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...**

In this section (...)

To do...

A. Unread (but found interesting) papers

- "**On building a Supercompiler for GHC**"',
  Peter A. Jonsson and Johan Nordlander, at Luleå University of Technology.
  A paper describing how to build a "Supercompiler", which skips intermediate representation (IR). Sounds relatively useless, but there are quite a few papers talking about inlining in conjunction with Supercompilers. On this list in case it becomes of relevance later on.
  http://pure.ltu.se/portal/files/2231262/nwpt08-scp.pdf
- "**Optimizing Generics Is Easy!**",
  written by José Pedro Magalhães and Stefan Holdermans and Johan Jeuring and Andres Löh, at Utrecht University, The Netherlands.
  Looks more like a description of how GHC inlines than anything else on the subject of inlining.
  http://www.andres-loeh.de/OptimizingGenerics/
- "**Should potential loop optimizations influence inlining decisions?**", by Christopher Barton and José Nelson Amaral and Bob Blainey, at University of Alberta and IBM Toronto Software Laboratory, Canada.
  Perhaps 2nd most interesting at this moment?
  http://dl.acm.org/citation.cfm?id=961329
- "**Region-based compilation: an introduction and motivation**",
  byRichard E. Hank and Wen-Mei W. Hwu, at University of Illinois, and B. Ramakrishna Rau, at Hewlett Packard Laboratories, Palo Alto California.
  3rd most interesting on this list at the moment.
  http://dl.acm.org/citation.cfm?id=225160.225189&coll=DL&dl=ACM&CFID=470941312&CFTOKEN=25125832
- "**Bigloo: a portable and optimizing compiler for strict functional languages**",
  by Manuel Serrano and Pierre Weis.
  Does not really seem interesting/relevant, but Manuel Serrano has written several relevant papers, and strict functional languages are some of the more viable languages for this type of optimization...
  http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.50.8424
- "**Losing functions without gaining data: another look at defunctionalisation**",
  by Neil Mitchell and Colin Runciman, at University of York, UK.
  Not really sure what to make of this one.
  http://dl.acm.org/citation.cfm?id=1596641
- "**Statically Unrolling Recursion to Improve Opportunities for Parallelism**",
  by Neil Deshpande and Stephen A. Edwards, at Columbia University NY.
  This one sounds cool, and being static is a plus. But that's all its got going for it without having read it more closely. It does discuss inlining.
  http://www.cs.columbia.edu/~sedwards/papers/deshpande2012statically.pdf
- "**An adaptive strategy for inline substitution**",
  by Keith D. Cooper and Timothy J. Harvey and Todd Waterman, at Rice University Houston and Texas Instruments Inc. Stafford, Texas.
  Perhaps most interesting at this time due to discussing global/static optimizations as well as being relatively new compared to other papers in this list?
  http://dl.acm.org/citation.cfm?id=1788381
- "**Adaptive compilation and inlining**",
  by Todd Waterman and Keith D. Cooper, at Rice University Houston Texas.
  Not sure if this is what Nico wants me to find, but this is a paper discussing aspects of inlining I don't think I've considered much in earlier paper searches.
  http://dl.acm.org/citation.cfm?id=1195126
- "**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 [2], 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

B. 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.

VII. 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
DICTIONARY

Figure out the terms found in papers...

A. [2]

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 [2], 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"

APPENDIX II
TESTY

Meeeeehhh....