Avoiding Unnecessary Updates

John Launchbury, Andy Gill, John Hughes,
Simon Marlow, Simon Peyton Jones, Philip Wadler
Computing Science Department,
Glasgow University

Abstract

Graph reduction underlies most implementations of lazy functional languages, allowing separate computations to share results when subterms are evaluated. Once a term is evaluated, the node of the graph representing the computation is *updated* with the value of the term. However, in many cases, no other computation requires this value, so the update is unnecessary. In this paper we take some steps towards an analysis for determining when these updates may be omitted.

1 Introduction

There are two obvious ways to reduce lambda expressions: outside-in or inside-out. The former is called normal-order reduction, the latter applicative-order. Neither of these mechanisms guarantee to perform fewer reductions than the other. Normal-order reduction only ever reduces terms that are definitely required, but it may end up reducing a single term more than once. On the other hand, while applicative order reduction is less likely to reduce a single term more than once, it may reduce terms unnecessarily, even to the extent of failing to terminate.

There is a popular middle ground, commonly called lazy evaluation. Semantically, lazy evaluation is equivalent to normal-order reduction—only terms which are known to be required are evaluated. Operationally, however, lazy evaluation matches exactly the applicative order behaviour in avoiding repeated evaluation. (Note that neither applicative-order reduction nor lazy evaluation is an optimal evaluation strategy in the sense of Levy [4], so both may sometimes repeat reductions.)

One common method by which lazy evaluation achieves its behaviour is *graph reduction*. When substitution takes place, a reference to an expression is substituted, rather than the expression itself. If the expression is ever evaluated, it is replaced by its value so that all other references to the term immediately see the reduced value rather than the original unreduced term. This replacement, or update, is precisely the point which distinguishes lazy evaluation from normal-order reduction. Hence, normal-order is sometimes called *tree reduction* in contrast to lazy evaluation's graph reduction.

While graph reduction supplies undoubted benefits, it also has associated costs. Updating the reference always costs instructions, and the cost of interrupting the computation at the appropriate time may be even greater. Normally of course this cost is very small when compared with the cost of recomputing a value, but it exists nonetheless. On the parallel machine GRIP [5]

updates are particularly expensive because the updated node has to be flushed from local memory out to the global store. Similarly, the need for updates creates a major complication in the TIM abstract machine [2], and the presence of the *update markers* interrupts the flow of evaluation. Indeed, Fairbain and Wray used a local analysis to cut down on such update markers, but unfortunately the analysis assumed a fairly naive model of implementation and precluded more efficient alternative implementations.

In this paper we take some steps towards an analysis which detects when updates may be omitted. It is a working paper and probably contains many omissions, but nonetheless addresses an important issue in the implementation of lazy functional programming languages.

The analysis is presented in the style of type rules. This has the advantage of allowing information to flow both forwards and backwards through a program, but it has the disadvantage of being that much further from an implementation. Currently the analysis does not handle products or other data structures, but it is higher-order. Also, explicit recursion is not presented here, but we do not expect it to pose much of a problem.

2 The Language

We will use a stylised form of lambda expression extended with *let*s, plus a few other constructs for convenience. The form of expressions is a simplification of the Spineless, Tagless, G-machine implementation language (STG) [6] used in the Glasgow Haskell compiler.

The underlying philosophy of the language is that it has a direct operational reading, a sort of "abstract machine-code" for functional languages. Closures are named explicitly using *let*s, and functions accept only such closures as arguments. This philosophy is particularly appropriate for our analysis because it provides an ideal place for update annotations to be placed. Note that the *only* means of constructing closures is by using *let*s.

We make two exceptions to the rule regarding function arguments, both purely for the sake of readability: explicit numbers may also be used as arguments to functions; and primitive operations such as + may be applied to arbitrary expressions.

$$
\begin{array}{lcl}
e \in \mathit{Expression} & ::= & a \\
 & | & \lambda x.e \\
 & | & e\ a \\
 & | & \mathit{let}^{i}\ x = e_1\ \mathit{in}\ e_2 \\
 & | & e_1\ *\ e_2 \\
a \in \mathit{Atom} & ::= & x \\
 & | & n \\
 & & \\
* \in \mathit{Primitive} & ::= & +\ |\ \ldots \\
x \in \mathit{Variable} & & \\
n \in \mathit{Integer} & & \\
i \in \mathit{Annotation} & &
\end{array}
$$

The aim of the analysis is to discover which *let*s create closures that need to be updated and which do not. The result of the analysis is expressed as an annotation placed on the *let*. The details of the annotations will be given later.

3 Why it's difficult

Examples are often valuable for providing intuition about a problem. Furthermore, in our case they will provide an informal understanding of the STG reduction model.

Consider the program fragment,

$$\begin{array}{l} let\ u = \ldots\ in \\ \quad let\ v = \ u + 3\ in \\ \quad\quad v + v \end{array}$$

To obtain the value of the expression, we need to know which two values to add together. The first is v. To evaluate v, we need to get the value of u. Suppose u evaluates to 5. Because graph reduction guarantees not to recompute values, u has its closure updated with the number 5. Now v can be evaluated, producing 8, and its closure updated (with the number 8).

We now have to find the value to the second argument of $+$, so again we need the value for v. However, as v's closure was overwritten with 8 we can obain its value immediately *and we do not have to reaccess* u. 8 is added to 8 to give the answer 16.

3.1 Hidden References

Because u was only accessed once, we could have omitted updating u's closure without causing any computation to be repeated. Note that u was only accessed once even though v was accessed twice, and v depended on u. This means that reaccessing is not necessarily transitive. Sometimes it is, however. Consider the next example.

$$\begin{array}{l} let\ u = \ldots\ in \\ \quad let\ v = \lambda x.u + x\ in \\ \quad\quad v\ 3 + v\ 4 \end{array}$$

This time v is a function which adds its argument to the value of u. Every time we use v we will need to know u's value. The problem is that even though v is already in weak head normal form (whnf) it still contains a reference to u. Thus in this example updating u's closure with its value is necessary to save recomputing that value. The next two examples show this very clearly.

$$\begin{array}{l} let\ u = \ldots in \\ \quad let\ v = (let\ \ w = u + 1\ \ in\ \lambda x.w + x)\ \ in \\ \quad\quad v\ 3 + v\ 4 \end{array}$$

In this case, while both v and w need to be updated once evaluated, u does not because it is only used once: on evaluating $v\ 3$, w is evaluated (accessing u) and is overwritten with its value. Now all reference to u is lost, so even when v is used again, u is not reaccessed.

Contrast this with

$$\begin{array}{l} let\ u = \ldots\ in \\ \quad let\ v = \lambda x.(let\ \ w = u + 1\ \ in\ \ w + x)\ \ in \\ \quad\quad v\ 3 + v\ 4 \end{array}$$

Each time v is used it constructs a new closure for w (because in principle w's value could depend on x) and so continues to retain a reference to u. Thus once u is evaluated, its closure must be updated to avoid recomputing its value on a subsequent use of v. Taken together, this and the previous example show how sensitive the issue of avoiding updates is to the precise form of the expression. Denotationally the two expressions are equivalent (one is a λ-lifted version of the other) but their operational behaviour is different.

The examples have demonstrated that there are two issues to be addressed to produce a useful analysis. The first is whether a closure is duplicated or not. The second is whether duplication of a descendant closure affects the original or not. The (fairly simplistic) approach we adopt here is to assume that duplication of functions possibly duplicates closures the functions refer to, whereas duplication of an atomic value does not. Once an atomic value is reduced to weak head normal form (which for atomic values is the same as normal form) it cannot contain references to other closures which may be accessed at a later point.

4 Update-Avoidance Analysis

In the analysis we use annotated types to register when multiple accesses are possible. We are not interested in the distinction between types such as *Integer* or *Bool*, but we are interested in the level of structure present in a type, in particular whether the object is a product or a function. For simplicity we restrict ourselves to consider functions.

Types to the left of function arrows carry annotations which specify whether the function possibly duplicates its argument or not. Thus types are of the form,

$$\begin{array}{lcl} S, T \in \mathit{Type} & ::= & K \\ & \mid & A \rightarrow T \\ A, B \in \mathit{AnnType} & ::= & T^i \end{array}$$

where *AnnType* is the annotated types. The annotations record whether a value may possibly be used zero, one or many times.

$$\begin{array}{lcl} i, j \in \mathit{Ann} & ::= & \mathit{Zero} \\ & \mid & \mathit{One} \\ & \mid & \mathit{Many} \end{array}$$

The annotations are interpreted in the following way:

Zero	Never used;
One	Certainly used no more than once;
Many	May be used any number of times.

and we assume an ordering of $\mathit{Zero} < \mathit{One} < \mathit{Many}$.

This interpretation incorporates a notion of *safe approximation*. We may end up deciding that a value could be used many times, when in fact it is only ever used once. Of course, the better the analysis is, the less frequently it will overapproximate in this way. Graph reduction is ultra-conservative in this sense in that it updates every closure whether it is used more than once or not.

$$\textit{Weak} \quad \frac{\Gamma \vdash e : T}{\Gamma, x : S^{Zero} \vdash e : T}$$

$$\textit{Cont} \quad \frac{\Gamma, x : S^{Many}, y : S^{Many} \vdash e : T}{\Gamma, z : S^{Many} \vdash e[z/x, z/y] : T}$$

$$\textit{Dere} \quad \frac{\Gamma, x : S^{i} \vdash e : T}{\Gamma, x : S^{j} \vdash e : T} \ (i \leq j)$$

$$\textit{Exch} \quad \frac{\Gamma, x : A, y : B \vdash e : T}{\Gamma, y : B, x : A \vdash e : T}$$

Figure 1: Structural rules

4.1 Structural Rules

The analysis is given in the form of type rules. Judgements are of the form,

$$\Gamma \vdash e : T$$

This is read that in the type environment Γ, we may deduce that e has type T (note, no annotation on T). Type environments are partial functions, mapping variables to annotated types. Thus each variable occurs at most once in a type environment. That is,

$$\Gamma, \Delta \in \textit{TypeEnv} \quad ::= \quad x : A, y : B, \ldots$$

We will often write the assumptions making the annotations on the types explicit.

The structural rules given in Figure 1 define the behaviour of type environments. The weakening rule allows any variable to be introduced with a *Zero* annotation, and the contraction rule allows two occurrences of a variable to be combined so long as they both have the *Many* annotation. The renaming is present to maintain the invariant that each variable occurs once only in the type environment.

In order to allow variables with possibly other annotations to be combined, the dereliction rule allows annotations to be degraded. This clearly has the potential for losing information so should only be applied when necessary. Finally the exchange rule shows that the order of assumptions is unimportant.

4.2 The Analysis Rules

The analysis rules appear in Figure 2. The variable rule ensures that any new variables appear in the type environment with annotation at least *One* (dereliction allows this to be degraded to *Many*), and the constant rule states that numbers are an atomic type.

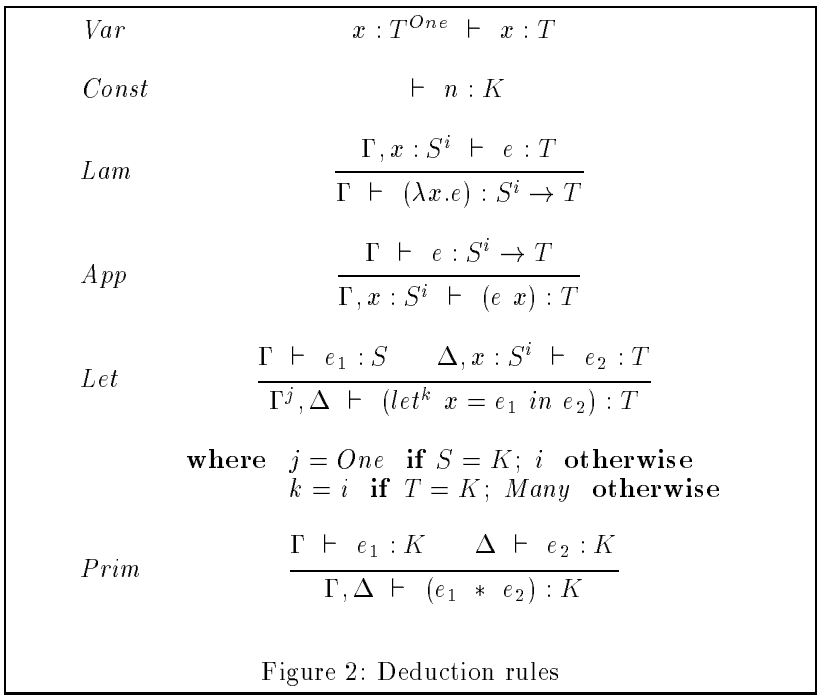

Figure 2: Deduction rules

One pleasant consequence of only ever applying functions to variables is that the lambda and application rules are dual to each other. They have the effect of moving annotations between type environments and types.

There are essentially four variants of the let rule, depending on whether the variable being bound has an atomic type or not, and whether the term being built has an atomic type or not.

When the bound variable is of atomic type, it can contain no references to other closures once it is evaluated. So multiple accesses of the value will not propagate to the references contained in Γ, and their annotations may remain unchanged.

This is not the case for a composite structure, however. A function may contain references to other closures which are accessed *each time* the function is used. Thus all the free variables used in the definition of the function must be given an annotation at least as high as that of the function, for if the function is accessed many times, then so may any internal references.

To model these two cases we introduce an operation on type environments whose effect is to propagate annotations to every type in the environment. We define,

$$(x_1 : T_1^{i_1}, \ldots, x_n : T_n^{i_n})^j = (x_1 : T_1^{i_1 . j}, \ldots, x_n : T_n^{i_n . j})$$

where

$$\begin{array}{lcll} i \,.\, Zero & = & Zero & \\ i \,.\, One & = & i & \\ i \,.\, Many & = & Zero, & \textit{if } i = Zero \\ & = & Many, & \textit{otherwise} \end{array}$$

A similar situation arises when the term being built is of atomic type. Even if the term is bound to a variable which is used many times, none of the references used in its definition can escape. That is, none of the closures used in defining e_2 will ever be accessed more frequently than the number of accesses given in e_2. By assumption, this is already recorded by their annotation. Thus x in particular cannot be accessed more frequently than described by the annotation i, and so in this case the *let* is also annotated with i.

If e_2 is of function type, however, then there is not sufficient information present to determine whether the closure being built will only be accessed once or not. If the function is bound to a variable which is used many times, then there may be multiple accesses of x, even if it only occurs once in e_2. So in this case we are conservative and assume multiple accesses are possible, and we record this on the *let* so that an update may be performed if necessary (let^{Many} constructs an updatable closure).

5 Results

In this section we show the analysis working on the examples presented earlier.

Example 1

$$\begin{array}{l} let\ u = \ldots\ in \\ \quad let\ v = \ u + 3\ in \\ \quad\quad v + v \end{array}$$

First we note that using contraction and dereliction, together with the *Var* and *Prim* rules, we have,

$$\dfrac{\dfrac{\dfrac{\overline{v_1 : K^{One} \vdash v_1 : K}}{v_1 : K^{Many} \vdash v_1 : K} \quad \dfrac{\overline{v_2 : K^{One} \vdash v_2 : K}}{v_2 : K^{Many} \vdash v_2 : K}}{v_1 : K^{Many}, v_2 : K^{Many} \vdash v_1 + v_2 : K}}{v : K^{Many} \vdash v + v : K}$$

We can use this to obtain,

$$\dfrac{\dfrac{\overline{u : K^{One} \vdash u : K} \quad \overline{\vdash 3 : K}}{u : K^{One} \vdash u + 3 : K} \qquad v : K^{Many} \vdash v + v : K}{u : K^{One} \vdash let^{Many}\ v = u + 3\ in\ v + v : K}$$

The *Many* annotation from the assumption about v has come to rest on the *let* binding v, but as v has an atomic type, its annotation is not propagated to the type environment, so u retains its original annotation.

Finally, assuming that we have some Γ for which we may infer $\Gamma \vdash \ldots : K$, we have

$$\dfrac{\Gamma \vdash \ldots : K \quad u : K^{One} \vdash let^{Many}\ v = u + 3\ in\ v + v : K}{\Gamma \vdash let^{One}\ u = \ldots\ in\ (let^{Many}\ v = u + 3\ in\ v + v) : K}$$

The final result is that u is used once only, but that v may be used more than once.

Example 2

$$\begin{array}{l} let\ u = \ldots\ in \\ \quad let\ v = \lambda x.u + x\ in \\ \quad\quad v\ 3 + v\ 4 \end{array}$$

The interesting part of this example is the binding of v. First note that we can derive,

$$\dfrac{\dfrac{\overline{u : K^{One} \vdash u : K} \quad \overline{x : K^{One} \vdash x : K}}{u : K^{One}, x : K^{One} \vdash u + x : K}}{u : K^{One} \vdash \lambda x.u + x : K^{One} \to K}$$

Following a path similar to that of example 1, we can also derive,

$$v : (K^{One} \to K)^{Many} \vdash v\ 3 + v\ 4 : K$$

Putting these together with the *let* rule gives,

$$u : K^{Many} \vdash let^{Many}\ v = \lambda x.u + x\ in\ v\ 3 + v\ 4 : K$$

Because v is not of atomic type, all the free variables involved in its binding receive v's annotation. Thus u has the annotation *Many*, so will be bound in an updatable closure.

Example 3
The final example covers the two code fragments

$$\begin{array}{l} let\ u = \ldots\ in \\ \quad let\ v = (let\ w = u + 1\ \ in\ \lambda x.w + x)\ \ in \\ \quad\quad v\ 3 + v\ 4 \end{array}$$

and

$$\begin{array}{l} let\ u = \ldots\ \ in \\ \quad let\ v = \lambda x.(let\ \ w = u + 1\ \ in\ \ w + x)\ \ in \\ \quad\quad v\ 3 + v\ 4 \end{array}$$

The analysis annotates u and v as updatable in both cases,[1] and w as updatable only in the first. When evaluating the term in the second case, a new closure w is generated each time the function v is called, and this fresh closure is only ever used once. Thus is may safely be marked as a non-updatable closure.

In contrast, in the first case, a single closure for w is generated and reused each time v is called, so requiring the closure for w to be updatable. This shows a weakness with the analysis, for if w is updated, future references to v will not refer to u, so in fact u need not be updated. However the analysis is conservative here, and is not able to distinguish between the uses of u in the two examples.

A more accurate (and presumably more expensive) analysis could keep track of the *depth* at which free variables occur in function valued expressions to determine whether references to them will remain once the function has been evaluated to whnf. Whether the relatively small gain in accuracy would be worth while or not is not clear.

6 Relationship to Linear Logic

There is obviously a close relationship between linear logic the rules presented here. The version of linear logic most closely related is *bounded linear logic* [1], where annotations are placed on the "bangs" to indicate how often a term is used. So rather than using $!T$ which describes a type that can be copied as often as required (ie. an unbounded number of times), types such as $!^n T$ are used. Such a type may be copied up to n times, but no more. The analysis presented here may be viewed as an abstraction of this as we capture two or more uses of a variable as *Many*, but retain *Zero* and *One*. One important aspect of our analysis, however, is that is deals only with banged types, and has no place for linear types.

7 Future Work

This report is a working paper, and quite a lot remains to be done. We currently have no correctness proof for this analysis. The difficulty lies in not having had

[1] Of course, as v is already in whnf, no update ever need take place. It is a simple matter to postprocess a term to remove update annotations from *let*s which bind variable to values already in whnf.

an appropriate level semantics of the STG language. The denotational semantics is too high—it doesn't distinguish between normal-order reduction and lazy evaluation—and the operation semantics is too low as it explicitly describes the heap, stack pointers and the like. New work has recently developed an intermediate level semantics [3] which, we hope, will turn out to be appropriate not merely for this proof, but for others that exploit lazy evaluation.

References

[1] Girard, Scedrov, and Scott, *Bounded Linear Logic*, J of Theoretical Computer Science, 97:1–66, 1992.

[2] J.Fairbairn and S.Wray, *A Simple Lazy Abstract-Machine to Execute Supercombinators*, in Proc. FPCA, Portland, pp 34-45, S-V, 1987.

[3] J.Launchbury, *A Natural Semantics for Lazy Evaluation*, in Proc. ACM SIGPLAN *Principles of Programming Languages*, Charleston, South Carolina, 1993.

[4] J.-J.Lévy, *Optimal Reductions in the Lambda Calculus*, in Seldin and Hindley eds., *To H.B.Curry: Essays in Combinatory Logic, Lambda Calculus and Formalism*, pp 159-191, Academic Press, 1980.

[5] S.Peyton Jones, C.Clack, J.Salkild, M.Hardie, *GRIP - a high-performance architecture for parallel graph reduction.* Proc IFIP conference on Functional Programming Languages and Computer Architecture, Portland. Springer Verlag LNCS 274, pp 98-112, 1987.

[6] S.Peyton Jones, *Implementing Lazy Functional Languages on Stock Hardware: the Spineless Tagless G-Machine*, Journal of Functional Programming, CUP, 1992, to appear.