In the imperative setting, probably the most well-known compiler with an LLVM back end is Clang [10]. Clang's main goal is to provide a production quality compiler with a reusable, library-like structure. However, certain functional language compilers also have LLVM back ends. The two most notable ones are the Glasgow Haskell Compiler [3] and MLton [11].

3 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.

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 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.

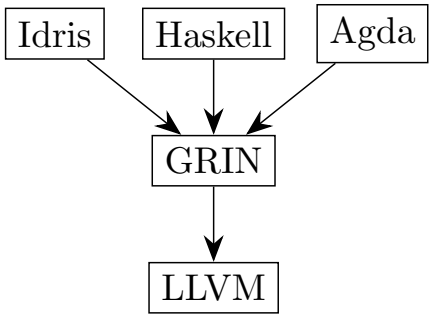

Figure 1: Possible representation of different function languages

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, 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.

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 [1] and further detailed in [5] 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 as well.

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.

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

```
main =
  n <- pure 5
  y <- pure (CInt n)
  pure 0
```

(a) Put into a data constructor

```
main =
  n <- pure 5
  foo n
foo x = pure 0
```

(b) Argument to a function call

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

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 can significantly decrease memory usage and reduce the number of heap operations.

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 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.

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

```
1 main =
2   x <- pure (CPair 0 1)
3   y <- snd x
4   pure y
5
6 snd p =
7   (CPair a b) <- pure p
8   pure b
```

(a) Before the transformation

$\xrightarrow{a \text{ is dead}}$

```
1 main =
2   x <- pure (CPair' 1)
3   y <- snd x
4   pure y
5
6 snd p =
7   (CPair' b) <- pure p
8   pure b
```

(b) After the transformation

Program code 6.1: A simple example for dead data elimination

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

Variable	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

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 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

[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.

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 2.

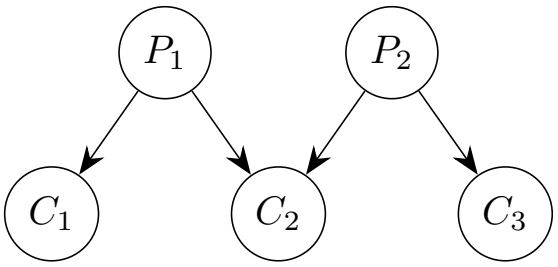

Figure 2: Producer-consumer group

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 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.

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. Unlike the other approach mentioned earlier, this alternative avoids any code duplication.

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] U. Boquist, "Code Optimisation Techniques for Lazy Functional Languages," Ph.D. dissertation, Chalmers University of Technology and Göteborg University, 1999.

[2] 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.

[3] C. V. Hall, K. Hammond, W. Partain, S. L. Peyton Jones, and P. Wadler, "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

[4] 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.

[5] 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

[6] A. Dijkstra, J. Fokker, and S. D. Swierstra, "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

[7] H. Liu, N. Glew, L. Petersen, and T. A. Anderson, "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

[8] 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

[9] 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

[10] "Clang: a C language family front end for LLVM." [Online]. Available: https://clang.llvm.org

[11] 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