When should one inline recursive functions?

Christian Chavez

Nico Reissmann, Magnus Jahre, and Christian Chavez are with the Norwegian University of Science and Technology (NTNU).

***Index Terms*—Function inlining, Jive, Compiler, 2015, NTNU**

***Abstract*—Lorem ipsum...**

I. Introduction

A. Problemsetting

II. Background

A. RVSDG

- ϕ
- λ
- γ
- θ

III. Scheme

IV. Methodology

V. Results

VI. Discussion

VII. Related Work

In this section (...)

To do...

A. Read papers

Below papers are listed according to perceived relevance/usefulness for this project.

1) "**Secrets of the Glasgow Haskell Compiler inliner**", [2]
   This paper explains how inlining is dealt with by the GHC compiler. It spreads this into the following sections:
   a) Short section describing what inlining is, and its most obvious and generic properties and pitfalls (Section 2).
   b) An explanation of a strategy of how (and when) to inline recursive functions (Section 3).
   c) How they so deal with name capture (Section 4).
   d) As well as discussing certain moments/situations when inlining is carefully considered by the GHC. They also have a section describe how they exploit their name-capture solution to support accurate tracking of both lexical and evalutation-state environments (Section 5+6).
   e) Finally they sketch their implementation (Section 7).
2) "**Inline expansion: when and how?**", [3]
   This paper focuses on *when* and *how* to inline functions, classified as either *recursive* or *non-recursive* functions. This is done in the programming language Scheme, with the Bigloo compiler M. Serrano has created.

confirm this?

   The paper gives an algorithm for when to inline (which [2] remarks upon), as well as comments and results on their inlining wrt. the two classes of functions the paper defines.
3) "**Automatic Tuning of Inlining Heuristics**", [1]
   This paper explores and explains the advantages of genetic algorithm heuristics when dynamically compiling Java. Since it is dynamically compiled, it's harder for the compilation unit to get a proper global view of the compilation job, required resources, and beneficial trade-offs. All of which play important roles when deciding whether to incur the cost of increased code size when inlining.

VIII. Conclusion

A. Further Work

References

[1] John Cavazos and Michael F. P. O'Boyle. Automatic tuning of inlining heuristics. In *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, SC '05, pages 14–, Washington, DC, USA, 2005. IEEE Computer Society.

[2] Simon Peyton Jones and Simon Marlow. Secrets of the glasgow haskell compiler inliner. *J. Funct. Program.*, 12(5):393–434, July 2002.

[3] Manuel Serrano. Inline expansion: When and how? In *Proceedings of the9th International Symposium on Programming Languages: Implementations, Logics, and Programs: Including a Special Trach on Declarative Programming Languages in Education*, PLILP '97, pages 143–157, London, UK, UK, 1997. Springer-Verlag.

APPENDIX I
REPORT DESCRIPTION

Project Description: An Inliner for the Jive compiler

Nico Reissmann

December 12, 2014

Compilers have become an essential part of every modern computer system since their rise along with the emergence of machine-independent languages at the end of the 1950s. From the start, they not only had to translate between a high-level language and a specific architecture, but had to incorporate optimizations in order to improve code quality and be a par with human-produced assembly code. One such optimization performed by virtually every modern compiler is *inlining*. In principle, inlining is very simple: just replace a call to a function by an instance of its body. However, in practice careless inlining can easily result in extensive *work* and *code duplication*. An inliner must therefore decide carefully when and where to inline a function in order to achieve good performance without unnecessary code bloat.

The overall goal of this project is to implement and evaluate an inliner for the Jive compiler back-end. The project is split in a practical and an optional theoretical part. The practical part includes the following:

- Implementation of an inliner for the Jive compiler back-end. The inliner must be able to handle recursive functions and allow for the configuration of different heuristics to permit rapid exploration of the parameter space.
- An evaluation of the implemented inliner. A particular emphasis is given to different heuristics and their consequences for the resulting code in terms of work and code duplication.

The Jive compiler back-end uses a novel intermediate representation (IR) called the Regionalized Value State Dependence Graph (RVSDG). If time permits, the theoretical part of the project is going to clarify the consequences of using the RVSDG along with an inliner. It tries to answer the following research questions:

- What impact does the RVSDG have on the design of an inliner and the process of inlining?
- Does the RVSDG simplify/complicate the implementation of an inliner and the process of inlining compared to other commonly used IRs?

The outcome of this project is threefold:

1. A working implementation of an inliner in the Jive compiler back-end fulfilling the aforementioned criteria.
2. An evaluation of the implemented inliner.
3. A project report following the structure of a research paper.