Draft: October 3, 2008

On building a Supercompiler for GHC

Peter A. Jonsson Johan Nordlander

Luleå University of Technology

{pj, nordland}@csee.ltu.se

Abstract

Supercompilation is a program transformation that removes intermediate structures and performs program specialization. We discuss problems and necessary steps for building a supercompiler for GHC.

1. Introduction

Supero [4] is a supercompiler for Haskell that has achieved runtime improvements of 16% for a subset of the imaginary part of the nofib suite compared to GHC. This is a remarkable result considering that it is doing a single program transformation beyond those of GHC and that GHC is a mature optimizing Haskell compiler. Not only is Supero showing great results, but the theoretical underpinnings of supercompilation have been investigated quite thoroughly as well. We know that it is possible to supercompile both strict [2] and lazy [9] languages, that the algorithms preserve the semantics of the program [7] and that the algorithms will terminate [8].

If the theory of supercompilers is well investigated, and runtime improvements are impressive – why does not every compiler include the optimization? Mitchell and Runciman [4] list three main areas in the future work for Supero:

Runtime performance For certain benchmarks earlier versions of Supero had better results. This is both evidence that there are improvements to be made, and it also highlights how small changes to the algorithm can have large effects on the result of the optimization.

Compilation speed Profiling Supero showed that 90% of the time was spent to ensure termination (the homeomorphic embedding test), which was done in a naïve way.

More benchmarks Supero was applied to a subset of the imaginary part of nofib. No one has ever benchmarked supercompiled real world Haskell programs.

We set out to build a supercompiler for GHC in order to tackle these three problems (Sections 3 and 4). Using GHC gives the additional benefit of support for many Haskell extensions for free, since they are all translated to System F_c, GHC's typed intermediate language [10]. We start with some examples of a supercompiler in action to try to convey the intuition behind the algorithm (Section 2).

2. Examples

We have left out many of the technical details of the algorithm due to space constraints and instead try to convey the intuition via a series of examples of how a supercompiler behaves. Readers who are interested in the gory details are encouraged to read some of the related work already cited, in particular Sørensen et al. [9] for a call-by-name algorithm, the work on Supero by Mitchell and Runciman [4], or Jonsson and Nordlander [2] for a call-by-value algorithm.

Supercompilation is a program transformation, closely related to deforestation [11], that both removes intermediate structures and performs program specialization. Removing intermediate structures will make the program allocate less memory and thus put less strain on the garbage collector. The program specialization will in practice remove many of the higher order functions, replacing them with a specialized first order variant. Higher order languages usually represent functions as closures on the heap, and these closures need to be garbage collected. Having removed them at compile-time reduces the amount of garbage collection necessary, and also avoids an indirect jump to the function pointer in the closure at runtime.

Our first example is transformation of $sum\ (map\ square\ ys)$. The functions used in the examples are defined as:

$$\begin{array}{lll}
square\ x & = & x * x \\
map\ f\ xs & = & \textbf{case}\ xs\ \textbf{of} \\
 & & \quad [] \rightarrow ys \\
 & & \quad (x : xs) \rightarrow f\ x : map\ f\ xs \\
sum\ xs & = & \textbf{case}\ xs\ \textbf{of} \\
 & & \quad [] \rightarrow 0 \\
 & & \quad (x : xs) \rightarrow x + sum\ xs
\end{array}$$

We start our transformation by allocating a new fresh function name (h_0) to this expression, inlining the body of sum and substituting $map\ square\ ys$ into the body of sum:

$$\begin{array}{l}
\textbf{case}\ map\ square\ ys\ \textbf{of} \\
\quad [] \rightarrow 0 \\
\quad (x' : xs') \rightarrow x' + sum\ xs'
\end{array}$$

After inlining map and substituting the arguments into the body the result becomes:

$$\begin{array}{l}
\textbf{case}\ (\ \textbf{case}\ ys\ \textbf{of} \\
\qquad\qquad [] \rightarrow [] \\
\qquad\qquad (x' : xs') \rightarrow (square\ x') : map\ square\ xs')\ \textbf{of} \\
\quad [] \rightarrow 0 \\
\quad (x' : xs') \rightarrow x' + sum\ xs'
\end{array}$$

We duplicate the outer case in each of the inner case's branches, using the expression in the branches as head of that case-statement. Continuing the transformation on each branch with ordinary reduction steps yields:

$$\begin{array}{l}
\mathbf{case}\ ys\ \mathbf{of} \\
\quad [\,] \rightarrow 0 \\
\quad (x' : xs') \rightarrow \mathit{square}\ x' + \mathit{sum}\ (\mathit{map}\ \mathit{square}\ xs')
\end{array}$$

Now inline the body of the first square and observe that the second argument to $(+)$ is similar to the expression we started with. We replace the second parameter to $(+)$ with $h_0\ xs'$. The result of our transformation is $h_0\ ys$, with h_0 defined as:

$$\begin{array}{ll}
h_0\ ys = & \mathbf{case}\ ys\ \mathbf{of} \\
& \quad [\,] \rightarrow 0 \\
& \quad (x' : xs') \rightarrow x' * x' + h_0\ xs'
\end{array}$$

This new function only traverses its input once, and no intermediate structures are created. If the expression $\mathit{sum}\ (\mathit{map}\ \mathit{square}\ xs)$ or a renaming thereof is detected elsewhere in the input, a call to h_0 will be inserted there instead.

The following examples are due to Ohori and Sasano [5]:

$$\begin{array}{ll}
\mathit{mapsq}\ xs & = \mathbf{case}\ xs\ \mathbf{of} \\
& \quad [\,] \rightarrow [\,] \\
& \quad (x' : xs') \rightarrow (x' * x') : \mathit{mapsq}\ xs' \\
f\ xs & = \mathbf{case}\ xs\ \mathbf{of} \\
& \quad [\,] \rightarrow [\,] \\
& \quad (x' : xs') \rightarrow (2 * x') : g\ xs' \\
g\ xs & = \mathbf{case}\ xs\ \mathbf{of} \\
& \quad [\,] \rightarrow [\,] \\
& \quad (x' : xs') \rightarrow (3 * x') : f\ xs'
\end{array}$$

Transforming $\mathit{mapsq}\ (\mathit{mapsq}\ xs)$ will inline the outer mapsq, substitute the argument in the function body and inline the inner call to mapsq:

$$\begin{array}{l}
\mathbf{case}\ (\ \mathbf{case}\ xs\ \mathbf{of} \\
\qquad\qquad [\,] \rightarrow [\,] \\
\qquad\qquad (x' : xs') \rightarrow (x' * x') : \mathit{mapsq}\ xs')\ \mathbf{of} \\
\quad [\,] \rightarrow [\,] \\
\quad (x' : xs') \rightarrow (x' * x') : \mathit{mapsq}\ xs'
\end{array}$$

As previously, we duplicate the outer case in each of the inner case's branches, using the expression in the branches as head of that case-statement. Continuing the transformation on each branch by ordinary reduction steps yields:

$$\begin{array}{l}
\mathbf{case}\ xs\ \mathbf{of} \\
\quad [\,] \rightarrow [\,] \\
\quad (x' : xs') \rightarrow (x' * x' * x' * x') : \mathit{mapsq}\ (\mathit{mapsq}\ xs')
\end{array}$$

This will encounter a similar expression to what we started with, and create a new function h_1. The final result of our transformation is $h_1\ xs$, with the new residual function h_1 that only traverses its input once defined as:

$$\begin{array}{ll}
h_1\ xs = & \mathbf{case}\ xs\ \mathbf{of} \\
& \quad [\,] \rightarrow [\,] \\
& \quad (x' : xs') \rightarrow (x' * x' * x' * x') : h_1\ xs'
\end{array}$$

For an example of transforming mutually recursive functions, consider the transformation of $\mathit{sum}\ (f\ xs)$. Inlining the body of sum, substituting its arguments in the function body and inlining the body of f yields:

$$\begin{array}{l}
\mathbf{case}\ (\ \mathbf{case}\ xs\ \mathbf{of} \\
\qquad\qquad [\,] \rightarrow [\,] \\
\qquad\qquad (x' : xs') \rightarrow (2 * x') : g\ xs')\ \mathbf{of} \\
\quad [\,] \rightarrow 0 \\
\quad (x' : xs') \rightarrow x' + \mathit{sum}\ xs'
\end{array}$$

We now move down the outer case into each branch, and perform reductions until we end up with:

$$\mathbf{case}\ xs\ \ \mathbf{of}\,\{[\,] \rightarrow 0;\ (x' : xs') \rightarrow (2 * x') + \mathit{sum}\ (g\ xs')\ \}$$

We notice that unlike in previous examples, $\mathit{sum}\ (g\ xs')$ is not similar to the expression we started with. For space reasons, we focus on the transformation of the rightmost expression in the last branch, $\mathit{sum}\ (g\ xs')$, while keeping the functions already seen in mind. We inline the body of sum, perform the substitution of its arguments and inline the body of g:

$$\begin{array}{l}
\mathbf{case}\ (\ \mathbf{case}\ xs'\ \mathbf{of} \\
\qquad\qquad [\,] \rightarrow [\,] \\
\qquad\qquad (x'' : xs'') \rightarrow (3 * x'') : f\ xs'')\ \mathbf{of} \\
\quad [\,] \rightarrow 0 \\
\quad (x' : xs') \rightarrow x' + \mathit{sum}\ xs'
\end{array}$$

We now move down the outer case into each branch, and perform reductions:

$$\begin{array}{l}
\mathbf{case}\ xs'\ \mathbf{of} \\
\quad [\,] \rightarrow 0 \\
\quad (x'' : xs'') \rightarrow (3 * x'') + \mathit{sum}\ (f\ xs'')
\end{array}$$

We notice a familiar expression in $\mathit{sum}\ (f\ xs'')$, and fold when reaching it. Adding it all together gives a new function h_2:

$$\begin{array}{ll}
h_2\ xs = & \mathbf{case}\ xs\ \mathbf{of} \\
& \quad [\,] \rightarrow 0 \\
& \quad (x' : xs') \rightarrow (2 * x') + \mathbf{case}\ xs'\ \mathbf{of} \\
& \qquad\qquad\qquad\qquad\qquad\qquad [\,] \rightarrow 0 \\
& \qquad\qquad\qquad\qquad\qquad\qquad (x'' : xs'') \rightarrow \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad (3 * x'') + h_2\ xs''
\end{array}$$

Kort [3] studied a ray-tracer written in Haskell, and identified a critical function in the innermost loop of a matrix multiplication, called vecDot:

$$\mathit{vecDot}\ xs\ ys = \mathit{sum}\ (\mathit{zipWith}\ (*)\ xs\ ys)$$

This is simplified by our positive supercompiler to:

$$\begin{array}{ll}
\mathit{vecDot}\ xs\ ys & = h_1\ xs\ ys \\
h_1\ xs\ ys & = \mathbf{case}\ xs\ \mathbf{of} \\
& \quad (x' : xs') \rightarrow \mathbf{case}\ ys\ \mathbf{of} \\
& \qquad\qquad\qquad (y' : ys') \rightarrow \\
& \qquad\qquad\qquad\qquad x' * y' + h_1\ xs'\ ys' \\
& \qquad\qquad\qquad \_ \rightarrow 0 \\
& \quad \_ \rightarrow 0
\end{array}$$

The intermediate list between *sum* and *zipWith* is transformed away, and the complexity is reduced from $2|xs|+|ys|$ to $|xs|+|ys|$ (since this is matrix multiplication $|xs| = |ys|$).

3. Towards a Supercompiler in GHC

The first step towards a supercompiler in GHC is to construct a supercompilation algorithm for System F_c, the typed intermediate language found in GHC [10]. The conversion of the algorithm is quite straightforward and in our experience it is rare to accidentally introduce non-termination or unsound transformation steps. The challenge is rather to achieve the desired transformation effects in the presence of the casts (▶) that might propagate inside expressions in System F_c. Previous work on supercompilation has been for untyped languages. Since GHC assumes well-typed expressions once the type-checker pass is done it is necessary to prove that the supercompiler preserves types as well

Mitchell and Runciman lists three choices that need to be made during optimization:

- Which function to inline.
- What termination criterion to use.
- What generalisation to use.

Our current implementation makes a fixed choice for function to inline (left-most), what termination criterion to use (the homeomorphic embedding), and what generalisation to use (the most specific generalisation, or simple splitting in the case where no common terms are found).

4. Analysis of Problems

4.1 Code Explosion

It is possible to construct examples where a module exports two mutually recursive functions, and if one supercompiles both these functions independently it might lead to code duplication. We have sidestepped this issue by only supercompiling one function, *main*, which makes our supercompiler unusable for libraries at the moment. However, upon compiling the entire program with the library present the desired optimization should occur. We expand on issues with whole program compilation in Section 4.3.

Our current strategy to always inline the left-most function is not always beneficial for performance since it might lead to code explosion, and possibly evaluating the same expression multiple times. An example is the following Haskell-program:

$$\begin{aligned} main \ = \ & \mathbf{do} \\ & \quad x \ \leftarrow \ getArgs \\ & \quad xs \ \leftarrow \ readFile\ (x \,!!\, 2) \\ & \quad ys \ \leftarrow \ readFile\ (x \,!!\, 3) \\ & \quad doCompute\ xs\ ys \\ & \quad return\ () \end{aligned}$$

which will give two copies of readFile: one $readFile_{!!2}$ and one $readFile_{!!3}$, something that is not necessarily faster than simply calling *readFile* directly with the different parameters.

Inlining itself is a difficult problem, which is not that well studied. The inliner of GHC has been investigated previously [6], and we expect many results from that investigation to carry over to our supercompiler. We find it likely that more work is necessary for inlining efficiently in our supercompiler, possible directions for work include, but are not limited to:

- Investigate how inliners of other compilers than GHC work.
- Characterize what a good or bad inlining is.
- Create a partial order between inlinings.

With the above knowledge, it should be possible to design a heuristic that works well in practice.

4.2 Compilation Speed

We propose to make the homeomorphic embedding test on a smaller part of the tree, which still preserves termination of the algorithm. Any improvements in the implementation of the homeomorphic embedding test will be of benefit both to our approach and to Supero. Changing the test makes it impossible to extend our algorithm to distillation [1], which removes more intermediate structures. This a trade-off we are prepared to accept considering that supercompilation is still an improvement over what is currently in use today. Should we change our mind in the future it should not be a problem to go back to the kind of test Supero uses. It is still too early to tell whether this change is enough, or if there are other bottlenecks that will show when supercompiling larger programs.

4.3 Whole Program Compilation

Currently, our experiments have only been on complete programs defined in one module, avoiding to import the Prelude. To gain the most out of supercompilation on real world programs GHC needs to be tweaked to handle whole program compilation. This can be done by removing the current constraint that expressions placed in the interface files (.hi) must be "small", and by annotating loop-breakers in the interface files. The size increases to the interface files from these changes need to be measured.

Whole program compilation has been used in MLton and they manage to compile programs larger than 100k lines [12]. If this number carries over to GHC and Haskell it makes whole program compilation a viable approach for a majority of the known Haskell programs.

5. Conclusions and Future Work

We have reported on our current work on building a supercompiler for GHC. Many problems that we intend to tackle have been mentioned already, among them a proof of type preservation of the algorithm, investigating inlining to avoid code explosion, and measuring the effects of making the interface files contain entire modules to achieve whole program compilation. We are however certain that problems which we have not foreseen will surface as we make progress on our supercompiler.

Acknowledgements

We thank Simon Peyton Jones for suggesting several of the examples and explaining how to achieve whole program compilation with GHC. Max Bolingbroke has patiently explained large parts of the GHC internals to us and deserves a special thanks.

References

[1] G. W. Hamilton. Distillation: extracting the essence of programs. In *PEPM '07: Proceedings of the 2007 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 61–70, New York, NY, USA, 2007. ACM.

[2] P. A. Jonsson and J. Nordlander. Positive supercompilation for a higher-order call-by-value language. In *POPL '09: Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, New York, NY, USA, 2009. ACM. To appear.

[3] J. Kort. Deforestation of a raytracer. Master's thesis, University of Amsterdam, 1996.

[4] N. Mitchell and C. Runciman. A supercompiler for core Haskell. In O. Chitil et al., editor, *IFL 2007*, volume 5083 of *Lecture Notes in Computer Science*, pages 147–164. Springer-Verlag, 2008.

[5] A. Ohori and I. Sasano. Lightweight fusion by fixed point promotion. In *POPL '07: Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 143–154, New York, NY, USA, 2007. ACM.

[6] S. L. Peyton Jones and S. Marlow. Secrets of the glasgow haskell compiler inliner. *J. Funct. Program*, 12(4&5):393–433, 2002.

[7] D. Sands. Proving the correctness of recursion-based automatic program transformations. *Theoretical Computer Science*, 167(1–2):193–233, 30 October 1996.

[8] M.H. Sørensen. Convergence of program transformers in the metric space of trees. *Sci. Comput. Program*, 37(1-3):163–205, 2000.

[9] M.H. Sørensen, R. Glück, and N.D. Jones. A positive supercompiler. *Journal of Functional Programming*, 6(6):811–838, 1996.

[10] M. Sulzmann, M. M. T. Chakravarty, S. Peyton Jones, and K. Donnelly. System F with type equality coercions. In *TLDI '07: Proceedings of the 2007 ACM SIGPLAN international workshop on Types in languages design and implementation*, pages 53–66, New York, NY, USA, 2007. ACM.

[11] P. Wadler. Deforestation: transforming programs to eliminate trees. *Theoretical Computer Science*, 73(2):231–248, June 1990.

[12] S. Weeks. Whole-program compilation in MLton. In *ML '06: Proceedings of the 2006 workshop on ML*, pages 1–1, New York, NY, USA, 2006. ACM. http://mlton.org/pages/References/attachments/060916-mlton.pdf.