Inlining in the Jive Compiler
Personal stuff

Christian Chavez

January 27, 2015

1 List of unread papers

- "**Practical and effective higher-order optimizations**",
  by Lars Bergstrom and Matthew Fluet and Matthew Le and John Reppy and Nora Sandler, at Mozilla Research CA and Rochester Institute of Technology NY and University of Chicago IL.
  This paper discusses inlining as well as some other optimizations. Picked for citing GHCPaper, and being from 2014, more than much else. I'm not sure I completely understand the abstract, but I got the feeling the paper might have an interesting take on inlining.
  `http://dl.acm.org/citation.cfm?id=2628153`

- "**Demand-driven Inlining Heuristics in a Region-based Optimizing Compiler for ILP Architectures***",
  by Tom Way and Ben Breech and Wei Du and Lori Pollock, at University of Delaware USA.
  This paper discusses a region-based compiling technique and how to utilize inlining in conjunction with this, while focusing on optimizations for ILP architectures. Picked paper for mentioning inlining in Keywords and abstract, as well as also dealing with region based compilation which seems relatable to Jive.
  `http://www.eecis.udel.edu/~hiper/passages/papers/waypdcsiasted01.pdf`

2 Read papers

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

1. "**Adaptive Compilation and Inlining**", by Todd Waterman (Ph.D. Thesis)

   Waterman's Ph.D. thesis [**?**] examines the use of techniques to adaptively decide which functions to inline during program compilation, to show that they can be used to improve the performance of specific optimizations.

   The adaptive technique both matches, and at times, beats ATLAS and gcc

need ATLAS cite?

on one of the object oriented biggest programs in the SPEC CINT2000 test suite. The technique accepts condition strings (like in [**?**]) to determine which call sites are inlined. The condition strings provide a flexible inliner by combining various program properties in CNF, and exposes a large space of different inlining decisions with the potential to outperform static techniques when used adaptively.

2. "**Inline Function Expansion for Compiling C Programs**", They discuss a lot of very relevant aspects regarding inlining in this paper. While some of the concerns they have are not as valid anymore due to newer technologies and hardware architectures, this paper is spot on for when and how to inline a C program from 1989.

3. "**An adaptive strategy for inline substitution**", Very much based on Todd Waterman's PhD Thesis it seems, they give a more updated version of his results, and explain a bit further how the parameterization scheme Waterman introduced in his thesis can and should work.

4. "**Secrets of the Glasgow Haskell Compiler inliner**", [**?**]

   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).

5. "**Inline expansion: when and how?**", [**?**]

   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 [**?**] remarks upon), as well as comments and results on their inlining wrt. the two classes of functions the paper defines.

6. "**Automatic Tuning of Inlining Heuristics**", [**?**]

   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.

7. "**Should Portential Loop Optimizations Influence Inlining Decisions?**",

   This paper discusses (as the title suggests) whether inlining should be improved upon or neglected, so as to improve potential loop optimizations. The authors of this paper tests their hypothesis on the Toronto Portable Optimizer (TPO), and their results show that loop-fusion is not affected in any major scale by furthering or removing inlining decisions. Their results are all comparable with the IBM XL compiler suite as is.

   They do however, discuss some of IBM XL compiler suite's criteria for inlining, but only very briefly in section 3.3, and I've not been able to find how their "Correctness" criteria is found/decided. The paper mentions no technical aspects of how they decide whether or not to inline in the rest of the paper.

8. "**Region-Based Compilation: An Introduction and Motivation**",

   Instruction level parallelism (**ILP**) has become an increasing focus in further development and improvement in code efficiency. This paper looks at how in a new technique, region-based compilation, can use inlining of functions to further the achievement of better code efficiency. The authors of this paper used the *Multiflow* compiler when doing their research for this paper.

9. "**Statitcally Unrolling Recursion to Improve Opportunities for Parallelism**",

   This paper focuses on how inlining should be done for the GHC, and how it is useful, and proving the correctness of their technique. It does not discuss when or why you should inline certain functions to any useful degree, beyond the usual "recursive functions can be very useful to inline with this technique".

Figure out the terms found in papers...

3 [?]

1. Section 1

   - "inlining subsumes" Something gets removed by inlining.
   - "lexical scopes" Scope of a program/function/block/variable. From StackOverflow: `http://stackoverflow.com/questions/1047454/what-is-lexical-scope`
   - "pure" (*language*) Program must always return same results with same inputs. There can be no "side-effect" which changes the state of the program.
   - "explicitly typed" (*language*) That the compiler knows at compile time (static) what types all the variables and return values have. Great explanation: `http://programmers.stackexchange.com/questions/181154/type-systems-nominal-vs-structural-explicit-vs-implicit`
   - "strictness analysis"

     Need more research

     `http://en.wikipedia.org/wiki/Strictness_analysis` `https://ghc.haskell.org/trac/ghc/wiki/Commentary/Compiler/Demand`
   - "*let*-floating" (*Haskell*) Nico: Not important.
   - "name capture" Making sure each name is an unique identifier. This gets complicated when compiler has to rename things for program correctness. Like when inlining.

2. Section 2

   - "β-reduction" A way of utilizing lambda-calculus to simplify a lambda expression. Closely related to simplifying anonymous functions (lambda functions) in pure functional programming languages like Haskell.
   - "invariant" (*language artifact/variable/expression?*) Constant/Unchanging: Unchanged by mathematical or physical operations or transformations.
   - "*trivial-constructor-argument invariant*" (*Haskell?*) See directly above.
   - "divergent computations"
   - "closure" (*scopes of functions?*) Ensuring correct lexical scope for (free)variables, mainly needed in languages permitting nested function.
   - "lambda calculus" An important language, focusing on "*computational mathematics*". It is *Turing complete*, and it can represent any program. As mentioned, it focuses on the computational operations of a (λ-)function, unlike "normal functions" which focus on their respective inputs and outputs.

- "literals" The string "abcd" is a literal. As is the number 1, when used to assign value to an integer, just like the string is used to assign value to a string-variable.
- "primitive operators" Operators that are in one way "basic", but not necessarily only (nor all) of the operations supported by hardware. Think of it as the most basic operators which together build up all others.

3. Section 3

   - "bound variable" A variable that is not free, that is, a variable which has been specified as an input to a function call. (The opposite of a free variable, which is a variable used in a function, but declared outside of it, and not listed as one of its parameters).
   - "recursive binding groups" When you have calls between groups of recursive functions, complicated situations arise. See [**?**], Section 3.5 for example of a "recursive binding group".
   - "strongly-connected components" A group of nodes (*components*) in a DAG where you can reach every other node/component/vertex from any other. Hence, a DAG might have several *strongly-connected components*, composed of a multiple of nodes/vertices/components.
   - "contravariantly" (*(..) it appears contravariantly in its own definition.*) Best guess: This term refers to variables or functions which are declared, and then used in a manner contradicting the nature of their declaration. Which possibly can become a problem when inlining recursive functions.
   - "untyped programs" Nico: Not relevant.
   - "pathological programs" Forgot, ask Nico again.
   - "static analysis" Analysis at compile time.

4. Section 4

   - "hash-consing"