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, and measure the framework's effectiveness compared to some of the most well-known functional language compilers such as the Glasgow Haskell Compiler and the Idris compiler. 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 a new optimization, the interprocedural dead data elimination.

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

Contents

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 approach results in more composable programs, but also presents more difficulties for compilation, since optimizing only individual functions is no longer sufficient.

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

2 Graph Reduction Intermediate Notation

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

First of all, interprocedural data-flow analysis preserves and propagates the flow of information through function calls allowing for optimizations across these functions. This means the framework can handle a large set of small interconnecting functions. Secondly, whole program analysis enables the optimization of global functions across modules. As a consequence, the implicit evaluation of suspended computations can be made explicit. In fact, the value forcing in GRIN is done by an ordinary function called `eval`. This is a global function uniquely generated for each program, implying it can be optimized just like any other function with the help of whole program analysis. Finally, GRIN can act as a common IR for all functional languages. After compiling programs written in those languages to the GRIN IR, they can be optimized and then low level machine code can be generated from them.

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 using conventional techniques already available. For example, it is possible to compile GRIN to LLVM and take advantage of an entire compiler framework providing a huge array of very powerful tools and features.

3 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.

3.1 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. The purpose of GRIN is to eliminate functional artifacts and restructure functional programs in a way 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, 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.

3.2 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 [2] 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.

4 Interprocedural 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 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.

4.1 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, in most cases the optimizations themselves do not even need to be modified.

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.

4.2 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. 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.

5 Compiled Abstract Interpretation

6 Related Work

7 Results

8 Conclusion

9 References

References

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

[2] 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