UvA-DARE (Digital Academic Repository)

Strictness optimization for graph reduction machines (why id might not be strict)

Beemster, M.

DOI

[10.1145/186025.186040](https://doi.org/10.1145/186025.186040)

Publication date

1994

Published in

ACM Transactions on Programming Languages and Systems

[Link to publication](#)

Citation for published version (APA):

Beemster, M. (1994). Strictness optimization for graph reduction machines (why id might not be strict). *ACM Transactions on Programming Languages and Systems*, 16(5), 1449-1466. <https://doi.org/10.1145/186025.186040>

General rights

It is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), other than for strictly personal, individual use, unless the work is under an open content license (like Creative Commons).

Disclaimer/Complaints regulations

If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: <https://uba.uva.nl/en/contact>, or a letter to: Library of the University of Amsterdam, Secretariat, Singel 425, 1012 WP Amsterdam, The Netherlands. You will be contacted as soon as possible.

Strictness Optimization for Graph Reduction Machines (Why it Might Not Be Strict)

MARCEL BEEMSTER
University of Amsterdam

Strictness optimizations in the implementation of lazy functional languages are not always valid. In nonoptimized graph reduction, evaluation always takes place at the request of case analysis or a primitive operation. Hence, the result of a reduction is always a data value and never a function. This implies that in an implementation no argument satisfaction check is required. But in the presence of strict arguments, “premature” reduction may take place outside the scope of a case or primitive operation. This causes problems in graph reducers that use an *aggressive take*. Two solutions are presented, one based on a run-time argument satisfaction check, the other on a weakened strictness analyzer. Experimental results are used to compare the two solutions and show that the cost of the aggressive take can be arbitrarily high for specific programs. The experimental results enable a trade-off to be made by the reduction machine designer.

Categories and Subject Descriptors: D.2.8 [Software Engineering]: Metrics—*performance measures*; D.3.2 [Programming Languages]: Language Classifications—*applicative languages*; D.3.4 [Programming Languages]: Processors—*code generation; compilers; optimization; run-time environments*; F.1.1 [Computation by Abstract Devices]: Models of Computation—*relations among models*

General Terms: Languages, Performance

Additional Key Words and Phrases: Graph reduction, lazy evaluation, strictness analysis

1. INTRODUCTION

An important research topic in lazy functional languages is implementational efficiency. An aspect of implementation efficiency is the amenability of the reduction process to compiler optimizations. No implementation of a lazy functional language will perform well if the compiler does not implement a number of optimizations specific to graph reduction. Many optimizations share one goal: they aim to reduce the use of *laziness* in the implementation.

Laziness combines two aspects: normal-order reduction and sharing. Normal-order reduction specifies that the leftmost outermost redex should be reduced first. Sharing assures that a redex is reduced at most once. Subsequent uses of the same redex should be able to obtain its reduced value

Author's address: Faculty of Mathematics and Computer Science, University of Amsterdam, Kruislaan 403, 1098 SJ Amsterdam, The Netherlands; email: beemster@fwi.uva.nl.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1994 ACM 0164-0925/94/0900-1449\$03.50

ACM Transactions on Programming Languages and Systems, Vol. 16, No. 5, September 1994, Pages 1449–1466.

directly. In practice, leftmost outermost execution requires that the evaluations of arguments to functions be delayed until needed by building suspensions for them.

Lazy languages implemented with pure graph reduction are not nearly as efficient as applicative-order languages and implementations. The mechanisms to delay the evaluation of an argument are costly. A suspension must always be built, even when the argument is not evaluated at all. In addition to the normal objects that an applicative-order implementation handles, a lazy implementation must also be able to handle unevaluated expressions. All of this costs valuable run time.

The answer to the efficiency problem lies in improved compilation techniques. Some of these techniques are specific to the graph reduction model chosen; others are generally applicable to most graph reduction models. For an overview see Peyton Jones [1987]. An important general optimization technique is strictness optimization [Clark and Peyton Jones 1985; Davis and Wadler 1989; Nöcker 1990]. The idea behind strictness optimization is that the compiler can often find out that an argument will be required even though normal-order reduction is used. Instead of building a suspension for the argument, it immediately evaluates it.

Two general classes of compiled graph reduction implementations can be distinguished, the G-machine model and the Spineless model. The *spine* of the graph is used to keep track of the next leftmost outermost expression to be reduced. In the pure *G-machine* model [Johnsson 1984] the spine is first built as a part of the graph and then copied onto the evaluation stack via a series of UNWIND steps. Optimizations can remove many, but not all, of these explicit spine manipulations. The second class, the *Spineless reduction* model, tries to avoid the costly UNWINDs altogether by directly pushing arguments onto a stack. This saves copying of the operations and the building of nodes in the graph. Two implementations of the G-machine model are the Chalmers LML implementation [Augustsson and Johnsson 1989] and the FAST compiler [Hartel et al. 1994]. Three different implementations of the spineless model are the Three-Instruction Machine (TIM) [Fairbairn and Wray 1987], the spineless G-Machine (SGM) [Burn et al. 1988], and the Spineless Tagless G-Machine (STG-M) [Peyton Jones 1992]. The STG-M is the basic machinery behind the Glasgow HASKELL compiler [The Grasp Team 1992].

In this paper the relation between these graph reduction models and the generally applicable technique of strictness optimization is studied. It will be shown that neither the TIM nor the STG-M can straightforwardly handle the result of global strictness analysis. In particular, they cannot handle the strictness of argument x in the function definition $\text{id } x = x$, which the SGM can handle without problems. Optimizing the G-machine model by removing the argument satisfaction check would make it incapable of using this strictness information as well. This is further discussed in Section 4.1.

Section 2 explains a typical example of graph reduction in the spineless reduction model. It shows that using strictness information may lead to a problem in two of the three implementations of spineless reduction.

Sections 3 and 4 give a more general description of where a mismatch between the use of strictness information and the underlying reduction machine may occur, and explores two solutions. One of the solutions is based on a modification of the reduction machine; the other weakens the strictness analyzer.

Section 5 looks briefly at related work, and performance implications are discussed in Section 6. Both solutions are investigated using execution statistics of relevant functional programs. We show that weakening the strictness analyzer will in general cases not lead to reduced performance, although for specific programs the loss can be high.

2. INTRODUCING GRAPH REDUCTION AND STRICTNESS ANALYSIS

The use of strictness information can change the evaluation order of the expressions to be reduced. As a running example of graph reduction and the effects of strictness analysis, we use the expression `expr`, together with the function definition for `swap`:

```
expr = swap 3 (add 5)
swap g f = f g
```

Many compiler optimizations might be applied to this example but, except for strictness optimization, they will be ignored. In pure graph reduction, as opposed to an implementation as discussed below, the evaluation of the expression proceeds in three steps:

```
expr = swap 3 (add 5)
expr = (add 5) 3
expr = 8
```

The same process is shown in graphical format in Figure 1. In Figure 1, (a) is the graph representation of the expression to be evaluated. The oval marked `eval` denotes the graph reducer, which is given a pointer to the graph to be reduced. Function application is represented by an *apply*-node, marked `@` in the figure. In an implementation, an `@`-node is actually a heap-allocated data structure and contains a left (function part) and right (argument part) pointer to further expressions. The definition of the `swap` function is shown in Figure 1(z). It states that an expression that matches the left-hand side of the equality, where *f* and *g* are variables to be bound in the matching process, can be replaced by the right-hand side. Such replacement is exactly what happens going from Figure 1(a) to Figure 1(b). The normal-order reduction strategy of lazy functional languages requires that the evaluator reduces the leftmost outermost reducible expression first. In the case of Figure 1(a), `eval` scans left along the `@`-nodes and finds `swap`. Looking up the definition of `swap`, the match is performed, and *f* and *g* are bound to the respective subexpressions 3 and (add 5). The rewrite is performed, and the result is Figure 1(b).

It is important to note that lazy graph reduction (as opposed to plain text-based rewrite systems) requires the results of previous computations to be shared with future work. This means that no expression should be

Fig. 1. Basic steps in graph reduction of `swap 3 (add 5)`.

evaluated twice if it is shared in the graph. In the current example, the `@`-node marked with a star (*) may have other pointers to it besides the one handed to `eval`. These other pointers to the expression must also see the effects of the current reduction. Hence, the `@`-node marked with a star must actually be the *same* node in an implementation. In the transition from Figure 1(a) to Figure 1(b), the rewrite actually *overwrites* that (possibly shared) node with its new value. This overwriting is also called *update*.

In the transition from Figure 1(b) to Figure 1(c) something similar happens, but with a twist. From Figure 1(b), the evaluator again finds the leftmost outermost reducible expression, it sees that the function `add` is applied to two arguments (as required by `add`), and it binds the two arguments to variables. At this point no rewrite happens because `add` is a primitive function, and is not defined in terms of rewriting. Control is now handed to the `add` function, which first has to make sure that both of its arguments are really numbers and not unevaluated expressions. In order to do so, `add` calls the evaluator recursively to reduce each of the arguments. If the expression is correctly type-checked, which is assumed, the results of these two recursive calls is that the two variables are now bound to numbers. In the example, the subexpressions to be evaluated cannot be further reduced, so `eval` can immediately return. After this, the addition itself is performed, and the marked `@`-node is overwritten with the result, 8.

The evaluator then finds the expression in Figure 1(c), which cannot be reduced further.

2.1 Looking at an Implementation

The explanation of the `swap` example by pure graph reduction will serve as a basis for the understanding of the reduction by spineless implementations.

The spineless reduction strategy presented below is an abstraction of three different actual implementations of spineless graph reduction. These are the TIM, the SGM, and the STG-M. All of these differ to some extent from the presentation below, but the key ideas are preserved.

A fundamental issue in spineless reduction is the handling of sharing in combination with spinelessness. Spinelessness tries to improve on the G-machine model by keeping candidate arguments on the stack instead of building an explicit graph representation. Not having such an explicit graph makes it difficult to figure out how redexes are shared. To handle sharing and updating correctly, the spineless model uses update markers, which are also present on the stack. An update marker contains a reference to the node to be updated. Update markers are pushed on the stack when a node is entered that needs to be updated with the result of its evaluation. Update markers are removed again when a function requires more arguments than are available below the first update marker on the stack.

The swap example running on a spineless machine is shown in Figure 2. The state of the machine at the start of the evaluation of the expression is shown in Figure 2(a). The stack contains a return address (Cont.), which is pushed by the code that requires the expression to be evaluated. Next on the stack there is an update marker, which points to some node in the graph that must be updated with the result of the evaluation. Were this update marker not there, the update would not be needed, and the evaluation could proceed unchanged. At the start of the evaluation all potential arguments are already on the stack, and there is no explicit spine. In this example the TIM-like pseudoinstruction Enter is used to indicate that the machine is to enter the top of the stack element. In actual implementations the current point of control will be swap, and swap will not be explicitly on the stack. For clarity the example is shown like this.

From Figure 2(a), swap is entered. It checks to see if there are two arguments available before the next update marker. This check can be made inexpensive by maintaining a list of update markers with the start of the list pointing to the update marker closest to the top. Another option is to keep the update markers on a separate stack. In this case the arguments are available, and swap proceeds by popping the two arguments from the stack and binding them to local variables. Then swap builds the representation for the expression (f g) by pushing the two arguments in the right order onto the stack. Swap has now finished and continues by entering the top element of the stack. This is the state in Figure 2(b). Note that swap did not have to build an explicit spine for the result expression, and that it did not need a new update marker because it did not introduce additional sharing.

Now the top element of the stack is entered. This is a representation of function application. To evaluate this, the arguments of the function application must be pushed on the stack, and the function is entered. Before this can be done, care must be taken that the result of computing this function application is shared in case there are other pointers to the node. An update marker pointing to the @-node is pushed on the stack. Then the arguments are pushed on the stack, and the machine is in the state in Figure 2(c). Note

Fig. 2. Graph reduction on a spineless machine.

that in an actual implementation the compiler may recognize that the @-node cannot be further reduced, and the update marker is not needed.

Add is entered and tries to get two arguments. Only one is available below the first update marker. This means that an update must take place with the current state of computation between the marker and the top of the stack. This is an update with the expression (add 5) that is now known to be reduced. After the update, the update marker is squeezed out of the stack. The state is as in Figure 2(d). The updated @-node is the same as before, but is now marked with an *r* to note that it is reduced. Reentering of this node in the future will not require an update marker to be pushed.

Again add is entered, and now both arguments are available. Both arguments are recursively evaluated to make sure that they are reduced. After that the addition is performed, and the machine arrives in the state in Figure 2(e). This looks a bit strange because the machine tries to enter the value 8. In tagless machines such as the TIM and the STG-M this is solved by making values and data structures (for example, for list constructors) into functions that require one argument. This argument is then subsequently treated as a (return) code address to which control is transferred. The SGM treats this slightly differently, but the outcome is the same.

In this case, the value 8 tries to get its return address-argument but finds the update marker. This signals that the node pointed to by the marker must be updated with the current value on top of the stack, which is 8. The update

is performed; the marker is squeezed out of the stack; and the machine arrives in the state in Figure 2(f). From here the computation of the expression is complete, and evaluation continues at the return address.

2.2 Adding Strictness Analysis to the Implementation

A strictness analyzer will detect that the function `swap` is strict in its second argument. Formally stated:

$$\text{swap } g \perp = \perp \quad g = \perp$$

where $\perp$ means undefined. This equation states that the second argument to `swap` is really needed to get a result. If something undefined is used, then the whole result will be undefined. An implementation of graph reduction can take advantage of strictness information about a particular argument by computing it *before* the function call is performed. This implies that the building of a possibly expensive suspension for the argument can be circumvented. Evaluation order is no longer dictated by the standard leftmost outermost reduction strategy, but this does not introduce extra reduction steps. The strictness analysis ensures that the premature argument reductions would be needed anyway. In the example below, the strict evaluation occurs as if it is taken care of by the function that is receiving the arguments. This simplifies the explanation because compiler-generated code does not have to be taken into account. In a real implementation, it would be the function call environment that takes care of strict evaluation. This would enable the environment to skip the building of suspensions for the arguments. Despite this deviation from reality, the mechanisms involved are correct.

Applying the strictness of the second argument of `swap`, nothing changes visibly in the pure graph reduction of Figure 1. Instead of applying `swap` directly from Figure 1(a), the second argument, bound to `f`, will be reduced first. However, this argument is already reduced, so reduction proceeds as shown.

The spineless machines are more interesting. Starting in Figure 3(a), `swap` is first going to reduce its second argument, (add 5). It pushes onto the stack both a continuation for its work and the argument to be reduced, `f`. The state is now Figure 3(b), and the `@`-node is to be entered. At this point a similar transition occurs as from Figure 2(b) to (c). An update marker pointing to the `@`-node is pushed, and the function application is unfolded onto the stack. This is the state in Figure 3(c). As in the original Figure 2, `add` is entered, tries to find two arguments, and finds the update marker after just one argument. The update is performed, setting the *r* qualifier to signal that the node is reduced and that the marker is squeezed out.

The situation is now as in Figure 3(d). Again `add` is entered, and now there is a problem. Instead of a second argument or another update marker, there is a continuation address on the stack. This can certainly not act as a second argument, and it was not intended to. In the TIM and the STG-M, this situation cannot be handled. There is no way of ending the evaluation with a

Fig. 3. Spineless reduction with use of strictness information.

partially evaluated function. The TIM and the STG-M would, when confronted with this situation, crash. These two spineless machines cannot handle certain types of strict arguments. On the other hand, the SGM does handle this correctly. The reasons are that the SGM operates using explicit stack frames and explicit checks are made to see if there are still enough arguments in the current stack frame.

As an interesting aside, consider a similar example with `apply` instead of `swap`, where `apply` is defined as `apply f g = f g`. It can happen that this may *accidentally* run correctly on some implementations (those that keep the continuation pointers on a separate stack). When evaluating `apply (add 5) 3`, 3 is pushed on the stack, and `(add 5)` is entered in a strict fashion. This will incorrectly reduce `(add 5 3)` to 8. Note that the 3 on the stack is also used although it should not be. Then `apply` is entered, which simply enters the function on top of the stack, which happens to be 8. The result is 8 without leaving garbage on the stack.

We have shown that there are implementations of graph reduction that cannot handle the strictness of some arguments. The problem is not due to the intricate ways of handling sharing and updates in spineless implementations. As described in the following sections, the problem is of a more general nature and arises from a combination of strictness analysis and the dynamic scope of the `eval` function. Hence, it is relevant to many implementations, including G-machine-style graph reducers.

3. CORRECTLY TYPED GRAPH REDUCTION AND STRICTNESS ANALYSIS

In the implementations of modern typed lazy functional languages, the reducer function `eval` is in principle recursively called from only three places:

- `Eval` is called from built-in functions to evaluate arguments to values or data structures. In the example this was shown for `add`, which had to reduce its arguments to values.
- `Eval` is called to reduce the argument to a `case-expression`. Unlike the STG-M, this case is only to be used for scrutinizing data structures. When reduced, the `case` can check the tag of the reduced expression to see at which branch evaluation must continue. Here, the `case`-argument will reduce to a data structure.
- `Eval` is called from the top-level routine, sometimes called the printer-driver, to evaluate the main program expression to Normal Form. This is often done by considering the function `main` to be the main expression. The result required from computing the main expression can be a list of characters or something as complicated as a list of output responses [Hudak et al. 1992]. However, it must definitely be either a data structure or value, and may not contain partial applications.

The bottom line is that in a correctly typed program, *a recursive call to eval will never return with a reduced value that is a partial application*. This property is important because its implication can also be used the other way round. In a correctly typed program, when the evaluator finds it must reduce an application, the arguments needed by the application will be there by definition. The evaluator does *not* have to check if there are enough arguments available.

As a result, an implementation can be streamlined in two ways. It does not have to include an argument satisfaction stack, and the evaluator never has to return a partial application as a result. Both the TIM and the STG-M are optimized in this respect. These two machines both use an *aggressive-take*-operation. The operation of getting the required arguments for a partial application is called the *take*-operation. In the TIM, there is a special instruction devoted to this, but all implementations of graph reduction have a point where the arguments must be acquired and bound to the function arguments. Implementations that rely on the evaluator being called only to reduce to a value or a data structure, and that do not check to see if there are actually enough arguments, are said to have an aggressive take.

3.1 Implications of Adding Strictness Optimization

When adding strictness analysis to an implementation, a fourth possibility for recursive calls to `eval` is created. The evaluator may now be called to reduce arbitrarily typed arguments to functions. Since there is no typing restriction on these arguments, `eval` must be prepared to return partial applications.

This leads to the following conclusions:

- (1) Implementations of graph reduction, including G-machine-style implementations, that do not make use of strictness analysis (or restrict strictness analysis as described below), are not required to implement an argument satisfaction check.
- (2) Implementations of graph reduction that have an aggressive take, such as the TIM and the STG-M, cannot make use of unrestricted strictness informations.

4. TRADE-OFFS

It seems that to make use of strictness analysis, a graph reducer must implement a nonaggressive take. A second alternative is also possible. It is still possible to implement the reducer with an aggressive take, provided the strictness analyser is weakened. The trade-offs between the alternatives will be further explored below.

4.1 An Aggressive Take or a Nonaggressive Take

The consequences of having or not having an aggressive take for the G-Machine, the TIM, the SGM, and the STG-M are as follows:

- The G-machine has a nonaggressive take. Rule 8 of Table 4 of Johnsson [1984] makes an explicit check to see if enough arguments are available in the current stackframe. If not, it falls back into Rule 9, which handles the return of a partial application. The G-machine can be made aggressive by removing the argument satisfaction check from Rule 8 and removing Rule 9 altogether.
- The TIM has an aggressive take. The description of the Take instruction of Fairbairn, and Wray [1987, p. 38] reveals that the Take before an instruction *l* will remain active until there are enough arguments to execute the instruction *l*. The TIM is a simple and elegant way of lazy graph reduction. Making it nonaggressive requires some complication. Besides the normal markers that now take care of updating, a second type of marker is needed to make the Take back up. The compilation schemes for generating TIM code also need to be extended to push the new marker when there is a chance of reducing a partial application.
- The SGM has a nonaggressive take. In a similar way to the G-machine, removal of the 6th SUNWIND rule in Section 9.3 of Burn et al. [1988] makes this machine aggressive. Unlike the G-machine, an argument satisfaction check is still needed to see if the end of a stack frame is encountered (the stack frame acts here as an update marker).
- The STG-M has an aggressive take. Rules 17 and 17a in Section 5.6 of Peyton Jones [1992] show that Enter keeps on entering until there are enough arguments. It will not return a partial application. Making this machine nonaggressive requires similar changes, as in the TIM.

The conclusion is that, for lazy graph reducers, it is easier to implement an aggressive take than a nonaggressive take. In all cases fewer rules are needed to build an aggressive version. This will probably lead to a faster implementation.

4.2 A Weaker Strictness Analysis

A weakened strictness analyzer is required if an aggressive implementation wants to make use of strictness information. The weakened analyzer makes it impossible that the machine needs to return a partial application. As described in Section 3, the problem only occurs when a strict argument evaluates to a partial application, which is always a function. To deal with this, some new terminology is needed. Besides the notion of strictness, the notion of *data strictness* is introduced. A strict argument in a function means that it is certainly going to be needed in the evaluation of the function. A data-strict argument means the argument will certainly be needed *and* that it will not reduce to a function. The following examples show the concept:

square	a\$	= a * a	::	int → int
seq	b\$ c\$	= IF b THEN c ELSE c FI	::	bool → α → α
iswap	d e\$	= e (d + 1)	::	int → (int → α) → α
id	f\$	= f	::	α → α

On the left are a few function definitions together with the output of a conventional strictness analyzer, as in Peyton Jones [1987, Chap. 22]. All strict arguments are marked with a \$. But strict does not imply data strict. Argument a to square is used as a value, so it is data strict. The same holds for argument b to seq. But the use of argument c is not clear. It is needed, but it is unknown whether it reduces to data or to a partial application. So c is strict but not data strict. Argument e is also strict, and it is certain that it is a function, not data. So e is not data strict. Finally, the familiar identity function, id, is not data strict in its argument.

A strictness analyzer that uncovers data strictness can be built by combining the results of a standard two-element domain strictness analyzer and the polymorphic type checker. In the examples above, the right column gives the type derivations by the type checker. A strict argument that corresponds to a polymorphic (α) or function type ($\alpha \rightarrow \gamma$) is not data strict.

In the generated code, only data-strict arguments can be evaluated before entering the function.

4.3 Further Refinement

A further refinement on weakened strictness analysis comes from the observation that the environment of a function call might know more about the types of the arguments than the callee. For polymorphic function arguments, such as the argument of id, it may well be the case that the caller does know the type.

To employ this knowledge, the strictness analysis has to be a normal strictness analysis. Only in the code generator where the strictness results are used must three cases of strict arguments be distinguished:

- A strict argument that has no polymorphic type and no function type can be evaluated strictly.
- Strict arguments that have a function type may not be evaluated strictly because that would suffocate the aggressive take.
- Strict arguments with polymorphic type (*) may only be strictly evaluated if it is known at the caller site that the argument does not reduce to a function.

The result is that more arguments can be evaluated strictly than in the weakened analyzer (third case is added), while the implementation can still use an aggressive take.

5. RELATED WORK

Fairbairn and Wray [1987] mention the possibility of extending the TIM with nonaggressiveness to take advantage of all available strictness. They state without further explanation that their experiments show that the cost of this is higher than the benefits. This statement contradicts with the results of the experiments in Section 6 below.

The difficulties with the aggressive take did not turn up in Argo's [1989] paper on improving the TIM. In that paper a number of strictness optimizations are described. However, these always apply to the strict evaluation of arguments to primitive functions like addition and multiplication. These arguments all reduce to a data value and never to a function. In that case, the problem does not occur, and the improved TIM works fine. The set of strict arguments considered by Argo is a subset of those found by a weakened analyzer.

Howe [1992] also considers the use of strictness optimization in a spineless machine. Howe shows that the use of strictness to change evaluation order can increase performance in some cases on the STG-M, chiefly by avoiding updates. This work is done in the context of strictness analysis by evaluation transformers. Evaluation transformers use strictness properties of data structures to allow some argument expressions to be evaluated in advance to a greater extent than is allowed by simple strictness analysis. The possibility of strict evaluation of arguments that reduce to a function is not explored. The strict arguments found by this method extend over those of the weakened analysis but only in the direction of data structures and not of partial applications.

For the SGM, Burn et al. [1988] offer the following explanation for the presence of the 6th SUNWIND rule: "The new stack frame is required because **E** is used to compile strict arguments, whose result may be partial applications[,] and so we must prevent them from taking other things from the stack." Hence, the SGM is not aggressive and needs no weakening of the strictness analyzer.

6. EXPERIMENTAL RESULTS

We will now show a number of relevant experimental results to gain understanding in the trade-off between the spineless reduction strategy with aggressive take (like the TIM and the STG-M) in combination with a data strictness analyzer, and the nonaggressive-take reduction strategy in combination with a conventional, more powerful, strictness analyzer.

The platform used is based on the STOFFEL compiler. STOFFEL is a simple typed lazy functional language [Beemster 1992] which may be used as an intermediate in the compilation of MIRANDATM or HASKELL. The STOFFEL compiler performs many of the standard optimizations on lazy functional programs and compiles into C [Beemster 1993]. All measurements are done on a Sun SPARC 10/41. The compiler performs global first-order strictness optimizations based on backward analysis. The quality of the strictness analyzer is comparable to that of other known implementations. For the experiment with the weakened strictness analyzer, taking into account the presence of an aggressive take, the results of the polymorphic type checker are combined with those of the strictness analyzer. This is the method of Section 4.2, and not the refined method. The results of the refined method will be in between the weakened analyzer and the nonaggressive variant, so only the extremes are shown.

Table I lists the test programs used for the experiments. The first three, arti, nfib, and queens, are toy-sized programs, not suitable on their own for final conclusions. The other programs are realistic medium-sized programs. The table lists the number of lines of each program, the number of functions, and a short description. These measurements are done by counting the output of a pretty-printer that strips comments and uses a uniform output format. nfib does not appear to be a one-liner because library conversion functions are included for I/O.

The arti program was specially constructed for this test. Its goal is to show that an aggressive implementation with weakened strictness analysis may perform arbitrarily less efficiently than a nonaggressive reducer with normal strictness analysis. The core code of arti is as follows:

```
main ::  $\alpha \rightarrow$  list char
main a =
    posit (total (map (doafun 1 2 3 4 5 6 7 8 9 10 11 12) (fromto 1 500000))))
doafun ::  $\mu \rightarrow \lambda \rightarrow k \rightarrow \iota \rightarrow \theta \rightarrow \eta \rightarrow \zeta \rightarrow \epsilon \rightarrow \delta \rightarrow \gamma \rightarrow \beta \rightarrow \alpha \rightarrow$  int  $\rightarrow$  int
doafun i1 i2 i3 i4 i5 i6 i7 i8 i9 i10 i11 i12 ding =
    select2 99 (afun ding i1 i2 i3 i4 i5 i6 i7 i8 i9 i10 i11 i12)
select2 :: int  $\rightarrow \alpha \rightarrow \alpha$ 
select2 i poly = IF i < 0 THEN select2 i poly ELSE poly FI
afun :: int  $\rightarrow \mu \rightarrow \lambda \rightarrow k \rightarrow \iota \rightarrow \theta \rightarrow \eta \rightarrow \zeta \rightarrow \epsilon \rightarrow \delta \rightarrow \gamma \rightarrow \beta \rightarrow \alpha \rightarrow$  int
afun ding a b c d e f g h j k l m =
    IF ding >= 0 THEN ding ELSE afun ding a b c d e f g h j k l m FI
```

TM MIRANDA is a trademark of Research Software Ltd.

Table I. Test Program Sizes and Descriptions

Program	Lines	Functions	Description
arti	42	13	Contrived program to fool non-aggressive implementations
nfib 30	39	19	The well known fibonacci program
queens 11	178	39	Computes all variants of 11 queens on an 11x11 chess board
ray	956	62	Ray tracer, computes projection of simple scene [14]
event	627	70	Event driven simulation of a flip-flop [15]
wang	838	79	System of equations solver [21]
sched	974	129	Finds optimal scheduling of dependent events [19]
typecheck	1790	176	Polymorphic typechecking according to [17]-Ch.9
parst	3470	356	Combinator parser for STOFFEL including semantic checks
transform	5898	295	Cycle lifting transformation for functional programs [20]

Note: [14] refers to Kelly [1989]; [15] refers to Muller [1993]; [17] refers to Peyton Jones [1987]; [19] refers to Vree [1989]; [20] refers to Vree and Hartel [1992]; and [21] to Wong [1981].

The interesting interaction in this example is between `doafun` and `select2`. The main expression takes care of creating a curried function from `doafun` and making sure that it is used many times so that the effects become visible. The `select2` and `afun` functions are recursive to prevent the STOFFEL optimizer from inlining and spoiling the action. The second argument of `select2` is the one that counts. It is strict and polymorphic. This means that a weakened strictness analyzer will not mark it strict, but a normal analyzer will. Therefore, in an aggressive-take implementation the application of `select2` cannot strictly reduce the second argument. Instead, an expensive suspension must be built that will be reduced later by `select2`.

The second and third columns of Table II show a compile-time count of how many arguments are computed in a strict fashion before they are passed to a function. This is a count of arguments actually computed strictly at the call site. It is given for both the normal and the weakened strictness analysis.

Here the first interesting result surfaces. It turns out that for six of the ten test programs, the generated code is unaffected by the weaker strictness analysis. For these programs there is no difference in executing with the aggressive or nonaggressive reduction scheme.

For the other four programs there are some arguments that can be passed strictly under a nonaggressive take, and not with an aggressive take. The relative number of these arguments is very small. However, even a single difference to the generated code might cause significant changes at run-time. Indeed, this is confirmed for the contrived `arti` program in Table II.

In the right part of Table II the dynamic results of running the programs compiled with normal and weakened strictness analysis are listed. The figures given are the number of kilobytes heap used. For lazy functional programs, the heap usage is usually a good indicator of performance.

In these run-time measurements, the trends from the compile-time count on strictly called arguments are largely confirmed. The programs with equal strict-call-count use equal heap as expected. Some of the programs with differing strict-call-count also use equal heap. This is the case for `typecheck` and `parst`. In these two programs the differences in generated code are either not executed often (in initialization or wrap-up, for example), or the strict

Table II. Static Count of Arguments Passed Strict and Dynamic Heap Usage in Kbytes, Both for Normal and Weakened Strictness Optimizations

Program	arg's passed strict		heap usage	
	normal	weak	normal	weak
arti	14	12	52000	84000
nfib	8	8	0	0
queens	26	26	116439	116439
ray	45	45	154965	154965
event	90	90	91533	91533
wang	101	101	172299	172299
sched	60	60	60547	60547
typecheck	199	197	228803	228803
parst	504	498	72443	72443
transform	785	782	478369	480847

arguments turn out to be already reduced, just like (add 5) in the swap example.

The arti program shows that the cost of weakened strictness analysis can indeed be huge. In this version of arti the extra heap usage is more than 50%. Experiments (not shown) confirmed that by adding extra arguments to arti's functions, the difference could be made arbitrarily large. The transform program shows that it is indeed sometimes beneficial in a realistic program to have a nonaggressive take and to be able to reduce arguments of unknown type strictly. However, the benefit is minimal.

The difficulties in the construction of the arti program prompted an experiment with compiling the programs without optimizing transformations. In the construction of arti it was difficult to convince the function inliner (one of the optimizing transformations) that it should not transform the program and remove the positive effects of normal over weak strictness analysis. The question arose of whether the optimizing transformations would remove all these positive effects in realistic programs, while nonoptimized programs would suffer a huge disadvantage from weaker strictness analysis. The experiment (not in a table) showed that this is not the case. Although turning off the transformations resulted, in many cases, in more than 50% increased heap usage, the relative differences between normal and weakened strictness remained within 1% of each other.

Finally, Table III presents execution times in seconds for the ten test programs. These measurements were done by running each variant of the programs ten times and taking the best run-time of these. Within the Unix workstation environment of the experiments, the measurement error was found to be within the order of 2 to 3%. To interpret the results it is important that the STOFFEL implementation uses a nonaggressive take, *also when the weaker strictness analyzer is used*. However, it can be argued that although the STOFFEL machinery is not as lean as it could be, the performance impact is minimal. STOFFEL's reducer is a spineless reducer. This

Table III. Execution Times in Seconds with Normal and Weak Strictness Analysis

Program	normal strict	weak strict
arti	25.2	35.3
nfib	1.1	1.1
queens	34.6	34.7
ray	33.7	33.6
event	30.3	30.0
wang	55.2	55.0
sched	14.7	14.6
typecheck	82.0	80.2
parst	54.4	52.3
transform	129.6	130.2

implies that argument satisfaction checks must be done anyway to check for update markers. Since this check can be combined with the check to see if there are arguments at all, no big differences can be expected.

Other differences in execution time that are to be expected are twofold. First is overhead for the manipulation of extra suspensions in the case of weak strictness. This should be in the same order as the extra heap usage. For *arti* this is indeed confirmed: heap usage is a useful indicator for execution-time performance. For all the other programs the difference in heap usage is less than the variations due to measurement errors in the time. No big differences can be expected and are not found.

The second cause for differences in execution time could have adverse effects on a nonaggressive implementation with normal strictness. It is caused by the strict evaluation of partial applications that cannot be reduced further, like (add 5). Such evaluation causes overhead and saves nothing. It must be noted, however, that the *STOFFEL* compiler can and will detect that simple expressions (like simple curried functions) cannot be further reduced and will not try to evaluate them. From the numbers in Table III, this adverse effect could not be confirmed. Except for *arti*, all differences in execution times are within the realm of measurement errors.

7. CONCLUSIONS

The elegant spineless reduction strategy of the TIM cannot be combined with standard global strictness optimizations because of the TIM's *aggressive take*. This little-known property of the TIM was later corrected in the SGM, but subsequently reintroduced in the third-generation spineless machine, the STG-M. The G-machine never suffered from it.

Two ways of dealing with the aggressive take have been proposed. One is removing it from the reduction engine; the other entails modification of the strictness analyzer. All of the graph reduction models discussed here can make a choice between one of the two options. The advantage of having an aggressive take lies in a leaner and, hence, possibly faster implementation of

lazy graph reduction. A potential disadvantage is the need for a weaker strictness analyzer when using the aggressive take.

With the STOFFEL platform, a number of experiments were conducted to evaluate the trade-off between the aggressive-take and normal strictness optimizations. It has been shown that in realistic programs the performance benefits of normal strictness analysis over weakened strictness are minimal. For a number of programs there is no difference at all. On the other hand, using a contrived program shows that there is potentially a huge loss when running with an aggressive take.

Taking all together, there is no decisive argument in favor of or against an aggressive take. An implementor of lazy graph reduction can choose a leaner reduction machine with aggressive take and more complicated code generation, or choose an evaluator that can evaluate any kind of redex and use the full benefits of strictness analysis.

ACKNOWLEDGMENTS

I thank Pieter Hartel, Koen Langendoen, Wim Vree, and Hugh McEvoy for detailed comments on draft versions of this paper. I am grateful to Geoffrey Burn, Denis Howe, and Simon Peyton Jones for their critical remarks and encouragement, and, to the anonymous referees for important and helpful feedback.

REFERENCES

- ARGO, G. 1989. Improving the three instruction machine. In the *ACM 4th Conference on Functional Programming Languages and Computer Architecture*, J. Stoy, Ed. ACM, New York, 100–115.
- AUGUSTSSON, L. and JOHNSON T. 1989. The Chalmers lazy-ML compiler. *Comput. J.* 32, 2 (Apr.), 127–141.
- BEEEMSTER, M. 1993. Optimizing transformations for a lazy functional language. In the *7th Computer Systems*, W.-J. Withagen, Ed. Eindhoven Univ. of Technology, The Netherlands, (Nov.), 17–40.
- BEEEMSTER, M. 1992. The lazy functional intermediate language STOFFEL. Tech. Rep. CS-92-16, Dept. of Computer Systems, Univ. of Amsterdam, The Netherlands.
- BURN, G. L., PEYTON JONES, S. L., and ROBSON, J. D. 1988. The spineless G-machine. In the *ACM Conference on Lisp and functional programming*. ACM, New York, 244–258.
- CLACK, C. AND PEYTON JONES, S. L. 1985. Strictness analysis—a practical approach. In *2nd Functional Programming Languages and Computer Architecture*, J.-P. Jouannaud Ed., Lecture Notes in Computer Science, vol. 201, Springer-Verlag, Berlin, 35–49.
- DAVIS, K. AND WADLER, P. L. 1989. Backwards strictness analysis: Proved and improved. In *Functional Programming*, K. Davis and R. J. M. Hughes Eds. Springer-Verlag, Berlin, 12–30.
- FAIRBAIRN, J. AND WRAY, S. C. 1987. Tim: A simple lazy abstract machine to execute supercombinators. In the *ACM 3rd Conference on Functional Programming Languages and Computer Architecture*, G. Kahn, Ed. Lecture Notes in Computer Science, vol. 274, Springer-Verlag, New York, 34–45.
- THE GRASP TEAM. 1992. The glorious HASKELL compilation system user's guide. Tech. Rep. Dept. of Computer Science, Univ. of Glasgow, Scotland.
- HARTEL, P. H., GLASER, H. W., AND WILD, J. M. 1994. Compilation of functional languages using flow graph analysis. *Softw. Pract. Exp.* 24, 2 (Feb.), 127–173.
- HOWE, D. B. 1992. Experiments with strict STG code. In the *4th International Workshop on the Parallel Implementation of Functional Languages*, H. Kuchen, and R. Loogen, Eds. Aachener Informatik-Berichte 92–19, Fachgruppe Informatik, RWTH Aachen, Germany, 1–8.

- HUDAK, P., PEYTON JONES, S. L., AND WADLER, P. L., Eds. 1992. Report on the programming language HASKELL—a non-strict purely functional language, version 1.2. *SIGPLAN Not.* 27, 5 (May) 1–162.
- JOHNSON, T. 1984. Efficient compilation of lazy evaluation. *SIGPLAN Not.* 19, 6 (June), 58–69.
- KELLY, P. H. J. 1989. *Functional Programming for Loosely-Coupled Multiprocessors*. Pitman, London, England.
- MULLER, H. L. 1992. Simulating computer architectures. Ph.D. thesis, Dept of Computer Systems, Univ. of Amsterdam, (Feb.), The Netherlands.
- NÖCKER, E. G. J. M. H. 1990. Strictness analysis using abstract reduction. Implementation of functional languages on parallel architectures. Tech. Rep. 90-16, Dept. of Computer Science, Univ. of Nijmegen, The Netherlands, (June), 171–201.
- PEYTON JONES, S. L. 1992. Implementing lazy functional languages on stock hardware: The spineless tagless G-machine. *J. Funct. Program.* 2, 2 (Apr.) 127–202.
- PEYTON JONES, S. L. 1987. *The Implementation of Functional Programming Languages* Prentice-Hall, Englewood Cliffs, NJ.
- VREE, W. G. 1989. Design considerations for a parallel reduction machine. Ph.D. thesis, Dept of Computer Systems, Univ of Amsterdam, The Netherlands, Dec.
- VREE, W. G. AND HARTEL P. H. 1992. Communication lifting: Fixed point computation for parallelism. Tech. Rep. CS-92-07, Dept of Computer Systems, Univ. of Amsterdam, The Netherlands, Dec.
- WANG, H. H. 1981. A parallel method for tri-diagonal equations. *ACM Trans Math. Softw.* 7, 2 (June), 170–183.