Optimal Loop Breaker Choice for Inlining

Bas den Heijer

Master Thesis

Supervised by
Dr. H.L. Bodlaender
Dr. A. Dijkstra

Thesis number
ICA-3019810

Universiteit Utrecht

Abstract

When inlining recursive functions care must be taken to ensure termination of the inliner. The Glasgow Haskell Compiler does this by choosing a *loop breaker* (which may never be inlined) for each recursion loop in the program. These breakers are currently chosen by a simple heuristic which makes little effort to minimise the number of breaker vertices chosen, even though there are several cases where a smarter choice of of a set of loop breaker vertices can be much smaller. In this paper we apply algorithmic techniques (related to the Directed Feedback Vertex Set and Feedback Arc Set) to determine the minimum size set of breakers necessary to break every loop. Our approach requires significantly fewer loop breakers.

Keywords: compiler, Haskell, GHC, optimisation, inlining, directed blackout feedback vertex set, blackout feedback arc set

1 Introduction

Inlining is the act of replacing a variable or function reference with its definition. It is an optimisation technique commonly used by compilers to speed up programs. This is especially useful in (pure) functional programming languages as every line tends to consist of many references and they can all be safely inlined without changing the programs semantics because there are no side-effects. There are two possible advantages to an inlining: firstly it saves a dereference at runtime; this advantage may be minuscule, but in a hot inner loop this small gain can add up to significant time savings. Secondly an inlining can expose other optimisation opportunities: for example it could remove the last reference to a variable, making it dead code so it won't have to be compiled or it could allow some static evaluation eliminating a branch.

Inlining is not without its downsides though: first of all it increases code size because it essentially copies code, which can make the size of the binary outgrow the processor cache and significantly slow the program down, and secondly it can lead to duplication of effort at runtime[1]. Also, since inlining creates new references it can also create new opportunities to inline so effort must be taken to design the inlining algorithms to terminate within acceptable time constraints. This is especially the case with recursive functions, where unscrupulous inlining could potentially never terminate. The prevention of this last scenario is the focus of this paper.

1.1 Current state of GHC

Peyton-Jones and Marlow have thoroughly described their experiences with the design and implementation the Glasgow Haskell Compiler inliner [14]. The details of the implementation have largely remained unchanged in the intervening decade. They introduce the concept of loop breakers to guarantee termination of the inliner even when faced with recursive functions.

Consider the programs reference graph: each variable is a vertex, and each reference is an arc from the vertex of the innermost variable of the reference-site to the reference variable's vertex. Every loop in this graph represents a group of mutually recursive functions and as such a place where the inliner might diverge if left unchecked. GHC's strategy is to choose certain variables to be loop breakers and never inlining those (removing its incoming arcs from the inliners perspective). By making sure that every cycle in the graph has at least

[1] Actually in this case the inverse of inlining is beneficial, this is called *common subexpression elimination*.

one loop breaker it is guaranteed that the inliner will never loop.

The challenge is to select a good set loop breakers, hereto there are two criteria we should try to optimise for the set of loop breaker variables:

- Try not to select a variable that it would be very beneficial to inline.
- Try to select as few variables as possible.

GHC currently employs a heuristic to choose loop breakers which uses the first of these criteria but puts little effort into the second. It assigns priorities[2] to every vertex based on language specific knowledge to gauge the desirability to inline every vertex, the higher the priority the more we would like to inline that variable. Some examples of priorities:

- Priority = 10: The expression is trivial (such as `f = g` or `f = 32`) or is marked with the INLINE pragma.
- Priority = 9: The expression is a dictionary tuple, which is a common pattern generated by the front-end to model the class system with overloaded functions.
- Priority = 5: The expression is a constructor application. This means inlining it might eliminate a case expression.
- Priority = 2: The variable occurs only once.
- Priority = 0: Any other expression.

It proceeds to choose the loop breakers as follows:

1. Break up the reference graph into strongly-connected components.
2. For each component with a cycle:
   (a) Choose a variable with the lowest priority available and mark it as a loop breaker.
   (b) Remove its incoming arcs and repeat the algorithm with the resulting graph from step 1.

A performance optimisation not mentioned in the paper is that per group of recursive functions (so per recursive let) it will only go through this loop twice. If the component is still cyclic after picking two variables as loop breakers it will make every variable with the lowest score loop breaker and restart the algorithm.

[2]In the original paper and GHC source code these values are called scores, but we will use the term priority to avoid confusion later on.

1.2 Our contribution

As mentioned, the heuristic only superficially tries to minimise the amount of loop breakers; consider the example in Figure 1. Here the vertices represent all the variables in a program, and arc (f, g) indicates that f has a top-level reference to g. We call this graph the *reference graph* or *call graph*. We would like to just break on h because that would break all loops using just one loop breaker. It is possible that the heuristic would 'guess' this variable to break on, but it could also choose one of f, g, p or q in which case another variable has to become a loop breaker to break the other loop. In more complicated recursive nests the difference between the minimum amount of loop breakers necessary and amount picked by GHC gets larger. We show more examples from real programs in Section 4.

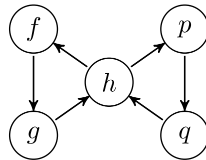


Figure 1: A simple reference graph with a double loop. All priorities are equal.

We have designed and tested an branch-and-bound algorithm that will find the minimum number of loop breakers necessary. In Section 2 we show how to reduce the problem to the well known Directed Blackout Feedback Vertex Set. Section 3 documents the algorithm we use to solve that problem using preprocessing rules adjusted to this use case. In Section 4 we discuss our findings on applying this algorithm to the nofib-benchmark suite.

Section 5 presents a parallel approach we propose, taking not only the reference-target into account, but also the call-*site*. Essentially looking at the arcs instead of the vertices in the call graph.

2 The Feedback Vertex Set

The problem the inliner faces (ignoring priorities for now) when choosing loop breakers is an instance of the Feedback Vertex Set problem from graph theory: given a graph, find the smallest subset of its vertices such that each cycle of the graph contains at least one vertex from the subset. More explicitly we look at the Directed Feedback Vertex Set problem (DFVS): which is the same problem applied to directed graphs, since a variable reference is fundamentally directed.

Or stated more formally: if $G = (V, E)$ is a directed graph, a feedback vertex set $F \subseteq V$ (FVS) is a set of vertices such that the removal of F and its incident arcs from the graph G leaves G without cylces, or equivalently, each cycle in graph G contains at least one vertex in F. The optimisation version of the Directed Feedback Vertex Set problem is: given a directed graph G, find a minimum FVS.

Apart from this use in inlining, the DFVS problem has applications in many areas including electrical circuit design [9] and deadlock mitigation in multi-threaded operating systems [19]. In the latter case threads are modelled as vertices and arcs signify a waiting-for-state between the threads so that a cycle in that graph indicates a deadlock where no thread is able to continue executing. The DFVS would give a minimum set of threads that have to be terminated to mitigate the deadlock. Even within the field of compilers there are there are more applications: pre-emptive concurrency sometimes requires a garbage-collection safe point on every loop [14].

Furthermore there is an application of DFVS in genetics that is not often cited in computer science literature that is richly documented and interestingly also referred to as loop breaking [3, 2, 21, 1, 13, 11].

2.1 Related combinatorial literature

The Directed Feedback Vertex Set is considered a classical combinatorial problem and is a well studied problem. Its decision variant is among Karp's seminal list of 21 NP-Complete problems (as the Feedback Node Set problem) [15]. A naïve exact branching algorithm costs $O^*(2^n)$ time. Recently the state of the art was improved to $O^*(1.9977^n)$ [18]. This is a theoretic breakthrough but forms no practical improvement for compilers.

Another major recent theoretical breakthrough was the discovery of a fixed-parameter algorithm for DFVS, which given a parameter k constructs a feedback vertex set of size at most k or reports that no such set exists in $O(4^k k^3 k! n^4)$, where n is the amount of vertices [6]. Unfortunately for our problem this solution is still intractable even for relatively modest values of k and we have no evidence that call graphs have particularly small feedback vertex sets, but this could be an interesting avenue for further research. Also there is something to be said for keeping it simple: an easier to understand algorithm is more maintainable.

The related undirected version of the classical problem, the Feedback Vertex Set, is also well studied. It seems there is no straightforward way to reduce a DFVS instance to an equivalent FVS instance; the problems are fundamentally different so we cannot apply these results directly. Still, the problems are similar

enough that the approaches and specifically the reduction rules can be used as an inspiration. The fastest exact algorithm known costs $O(1.7548^n)$ [10]. From the fixed-parameter perspective some significant recent results include a cubic kernel [4] and a more recent $4k^2$ kernel [20]. These kernalization algorithms reduce – in polynomial time – the input graph into a smaller equivalent graph bounded in size by a function of k, in this case $4k^2$. Some of the reduction rules used in these algorithms have close equivalents in our algorithm.

2.2 Priorities to black outs

The missing ingredient in DFVS for our problem are the priorities that are assigned to the vertices: the classical DFVS problem has no mechanism to account for these. In order to add these to the input we use the Directed Blackout Feedback Vertex Set problem (DBFVS), a generalisation of DFVS which allows some vertices to be forbidden from being chosen in the Feedback Vertex Set. I.e. an instance of the DBFVS consists of a directed graph $G = (V, E)$ and a subset of the vertices $W \subseteq V$; the vertices in W are called "blacked out". In the problem we look for a minimum size set of vertices $X \subseteq V - W$ such that each cycle in G contains a vertex in X.

We preprocesses the graph with priorities for each vertex (the input the loop breaker algorithm gets in GHC) to an instance of DBFVS. We call this part of the algorithm the *priowiggle*:

1. Split the graph into strongly connected components.

2. Find the lowest priority p such that removing every vertex with a priority equal to or lower than p leaves an acyclic graph. We will call this p the *hot priority*; at least one vertex of this priority will have to be a loop breaker, and there are no cycles on which all vertices have a priority higher than p.

3. Black out every vertex of priority higher than p.

The second step is accomplished by iterating over the occurring priorities (P) in ascending order and removing vertices of that priority, each time checking with a depth first search if the remaining graph is still cyclic and stopping when it is. This takes $O(|P||V|)$ time and dominates the other two steps which only take $O(|V|)$ time.

The resulting DBFVS-instance follows the same rules as the current GHC algorithm in that it will not break a loop on a vertex of priority p when it is possible to break every loop in that component by only breaking vertices with

a lower priority. There is however a subtle difference because it will not force vertices with a priority lower than the hot priority to be loop breakers.

Consider Figure 2. The GHC-algorithm would start out looking only at the vertices of priority 1 as potential loop breakers, and as there is a cycle of vertices of priority 2 removing these will never break all cycles, make all of these vertices loop breakers before considering the priority 2 vertices. That is undesirable because it is obvious that every loop in this graph can be broken by marking only the grey vertex as a loop breaker (i.e. there is a Feedback Vertex Set of size 1). The priowiggle would in this case not black out any vertex and leave the vertices of priority 1 available for inlining.

Note that after this step, we will not look at priorities again, as the relevant information is encoded in the set of blacked out vertices.

3 An Exact Algorithm For Loop Breaker Choice

After the preprocessing described in Section 2.2 we are left with an instance of the Directed Blackout Feedback Vertex Set problem, consisting of a single strongly connected component. Our algorithm consists of two phases and results in a minimum size set of vertices that can break all cycles. The first phase simplifies the input component, removing as many edges and vertices as possible without changing the result, the second phase applies a branch and bound strategy to determine the fate of the remaining vertices.

This form of preprocessing resembles the notion of kernalization [7], but without a proven bound on the resulting instance sizes.

Generally, a DBFVS instance is not always solvable. For example, the instance could contain a blacked out vertex with a self-loop. However due to the design of the priowiggle, it is guaranteed that we will not encounter such instances: specifically, each cycle in the input contains at least one non-blacked

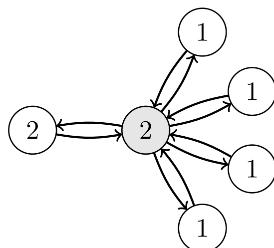


Figure 2: An example where the GHC priority system fails. Numbers represent priorities.

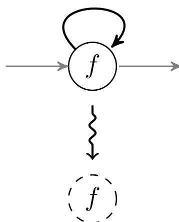
out vertex. We maintain this property during the preprocessing.

3.1 Phase 1: Preprocessing

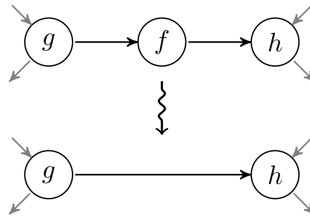
Before we surrender our graph to the exponential-time second phase we try to simplify the graph by repeatedly applying some preprocessing rules until they cannot be applied any more. These likely do not yield a better asymptotic worst case running time on general graphs, but allow us to tune the algorithm to the application and greatly speed up the second phase by attacking common patterns in the call graphs we have encountered. Each of the preprocessing rules transforms the graph to a smaller, equivalent graph and can be said to be *safe*: if G is transformed into G', then a minimum FVS for G' can be directly translated into a minimum FVS for G, i.e., the rules to not affect optimality.

We now discuss each of the preprocessing rules separately. In the illustrations a wavy arrow represents the application of a rule: starting with the precondition and ending with the result after application; grey arcs represent arcs that might be present but are not necessary for the rule; blacked out vertices are black; and loop breakers have a dashed circumference (being invisible to the inliner).

1. **Break Self-loop** If a vertex f has a self-loop, i.e. there is an arc (f, f), then f must be a loop breaker. Mark f as such and remove it and its incident arcs from the working graph.

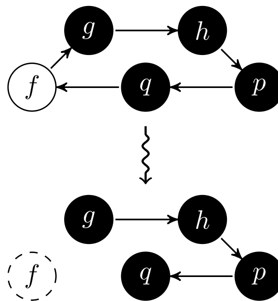


2. **Shortcut Degree Two** Any vertex f with degree two that remains after all self-loops are broken with Rule 1 will have one incoming and one outgoing edge and no self loop (if f had only incoming or outgoing arcs, these arcs would not be part of a strongly connected component). If at least one of its neighbours is not blacked out, we shortcut f by removing it and connecting its neighbours to each other in correct direction.



Note that the application of this rule can make new self-loops. Additionally, if f is blacked out, the rule can be applied whether or not its neighbours are blacked out, but we do not check for this case as removing a blacked out vertex does not decrease the amount vertices we have to branch on in the second phase.

3. **Cut Black Cycle** If there is a cycle of only blacked out vertices and one non-blacked out vertex f, then f must be a loop breaker. Mark f as such and remove it and its incident arcs from the working graph. (In the following example all vertices could have other incoming and outgoing arcs.)



4. **Remove Dangling Vertices** Any vertices with degree zero can be safely removed from the graph.

During this process the input can break apart in multiple strongly connected components, when this happens we remove the arcs that go between the components to allow Rules 2 and 4 to trigger.

These rules may seem arbitrary and undoubtedly many more can be made. We have chosen these because they represent situations we encountered often, and for many instances, the rules gave significant reductions in the size of the graph, thus allowing to find optimal FVS's for instances that would otherwise be intractable.

3.2 Phase 2: Branch and bound

We carry out Rules 1-4 exhaustively. When no rule can be applied anymore we then branch on the remaining non-blacked out vertices using a standard branch and bound approach. When we encounter a non-blacked vertex; we first branch on the case that this vertex *is* a loop breaker, and then branch on the case where the vertex is *not* a loop breaker. This order of branching helps to have a faster decrease of the bound, and thus speeds up the branch and bound process. We test if loops remain after each branch using a depth first search.

This phase dominates the rest of the algorithm by far and has a worst case running time of $O^*(2^{|V|})$. To limit the compile-time we only branched on components with less than 40 non-blacked out vertices. 40 branch-vertices would take 2^{40} branch steps in the worst case, but these components were just tractable in our experiments (See Section 4); likely because the bound lowers quickly and not all branches had to be explored. In a production compiler we would recommend a lower limit to err on the side of caution. We encountered very few components with more than 40 vertices. In components that exceed this limit we break on all remaining vertices, which still tends to outperform the old algorithm since at that point the preprocessing rules have already removed a lot of vertices.

3.3 Possible weights

The priorities in GHC give a strict global order of importance: a variable with priority 2 will never be a loop breaker while a variable of priority 1 is still available somewhere in the strongly connected component, even if that many variables of priority 1 have to be broken only to spare the one variable with priority 2. In reality it is likely there is some break-even point where it is more beneficial to make the one priority 2 variable the loop breaker and inline the many lower priority variables. Alternatively, it might be sensible to distinguish different inlining-desirabilities even between variables of the same priority.

The branch and bound algorithm can be modified to allow for each variable to be assigned an arbitrary real score or weight. This allows the compiler to attach more detailed information to each vertex about how beneficial it would be to inline. For example by taking information from a profile of a typical run of the program to note how often each arc is travelled and the size of the expression to be inlined to give a very accurate figure on the cost and benefit of inlining a certain arc. As far as we know there is no automatic profile guided optimisation in GHC as of yet even though it has shown to yield significant

speed gains in imperative languages[12].

We have not tried to gather such richer weights and this feature remains untested.

4 Benchmark results

In Section 3 we have described an algorithm to compute minimum size FVS's in directed graphs and its role for choosing loop breakers. We have tested the heuristic from the GHC compiler and our new algorithm on the nofib benchmark suite [17] to compare these algorithms with respect to the following practical considerations:

- How often do complex recursive nests of functions actually occur?
- Does the current heuristic really perform badly in such cases?
- How much time does it take to run the exact algorithm?
- Do the binaries compiled with the exact algorithm actually run faster?

The tests were run on a modern computer with an Intel Core i5-2500 quad-core CPU with a maximum frequency of 3.7 GHz and 4 GB RAM running a 64 bit version of Windows 7. We modified GHC 7.0.2 to use our algorithm and compared that to the official 7.0.2 release, using the -O1 flag for both.

4.1 The strongly connected components of the nofib suite

We have extracted the reference graph of all programs in the *real* subset of the suite[3] from GHC after the front-end and right before the core-language optimisation takes place. At that point no non-trivial inlining has taken place. This is the graph as the inliner will first encounter it. To give an overview of the call graphs we encountered we have collected some statistics: the entire collection of 30 programs combined has 30491 vertices and 98218 arcs. These consist of 29066 strongly connected components, 98% of which are singletons (either non-recursive or only self-recursive, in both cases trivial loop breaker-wise). Of the 2% of non-trivial components 52% are dictionary-nests containing one or more vertices with priority 9. The largest strongly connected component in this set contains 85 mutually-recursive vertices There are 24 strongly connected components with more than 10 vertices.

[3]The nofib suite is divided into three distinct subsets: imaginary, spectral and real. The last one is considered most representative of real programs and is therefore the main focus of out comparison.

We show a few examples of interesting call graph components in Figure 3, 4 and 5. The number in the vertices show the priority assigned by GHC, the text is a simple version of the variable name (which is generated by the front end in some cases). The vertices with a thick outline are chosen by the GHC heuristic, the vertices with a grey background are chosen by our algorithm. These are all examples where the heuristic marks all or most of the component as loop breaker, whereas only a few vertices are really necessary. Figure 5 shows the priority system at work: all vertices of priority 0 are loop breakers but the vertices of priority 5 are remain available for inlining.

The old heuristic chooses 2085 loop breakers in the entire real subset of the suite, while the exact algorithm only needs 1754 to break all loops, which gives a decrease of 15.9%.

The complexity of the reference graphs varies from program to program: 10 of the 30 programs have no non-trivial strongly connected components at all; some have only a couple of components that are handled adequately by the old heuristic. Some of the programs however have rather large components where the exact algorithm outperforms the heuristic by an order of magnitude. The larger components tend to be part of parsers or other tree-walks, in for example compilers this could be frequently used code that could benefit from some extra inlining.

4.2 Compiling speed

An improvement of the runtime performance should not come at unreasonable extra costs at compile-time. Calculating the optimum Feedback Vertex Set of the call graphs using our algorithm takes 2.5 seconds on our machine. Only one

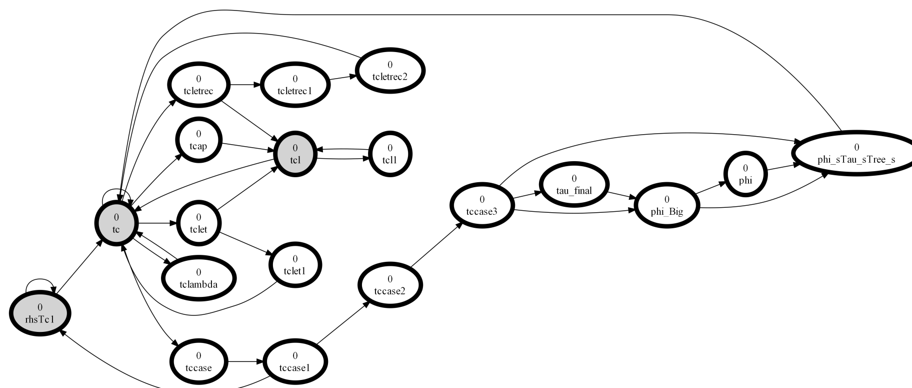


Figure 3: A strongly connected component in real/anna.

Figure 4: A strongly connected component in real/bspt

Figure 5: A strongly connected component in real/hpg. The nodes that are not chosen as loop breaker by the heuristic all have priority 5.

component was too large for the branching threshold and was approximated as described in Section 3.2. As a reference, compiling the same set of programs with a stock version of GHC with the -O1 option takes 59 seconds on the same machine.

We have no doubt that performance can be improved further by adding more preprocessing or tweaking the branching strategy but we felt we have sufficiently proven the feasibility of this approach. Note that the given time only reflects one pass of the original graph, in practise the compiler will do multiple passes.

4.3 Nofib results

Table 1 shows the results of the comparison between the two algorithms. The *Size* column shows the size difference of the compiled binaries, which is virtually imperceptible in all cases. This is probably due to the size of the programs being so small compared to the runtime system which is compiled into each program. The binaries do differ. *Allocs* shows the allocations done during runtime, *Runtime* is the wall clock time and *TotalMem* shows the memory used during the test.

The benchmarks show a very modest runtime improvement: in most cases changing the loop breaker algorithm only changes the runtime of the compiled programs very little, sometimes even increasing it slightly. Only 3 programs improve by more than 2% in the time required to complete the tests. There are several explanations possible:

- In the bigger picture, loop breaker choice is not a very significant factor for the total runtime. This would mean that even though the inliner has more options to choose from, it does not benefit from them. (We do see that it does make different optimisations.)
- The benchmark programs either are not as loop-breaker-sensitive as could be or the improved components are not covered much by the tests.
- Many of the tests take very little time even without optimisations and thus reliable measurements are difficult. Most take less than 500 ms. In fact 48 of the 91 tests in the entire suite take less than 200 ms and are ignored in the aggregates.

Program	Size	Allocs	Runtime	TotalMem
ansi	+0.0%	+0.0%	+0.3%	+0.0%
atom	+0.0%	+0.0%	+0.9%	+0.0%
bernouilli	+0.0%	+0.0%	+0.9%	+0.0%
boyer	+0.0%	-4.9%	-6.4%	+0.0%
cacheprof	+0.0%	+0.0%	+1.0%	+0.0%
calendar	+0.0%	+0.0%	+0.1%	+0.0%
circsim	+0.0%	+0.0%	-0.6%	+0.0%
clausify	+0.0%	+0.0%	-1.7%	+0.0%
comp_lab_zift	+0.0%	+0.0%	+0.6%	+0.0%
constraints	+0.0%	+0.0%	-0.0%	+0.0%
cryptarithm1	+0.0%	+0.0%	-1.2%	+0.0%
event	+0.0%	+0.0%	-0.7%	+0.0%
exp3_8	+0.0%	+0.0%	+0.9%	+0.0%
fft	+0.0%	+0.0%	-2.9%	+0.0%
fft2	+0.0%	+0.0%	-0.4%	+0.0%
fibheaps	+0.0%	+0.0%	-1.5%	+0.0%
fulsom	+0.0%	+0.0%	+1.1%	+0.0%
gen_regexps	+0.0%	+0.0%	+0.1%	+0.0%
genfft	+0.0%	+0.0%	-0.9%	+0.0%
ida	+0.0%	+0.0%	-0.8%	+0.0%
integer	+0.0%	+0.0%	-0.3%	+0.0%
integrate	+0.0%	+0.0%	+0.8%	+0.0%
knights	+0.0%	+0.0%	-0.3%	+0.0%
lcss	+0.0%	+0.0%	+0.3%	+0.0%
multiplier	+0.0%	+0.0%	-0.2%	+0.0%
para	+0.0%	+0.0%	-0.3%	+0.0%
paraffins	+0.0%	+0.0%	+0.7%	+0.0%
primes	+0.0%	+0.0%	+0.2%	+0.0%
primetest	+0.0%	+0.0%	+0.1%	+0.0%
queens	+0.0%	+0.0%	-0.2%	+0.0%
rewrite	+0.0%	+0.0%	-1.1%	+0.0%
rfib	+0.0%	+0.0%	-0.1%	+0.0%
sched	+0.0%	+0.0%	-0.3%	+0.0%
scs	+0.0%	+0.0%	-0.6%	+0.0%
solid	+0.1%	-0.4%	+1.2%	+0.0%
sphere	+0.0%	+0.0%	-2.1%	+0.0%
tak	+0.0%	+0.0%	+0.6%	+0.0%
transform	+0.0%	+0.0%	-0.8%	+0.0%
typecheck	+0.0%	+0.0%	+0.4%	+0.0%
wang	+0.0%	+0.0%	-0.4%	+0.0%
wave4main	+0.0%	+0.0%	+0.1%	+0.0%
wheel-sieve1	+0.0%	+0.0%	-0.1%	+0.0%
wheel-sieve2	+0.0%	+0.0%	-1.8%	+0.0%
Min	**-0.0%**	**-4.9%**	**-6.4%**	**+0.0%**
Max	**+0.1%**	**+0.0%**	**+1.2%**	**+0.0%**
Geometric Mean	**+0.0%**	**-0.1%**	**-0.4%**	**-0.0%**

Table 1: Results of the comparison between the GHC heuristic and our algorithm as compiled by the nofib-analyse utility. Based on averages of 25 runs for each test.

5 A new approach considering individual references

It is possible to break the loops at arcs instead of vertices of the reference graph. For the inliner this would mean making individual function *calls* loop breakers instead of entire functions. Doing it this way gives a more accurate portrayal of the costs of making loop breakers.

Consider the example in Figure 6; in this example of a simple loop f has many incoming arcs outside of the strongly connected component. Here f could be the exposed function of a library that is implemented recursively. The inliner can choose any of the four functions as the loop breaker, each would break all the loops of this component, however it would be unfortunate if the inliner would choose f as the loop breaker as – even though it is just one vertex – this prevents many potential inlinings. If the inliner would consider individual arcs as possible loop breakers we could for example mark the call from p to f as a loop breaker and leave f's other references available for inlining.

Looking at the loop breaker problem from the perspective of the arcs has several advantages:

- It more closely represents the actual costs of choosing a loop breaker. Moreover arcs not on a cycle will not be broken without need as illustrated in the example discussed.
- It is a generalisation of the original approach with vertices: a loop breaker is just an implicit removal of all its incoming arcs, so a good solution for vertex-loop-breakers will guarantee an at least equally good solution for arc-loop-breakers.
- As this approach is call-site aware, it allows for more finely grained priorities that take into account not only the function to be inlined but also where it will be inlined.

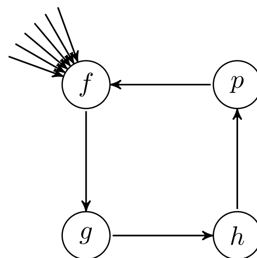


Figure 6: A small reference graph component with some context from outside of the strongly connected component.

For example, in Haskell the third advantage could be applied to case-elimination: a function that is a constructor application gets a higher priority because it is desirable to inline it because it might end up in a case-statement which can then be dissolved. Using the new approach we could give a call *from a case-statement* a higher priority for the same reason.

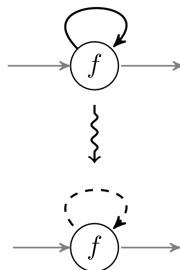
This also eliminates the need for the special priority for dictionary tuples as they might have seemed like a good choice for loop breaker for the DBFVS – breaking many cycles with just one vertex – they no longer do when you are confronted with the fact that they have many incoming arcs, are a constructor application and are called from a case-statement.

5.1 Algorithm

The equivalent graph theoretical problem is aptly named the Blackout Feedback Arc Set problem (BFAS). The directed-qualifier is left out here since it is implicit in the term arc. The BFAS problem can be defined as follows: given a directed graph $G = (V, E)$ and a set $D \subseteq E$ find a minimum size set $F \subseteq E - D$ such that each cycle in G contains an arc in F. The arcs in D are called blacked out. In the literature this problem is considered almost equivalent to the DBFVS problem; direct polynomial reductions are shown in [8].

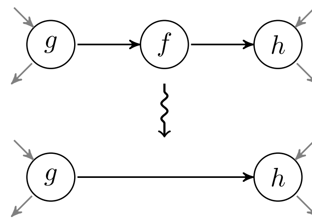
The algorithm we use is exactly the same as for the vertices case switching out arcs for vertices, with the exception of somewhat different preprocessing rules. In the following illustrations a dashed arc signifies an arc that is selected as a loop breaker.

1. **Break Self-loop** If a vertex f has a self-loop, i.e. there is an arc $a = (f, f)$, then a must be a loop breaker. Mark a as such and remove it from the working graph.



2. **Shortcut Degree Two** Any vertex f with degree two that remains after all self-loops are broken with Rule 1 will have one incoming and one outgoing edge and no self loop (if f had only incoming or outgoing arcs, these

arcs would not be part of a strongly connected component). Let its incoming neighbour be g and outgoing neighbour be h. We shortcut f by removing it and its incident arcs (g, f) and (f, h) and adding the arc (g, h). If and only if at most one of the two arcs (g, f) and (f, h) was blacked out, the new arc (g, h) should not be blacked out.



Note again that the application of this rule can make new self-loops.

These are fewer and less effective rules than with the vertices case. A self-loop in the vertices case would for instance mean we could remove the vertex and all its other edges from the component, usually triggering more rules. Again, more rules can be developed.

5.2 Practical results

GHC does not break loops at the arcs. However, we can still compare its performance to the optimum BFAS because a loop breaker vertex is in essence an implicit breaking of all its incoming arcs. This way GHC uses 5800 arc loop breakers (from 2085 vertices) on the real subset of the nofib suite. Our algorithm only needs 3250 arc loop breakers, which is a reduction of 44%. It encountered 7 strongly connected components with more than 40 branch arcs which it only approximated, the largest one consisted of 248 branch arcs (in the veritas program). Calculating the BFAS on the entire set took 9.5 seconds, which is slower than the DFVS algorithm, but we have spent less time optimising this version.

A 44% reduction in loop breakers seems like a lot but it is hard to estimate practical benefits from only this number. For example a directly recursive function can now be inlined (but not in itself) and its incoming arcs will no longer be loop breakers, but inlining such a function would result in unrolling one iteration of the top of the loop; it is not obvious that this would yield any measurable improvement in runtime.

We have not implemented this approach of breaking loops at the arcs into GHC to test its practical effectiveness because that would require a significant rearchitecturing of the loop breaker framework in the code base. A practical approach to this problem could be to modify the program to make each variable reference indirect by adding a new trivial intermediate variable and then only considering those new trivial variables when picking loop breakers. From the perspective of the call graph this looks like inserting a dummy vertex between each arc or more formally: make a graph $G' = (V', E')$ where $V' = V \cup E$ and $(f, g) \in E \longleftrightarrow \{(f, (f, g)), ((f, g), g)\} \in E'$ and all vertices $V \subset V'$ are blacked out. The new variables effectively model the references of the original program one to one and thereby arcs can be broken using the existing infrastructure. This does have its downside as it generates indirection, but the inliner will then likely inline all the dummy variables that are not loop breakers as they are trivial expressions thereby removing some of the run time costs of the indirection.

6 Further ideas

As with most research a lot of ideas we have pursued did not make it into the final result. We document the most interesting side-paths here.

6.1 Improving the exact algorithm

We consider it shown that our approach is feasible even though it does not yield immediate improvements in compiled code. Still, there are ways not mentioned in the other sections to improve the performance of the algorithm further.

- **Improving the cyclicity-test** During the branch and bound phase most of the time is spent testing whether the graph is currently cyclic; any improvement to that process will make the whole algorithm significantly faster. Since the graph only changes minimally (a couple of edges are removed) on each branch, it is tempting to try to reuse some information from previous iterations. One heuristic to incrementally check cyclicity, when starting out with a acyclic graph, is to sort the graph topographically. When an edge is added to the graph that doesn't make a cycle, there is a good chance the topographical ordering will still be valid (the new edge will point forward). Only in the case that the edge points backwards do we have to check if we can find another topographical ordering. Unfortunately, in our current implementation, we don't start with an acyclic

graph and add edges, instead we start with a cyclic graph and remove edges.

- **Incremental reuse of feedback sets** The compiler typically does several iterations of the optimisation phase; in each step changing the code further. This could be used to our benefit as that makes it likely we will encounter the same or similar components again, and we can perhaps somehow reuse information from previous runs. An idea is to save the feedback set and use it as an initial solution the next time we encounter the component, perhaps adding possible new vertices to the set. Thereby starting out with a relatively tight bound. How this would interact with other optimisations is not obvious, and coordinating the information storage between optimiser runs is challenging.

6.2 Orthogonal inlining dilemma's

Breaking loops to ensure termination is only a small part of the inliner which itself is only a part of a system of optimisation techniques.

A small idea comes in the form of *loop unrolling*. The loop breaker algorithm is based on the assumption that it is bad to inline a self-recursive function. Generating an infinitely large function is obviously bad, but inlining a recursive function several times can be very beneficial: similar to loop unrolling in imperative languages, very tight loops might be *unrolled* to save a big percentage of the jumps back to the start of the loop. This is also likely to expose new opportunities for optimization. So some limited inlining of loop breakers might be desirable.

6.2.1 What to inline

The product of the loop breaker algorithm is essentially a list of variable-references that are safe to inline. The inliner then faces the choice of what references to choose to inline. A typical way to evaluate inliner-effectiveness is by comparing the run-time decrease to the code size increase of a suite of benchmark programs (whether or not the inliner is not explicitly optimized toward this cost-benefit analysis). Since it is usually impossible to predict the exact memory (cache) situation of the target machine and the runtime behaviour of the program, so no hard bound is given as acceptable code size, but trying to get as fast a program as possible without letting the code grow by more than $10 - 20\%$ is a common way to try to tune the inliner.

We can make targets such as these explicit in the inlining model by assigning each arc not only a benefit-value (determined by the likeliness for more optimization and possibly a profile) but also a cost (how much will it cost to inline this function, mostly determined by the size of the right hand side of its definition). The question then becomes: select the arcs with maximum benefit for which the cost value does not exceed some quotum (similar to the Knapsack problem, which also NP-complete albeit weakly). This describes the inlining problem very well and has great potential for benefit compared to the current inliner as even large function might sometimes be worth inlining. Additionally, this gives the compiler great flexibility to make more informed decisions when more information is available by new kinds of program analysis by weighing the different factors in the cost- and benefit-weights.

However, we have been unable to make a reasonable concrete model using this approach. The Knapsack problem is insufficient to model the inliner situation completely:

- A function that isn't top level and is only called once can be inlined for free, the cost doesn't count.
- This also means that when all but one of the call sites are inlined, the last call site can be inlined for free.
- A function call that is inlined means it copies all outgoing arcs of the function it copies, this increases the size of the graph and therefore gives more opportunities to inline.

6.2.2 Supercompilation

Recently the field of Haskell optimization has seen multiple papers on the subject of supercompiling [5, 16] which can be seen as a generalization of inlining and several other optimization techniques. A supercompiler tries to evaluate as much of a program as possible at compile time, including dereferencing functions (inlining). This has very promising speed gains in benchmarks but has its own problems:

- Since the compiler partially evaluates the program, care must be taken to still guarantee termination;
- Even when termination is safe, some worst-case run time guarantees on the supercompiler are needed;
- Supercompiler, like inlining, is prone to code bloat which can massively increase code size and thereby cause performance issues as well.

Supercompiling is obviously a very interesting subject with its own algorithmic challenges but it's much more complex and harder to apply in practice than inlining. Breaking every loop seems too coarse for this approach as the most interesting cases for supercompilation involve evaluating recursive functions. However, termination guarantees in this field are still in their infancy: while termination is proven, generally no bounds are given and in fact the compiler often takes undesirably long to compile large programs. There is still room for improvement.

7 Conclusions

We have shown that the GHC loop breaker choosing heuristic commonly chooses more loop breakers than necessary. Our algorithm can compute the exact minimal set in most cases and do so within reasonable time. In the real subset of the nofib benchmark suite our algorithm needs 15.9% fewer loop breakers. Programs with parsers and complicated tree-walks receive the largest benefit. Unfortunately this results in only a modest 0.4% run time improvement in the benchmarks. The poor performance could be explained by problems with the benchmark suite.

The algorithm has an untested advantage in that it could be used in more advanced scenarios as it allows using for example profile guided optimisation.

We have also presented a promising alternative approach where we consider individual variable references as loop breakers instead of the variables themselves. This represents the cost of a loop breaker more accurately, allows for more inlining and gives us the opportunity take the call-site into account when estimating the desirability of a potential inlining. Our offline tests show this approach requires 44% fewer loop breakers than GHC's current implementation, but we have not tested the practical benefits.

7.1 Acknowledgements

My gratitude to Hans Bodlaender and Atze Dijkstra – who attentively supervised all this work – exceeds my ability to enunciate. Both experts in their fields, they dedicated many hours to help me and my research; not only with wisdom, but also with great encouragement and inspiration. Thank you.

Jeroen Fokker deserves special thanks for originally suggesting loop breaker selection as an interesting graph problem. My thanks also go out to Thomas van Dijk and José Pedro Magalhães for our many fruitful discussions, this thesis

would have been incomplete without them; and Simon Peyton Jones for his expert help and patience, specifically for suggesting the way to implement the FAS approach with dummy variables.

References

[1] T. I. Axenovich, I. V. Zorkoltseva, F. Liu, A. V. Kirichenko, and Y. S. Aulchenko. Breaking loops in large complex pedigrees. *Human Heredity*, 65(2):57–65, 2008.

[2] Ann Becker, Reuven Bar-Yehuda, and Dan Geiger. Random algorithms for the loop cutset problem. In *Proceedings of the Fifteenth conference on Uncertainty in artificial intelligence*, pages 49–56, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[3] Ann Becker, Dan Geiger, and Alejandro A. Schäffer. Automatic Selection of Loop Breakers for Genetic Linkage Analysis. *Human Heredity*, 48(1):49–60, 1998.

[4] Hans L. Bodlaender and Thomas C. van Dijk. A cubic kernel for Feedback Vertex Set and Loop Cutset. *Theoretical Aspects of Computer Science*, 46(3):566–597, February 2010.

[5] Maximilian Bolingbroke and Simon Peyton Jones. Supercompilation by evaluation. In *Proceedings of the third ACM Haskell symposium on Haskell*, Haskell '10, pages 135–146, New York, NY, USA, 2010. ACM.

[6] Jianer Chen, Yang Liu, Songjian Lu, Barry O'sullivan, and Igor Razgon. A Fixed-Parameter Algorithm for the Directed Feedback Vertex Set. *Journal of the ACM*, 55(5):21:1–21:19, November 2008.

[7] Rod G. Downey, Michael R. Fellows, Rolf Niedermeier, and Peter Rossmanith. *Parameterized Complexity*. 1998.

[8] Guy Even, Joseph Naor, Baruch Schieber, and Madhu Sudan. Approximating Minimum Feedback Sets and Multi-Cuts in Directed Graphs. *Algorithmica*, 20:151–174, 1998.

[9] Paola Festa, Panos M. Pardalos, Mauricio, and Mauricio G.C. Resende. Feedback Set Problems. In *Handbook of Combinatorial Optimization*, pages 209–258. Kluwer Academic Publishers, 1999.

[10] Fedor V. Fomin, Serge Gaspers, and Artem V. Pyatkin. Finding a Minimum Feedback Vertex Set in Time $O(1.7548^n)$. In *Proceedings of the 2nd International Workshop on Parameterized and Exact Computation*, page 184. Springer, 2006.

[11] Jiong Guo. *Algorithm Design Techniques for Parameterized Graph Modification Problems*. PhD thesis, 2006.

[12] Rajiv Gupta, Eduard Mehofer, and Youtao Zhang. Profile-Guided Compiler Optimizations. In *The Compiler Design Handbook*, pages 143–174. 2002.

[13] Falk Hüffner, Rolf Niedermeier, and Sebastian Wernicke. Developing Fixed-Parameter Algorithms to Solve Combinatorially Explosive Biological Problems. In *Bioinformatics*, volume 453 of *Methods in Molecular Biology Series*, pages 395–421. Humana Press, 2007.

[14] Simon Peyton Jones and Simon Marlow. Secrets of the Glasgow Haskell Compiler inliner. *Journal of Functional Programming*, 12(4):393–434, July 2002.

[15] Richard M Karp. Reducibility among combinatorial problems. *Complexity of Computer Computations*, 40(4):85–103, 1972.

[16] Neil Mitchell. Rethinking supercompilation. *SIGPLAN Not.*, 45:309–320, September 2010.

[17] Will Partain. The nofib Benchmark Suite of Haskell Programs. In *Proceedings of the 1992 Glasgow Workshop on Functional Programming*, pages 195–202, London, UK, UK, 1993. Springer-Verlag.

[18] Igor Razgon. Computing Minimum Directed Feedback Vertex Set in $O(1.9977^n)$. In *Italian Conference on Theoretical Computer Science*, pages 70–81, 2007.

[19] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating system concepts (7. ed.)*. Wiley, 2005.

[20] Stéphan Thomassé. A $4k^2$ Kernel for Feedback Vertex Set. *ACM Transactions on Algorithms*, 6(2), 2010.

[21] Z. G. Vitezica, M. Mongeau, E. Manfredi, and J. M. Elsen. Selecting Loop Breakers in General Pedigrees. *Human Heredity*, 57(1):1–9, 2004.