Hoisting: Lazy Evaluation in a Cold Climate

Simon Finn

Department of Computing Science
University of Stirling
Stirling FK9 4LA

1 Introduction

There has been much work recently aimed at combining the formalisms of functional and logic programming; some, such as the original work on LOGLISP [Robinson and Sibert 82], treat reduction and deduction as two separate mechanisms, while others, such as [Goguen and Meseguer 84] and [Darlington et al. 86] attempt a more comprehensive unification based on narrowing. The author's language, Simplex, follows the LOGLISP approach of having two separate sublanguages (and consequently two implementation mechanisms). The implementation of Simplex differs from that of LOGLISP in two ways: the functional sublanguage is lazy (not applicative order) but the implementation of the logic language uses only depth-first (not breadth-first) search. (So Simplex has a 'good' implementation of functional programming allied with a 'bad' implementation of logic programming, while LOGLISP has the reverse.)

The unfortunate interaction of lazy evaluation and logic programming produces an implementation problem which is examined in this paper. Lazy evaluation is a strategy that delays performing work in the hope that the work will ultimately prove to be unnecessary. The basic assumption is that there is no penalty for the delay - the postponement should not increase the total amount of work required. In a purely functional setting, the chief challenge to the validity of this assumption is posed by beta-reduction. Beta-reduction requires the copying of expressions; if an expression is not already in normal form, this seems to imply a duplication of the work required to normalise it. Fortunately, this problem has been solved by the development of techniques such as graph-reduction; rather than physically copying an expression, an indirection is used instead, so that when (if) the original expression is reduced, the benefit is shared by all of its instances.

The addition of logic programming introduces another, more subtle, way in which an unreduced expression may be copied. The difficulty arises from the branching of the search tree - whenever a disjunction is processed, each possibility must be examined independently (either sequentially or in parallel) and each path explored requires its own copy of every extant

expression. The copying that this implies also poses a problem for lazy evaluation, but one that is not addressed by graph-reduction or suchlike techniques. The difficulty is that delaying the reduction of an expression may lead to the duplication of work since the same reduction may need to be performed on several different search paths. The total amount of work performed on any one search path is not increased by this (so with a processor-per-path machine there might be no difficulty) but the increase in total workload will reduce the overall performance where the number of processors is limited. Some mechanism to allow the results of reductions to be shared between separate search paths is needed to avoid this overhead.

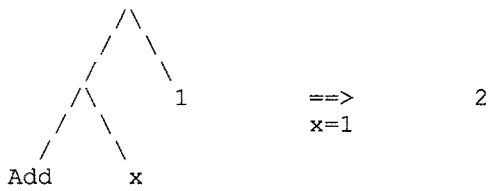
Unfortunately, there is more to the problem than this: sharing the result of a reduction between two different search paths may not be safe. For example, the result of reducing the expression 'x+1' depends on 'x'; if two different search paths instantiate this variable to different values then each must perform the reduction independently. On the other hand, if 'x' has the same value, it may be possible to save a reduction by storing and then reusing the result.

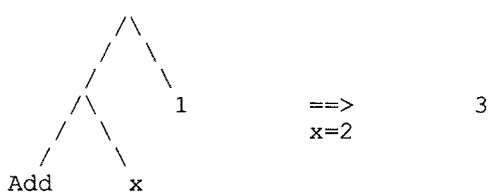
The subject of this paper is a technique - hoisting - which is designed to detect whether (and how widely) the result of a reduction may be shared, and to implement its sharing. (The name 'hoisting' derives from the way that a long-delayed reduction may be 'hoisted' up the search tree so that its result may be shared by many search paths.) In a sense, hoisting is a generalisation of lazy evaluation; for purely functional programs, a lazy reducer performs no more (and possible fewer) reductions than an applicative-order one. The addition of the branching occasioned by logic programming destroys this pleasant property: for any given program, lazy evaluation may or may not require fewer reductions. Whatever the program, a system employing hoisting will do at least as well (in terms of the number of reductions performed) as a comparable lazy or applicative-order system. The implementation of hoisting will necessitate some additional run-time overheads; the amount required will depend on the particular search strategy employed. The combination of hoisting with parallel or breadth-first searching does not seem attractive - the overheads required are too great - but for depth-first sequential search hoisting can be implemented as a combination of graph-reduction and backtracking.

2 Hoisting

Let us ignore for a moment the mechanics by which reductions are to be shared and concentrate on the problem of deciding when a reduction is sharable. What we require is a characterisation of those parts of the search tree where the work involved in performing a reduction may be safely shared. The key to finding such a characterisation is the following important property of graph-reduction: once a redex appears in the graph, reducing it will always produce the same result (interpreted as meaning the effect on the local structure of the graph) no matter how long

the reduction is delayed or what other reductions are performed in the meantime. (This observation is just a rather strong Church-Rosser result; its strength derives from the ability to consider part of the graph in isolation and ignore changes wrought by other reductions to the rest of it.) The addition of 'logical variables' appears to spoil the situation since, as we have seen, the same expression may be reduced in different ways depending on how these variables become instantiated. For example:



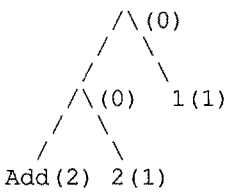


The property can be saved if we insist that a redex has not actually 'appeared' in the graph until all the variables that its reduction will dereference have been instantiated. (Or, put more simply, its not a redex until it can actually be reduced.) The, on any one search path, the previous result still holds since no instantiated variable can later change its value. (The problems caused by variables which 'change value' only occur when the system jumps from one search path to another.) In summary, once a redex has appeared in the program graph, nothing that happens lower in the search tree, whether the instantiation of further variables or reductions elsewhere in the graph, can ever change the effect of reducing it.

It was observed in the introduction that the problem of the interaction of lazy evaluation and logic programming is that delaying the reduction of an expression may require the same reduction sequence to be performed many times at different point in the search tree. The result of the previous paragraph guarantees that if the expression in question is a redex, then every attempt to reduce it will produce exactly the same change to the graph structure. It should now be obvious that any redex is a prime candidate for 'hoisting' - once it appears in the graph, the result of its eventual reduction is fixed. This being so, it should be safe to delay the reduction, perform it just once, and then share the result over some subtree of the search space.

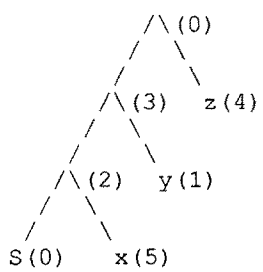
A problem now appears: how can the system econnomically determine, when it performs a reduction, the point in the search tree at which that redex first appeared (i.e. the first point at

which the reduction could have been performed) ? This question needs to be answered in order to determine the largest subtree of the seach space that can safely share the result of each reduction step. The solution is to associate an extra integer field - the tag - with each graph node. The tag of a node will be used to record the depth of the search tree when that node acquired its current value. When a node is first created, its tag will be initialised to the then current tree depth; whenever the node is updated , its tag is updated too. A redex appears in the graph (precisely) when all nodes which constitute its body receive values matching the requirements of some reduction rule. Since the tags indicate when each individual node received its current value, it is trivial to calculate the level of the tree at which the whole of the redex first appeared. For example, using brackets to indicate the tags attached to node, the following redex:



first appeared at level 2 of the graph (since 2 is the maximum i.e. most recent level of 0, 0, 1, 1 and 2). Here we have to include all five tags in the calculation; it is not enough to know that the arguments are 2 and 1, we also have to know that the operator is 'Add' and that it is applied to these two operands.

Deciding which tags to use in the calculation can be a rather subtle business; for example, the following redex first appeared at level 3 (not level 5) of the graph:



The reason for this is that to perform an S reduction, the system only needs to know that the S combinator is applied to three arguments (so level 3 is calculated as the maximum of 0, 2, 3 and 0). It doesn't need the actual values of the arguments to perform the reduction, so their tags are not included in the calculation. If the value of one of the arguments were to change, the reduction would still be correct - the new graph would simply contain the new value in the same place that the old graph had the old value. In operational terms, the value of a tag must be included in the

calculation if (and only if) the graph-reducer has to examine the value of the corresponding cell to perform the reduction.

It might seem that we have completely solved the problem posed at the beginning of the section: the result of reduction is sharable from the moment that the redex appears in the graph, which in turn can be determined by using the tags of the body of the redex. It would indeed be possible to implement a system on the above lines, and the system would perform some measure of hoisting, but it would be suboptimal. The problem is that not only can lazy evaluation delay the reduction of an expression once it has appeared in the program graph, it can also delay the moment at which the expression first appears at all (by delaying the reduction which gives rise to it). The key to getting the full benefit of hoisting is to realise that not only is it safe to share the result of performing a reduction from the point in the search tree at which the redex actually appeared but that sharing is also safe from the highest point at which the redex could (by using a different reduction strategy) have been made to appear. The advantage of this is that the higher in the search tree the result can be hoisted, the more widely it can be shared and the fewer times the reduction must be repeated.

The justification of 'full' hoisting again relies on the Church-Rosser property mentioned earlier. So far as the effect on the program graph is concerned, it doesn't matter whether a sequence of reductions is performed as soon as each is possible or only when the final result is required. Delaying reductions (as with lazy evaluation) may delay the appearance of the hoistable redex but can't alter its form and so can have no effect on the result of its eventual reduction.

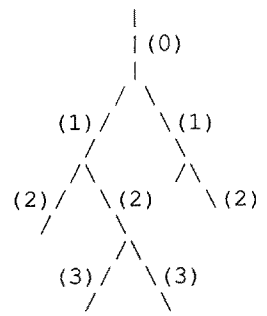
One previously solved problem now returns: how does the system, when performing a reduction, determine the highest point in the search tree at which a redex could have been produced ? The previous scheme won't do - it will only say when a redex was actually produced. What is needed is a subtle adjustment in the meaning of the tags: rather than record the point in the search tree at which each cell received its current value, each tag will now indicate the highest point in the tree at which the corresponding cell could possibly have been reduced to its current value. The highest point in the tree at which a redex 'could have appeared' can than be calculated from the tags of the nodes which constitute its value. Since a redex 'could have been reduced' as soon as it 'could have appeared', the same calculation will also give the correct tag for the result of the reduction. The simultaneous calculation of the new result and it tag, 'tagged reduction', is described in more detail in the next section.

3 An Implementation of Hoisting

This section describes the author's implementation of a hoisting system based on the combination of combinator graph-reduction [Turner 79] with depth-first sequential search. The particular choice of reduction system is not essential (for example, using supercombinators [Hughes 82] would be a straightforward extension); what is important is the way the system uses tags to calculate the highest point of the search tree at which each reduction could have been performed (section 3.1). The restriction to depth-first search is also important; this means that there is no need to copy the entire program graph when a choice-point is encountered. Instead, changes to the graph structure are logged and reversed on backtracking; the tags are used to decide which changes should be logged and when they should be reversed (section 3.2). Finally, a small change to the tagging scheme is introduced which result in a decrease in the number of store updates (but not reductions) required; this is described in section 3.3.

3.1 Tagged Combinator Reduction

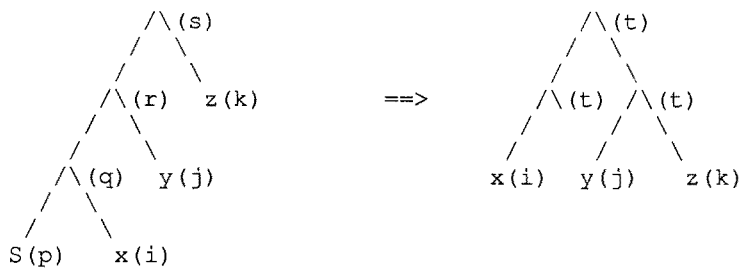
For each reduction that it performs, the graph-reducer has to calculate how widely the result may be shared i.e. how high in the search tree it may be hoisted. This section describes this calculation; the description is only concerned with what happens on one particular search path; the actions taken on backtracking are described in the next subsection. Provided we only consider one search path at a time, a single integer (the tag) will be enough to indicate unambiguously the highest point of the search tree to which any particular result may be hoisted. The tag used is the depth in the search tree; any two edges on the same search path will correspond to different tags.



Search Tree with edges labeled with tree-depth
(number of choice-points encountered)

Every graph node has a corresponding tag which indicates the highest point of the tree at which its current value could have been computed. When the program is first loaded, all the tags are set to zero (the original program is valid over the whole search space). When new graph nodes are created, or existing graph nodes are overwritten, as the result of a reduction, the new tag is calculated in a manner to be described shortly. Any graph node created by the system for any other reason (for example as the consequence of standardising clauses apart) is tagged with the then current depth in the search tree.

Whenever the system performs a reduction, it overwrites the root node of the redex with a new value; tagged reduction must calculate an appropriate tag to go with this value. The old and new tags will be compared to discover whether the old value should be saved and, if so, when it should be subsequently restored; this is described more fully in the next subsection. The rule used to calculate the new tag is very simple: the new tag should be the maximum of the tags of all the nodes whose values are examined to perform the reduction. This is a reflection of the fact that the reduction could have been performed (precisely) when all the values used to match the reduction rule could have become available. For example, the rule for the S combinator is:



t = max(p,q,r,s)

Note that it is important to include the tags of the internal application nodes in this calculation - its no use S being available for reduction if it has not been applied to three arguments. In contrast, the tags of the arguments themselves are not used in the calculation. This is because the precise values of x, y, and z are not needed to perform this reduction - it is valid so long as each of them is the same after the reduction as before it. The new tag is the maximum of the tag of the combinator itself and of the tags of the application nodes; the tags of the actual arguments are disregarded. This treatment of tags is typical of a combinator that simply embeds its arguments in a new applicative structure. The new tag is used to update the root node of the subgraph and to label any new graph nodes that are introduced (here the applications of x and y to z).

The treatment of combinators that examine the values of their arguments is different; here the tags of the arguments must be taken into consideration. For example, the rule for 'Add' (assuming the arguments are already in normal form) is:

$$u = max(p,q,r,s,t)$$

The reason for the difference is that the actual values of the arguments are used in the reduction; if either were to change then the result of the reductions would becone incorrect. (If the arguments are not already in normal form but have to be forced fisrt then it is important to use the post-forcing tags for the arguments rather than the pre-forcing ones as the tags may have been increased by the intervening reductions.) The general rule is that a tag is included in the calculation if (and only if) the graph-reducer has to examine the contents of the corresponding cell to perform the reduction. (Less operationally, this means that the tag of an argument is included in the calculation precisely when the reduction rule constrains the argument to be in some normal form.)

Dereferencing a 'logical variable' may be treated as if it were a special kind of reduction (in particular, it is analogous to I reduction). The result of 'reducing' a bound variable is the expression to which it is bound; the tag of the result is the maximum of the tag of the expression and the tree level at which the variable was bound (stored as the tag of the variable).

$$r = max(p,q)$$

Actually, this account is a little too simple, since if the expression (e) is not in head-normal form, then copying it may require it to be normalised several times, losing lazy evaluation. A similar problem arises with the reduction of an application of the I combinator; the soultion in each case is to force the expression before copying it.

3.2 Backtracking

The previous subsection described how tagged reduction maintains a tag for each graph node which specifies how 'sharable' its value is. The mechanism described there dealt with only one search path at a time; the problem is more general than this (if the system only had to try one search path, there would be no point in bothering with tags at all). The purpose of having tags is to determine when a result can be safely shared between different search paths; in terms of a depth-first implementation, this means that the tags will be used to decide which store updates must be trailed and when each such update must be reversed.

The graph structure is not duplicated when a choice-point is encountered, although a separate copy of it is logically required for each branch. Instead, a strategy is used that is similar to the treament of variable bindings in Prolog; changes are logged on a 'trail' and reversed on

backtracking (e.g. see Warren 77). The new trail (keeping this name for want of a better one) will store address / value / tag triples in a number of pools, with one pool for each level of the search tree. Each pool will contain a set of triples; when a update which needs trailing is performed, the address / old value / old tag triple is added to the pool indexed by the new tag. Later, when the system backtracks from level n+1 to level n, the n+1th pool is emptied and its contents used to roll back those changes to the graph structure that are no longer valid. (In general the system may have to backtrack by more than one level at a time. This is handled by processing each pool in turn, starting with the highest numbered.)

The algorithm for deciding when the old value of a cell should be saved is simple: compare the new tag with the old tag. If the new tag is greater, then the old value should be saved before the update is performed; if the two tags are equal (by construction, the new tag will never be the lesser) then no special action is required. The justification of this rule is that when the two tags are equal, both the old and new values of the node are valid in the same subtree of the search space, so that when backtracking invalidates the new value, there will be no point in reverting to the old one. By contrast, if the old tag is smaller, then the old value is valid in a larger (containing) subtree, so restoration will be worthwhile.

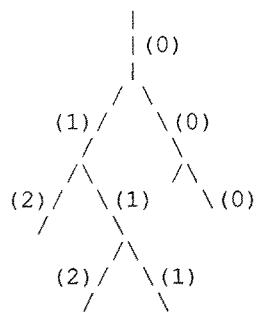
This doesn't mean that the old value should necessarily be restored when the system returns to the immediately previous choice-point; the entire purpose of maintaining the tags is to avoid this. A little thought will show when the old value should be restored - when the system backtracks out of the subtree in which the new value is valid. The new tag defines the subtree in which the new value is valid, so there is a simple test for this circumstance: the system should restore the old value when backtracking reduces the then current depth in the search tree to less

than the new tag. In effect, the trail is used as a stack of sets; any of the sets may be enlarged at any time, but they are discharged using a strictly last-in / first-out discipline.

We now have an implementation of hoisting for a sequential depth-first search strategy; only one refinement remains unexplained. By taking further advantage of the particular search strategy used (subtrees are explored left-to-right), the size of the trail (though not the number of reductions performed) can be reduced.

3.3 A Last-Call Optimisation

The tagging scheme based on tree-depth works well except in one respect. When the system returns to a choice-point after exploring the last alternative branch, it restores the program graph ot its state when the choice-point was first encountered. This is wasteful, since the next thing the system will do is backtrack further, which may discard or further restore the just-altered graph nodes. The purpose of saving the old values on the trail is so that they may be later reused; if there is no further use for a node, then there is no point in saving and later restoring it. This will be the case precisely when there are no unexplored alternatives to try between the choice-point indexed by the old tag and the choice-point indicated by the new tag. In this situation, backtracking past the latter choice-point (invalidating the new value of the node) will be followed, inexorably and immediately, by backtracking past the former (invalidating the old value of the node). It would be possible, but rather expensive, to check this condition whenever an update is attempted; a better solution is to alter the tagging scheme slightly:



New Tagging Scheme

The rule generating the new tags is that the tag is the depth in the search tree, excluding rightmost branches. In operational terms, the old scheme used the number of choice-points encountered along a path from the root while the new scheme uses the number of choice-points with unexplored alternatives. The effect of this change is to perform a kind of last-call

optimisation; the value of a node will not be trailed if it will never be needed again. (There is no change in the number of reductions required - both tagging schemes are equally good in this respect - the difference is in the number of store updates required.) This optimisation is particulary important in systems such as the author's which only permit binary disjunctions since the natural way to simulate multi-way choice on such a system (tail-nesting alternatives) would otherwise lead to an unnecessary proliferation of different tags and consequent waste of trail space.

4 Limitations

The scheme for hoisting described in the body of this paper has two important limitations which restrict its applicability. The first problem is that the particular tagging scheme described is only sufficient for a depth-first sequential system. A parallel system would require a more elaborate tag to indicate the subtree in which each reduction is valid. Also, a graph node would no longer be a simple memory cell; since its value may depend on which part of the search space is being explored, it must appear to have different values to different processes. (This can be acheived, for example, by using the tag of the subtree currently being explored to determine which value is appropriate.) These considerations do not make the use of hoisting impossible with a parallel implementation; they do, however, make it unduly expensive.

The second limitation is that the whole idea of hoisting assumes that the processes of reducing the program graph and exploring the search tree are distinct. In systems based on narrowing this is not the case; reduction, unification and search interact. A narrowing in one part of the program graph may (by instantiating a non-local variable) render impossible a potential narrowing in a completely different part of the program graph. This non-local interaction appears to prevent hoisting, since a potential narrowing may not, unlike a reduction, be valid in a whole subtree of the search space. The situation may not be as simple as this, however; for a language such as Darlington's [Darlington et al. 86], it may be expected that most narrowings are simply reductions (i.e. they are deterministic and instantiate no non-local variables). Moreover these reductions behave in just the way needed to make hoisting work; they can be implemented by graph-rewriting (since they are deterministic) and are valid in a whole subtree of the search space (since no non-local variables are instantiated). The 'genuine' narrowings (where an expression matches more than one rewrite rule) are responsible for the branching of the search space; these could be treated as the combination of a disjunction (forming a choice-point) and a reduction.

5 References

[Darlington et al. 86]
The Unification of Functional and Logic Languages
J. Darlington, A.J. Field and H. Pull
in 'Logic Programming: Functions, Relations and Equations'
ed. D. DeGroot and G. Lindstrom
Prentice-Hall 1986

[Goguen and Meseguer 84]
EQLOG: Equality, Types and Generic Modules for Logic Programming
J. Goguen and J. Meseguer
Journal of Logic Programming, 1984, Vol 1, No 2, pp 179-209

[Hughes 82]
Super-Combinators: A New Implementation Method for Applicative Languages
R. J. M. Hughes
1982 ACM Symposium on LISP and Functional Programming, Pittsburgh 1982

[Robinson and Sibert 82]
LOGLISP: Motivation, Design and Implementation
J.A. Robinson and E.E. Sibert
ed. K.L. Clark and S.-A. Tarnlund,
Academic Press 1982

[Turner 79]
A New Implementation Technique for Applicative Languages
D.A. Turner
Software Practice and Experience, 1979, Vol 9

[Warren 77]
Implementing PROLOG - Compiling Logic Programs
D.H.D. Warren
DAI Research Reports No 39,40, Edinburgh University 1977