Effective Function Cache Management for Incremental Attribute Evaluation

João Saraiva*, Matthijs Kuiper and Doaitse Swierstra

Department of Computer Science, University of Utrecht
P.O. Box 80.089, 3508 TB Utrecht, The Netherlands
email: {saraiva,kuiper,swierstra}@cs.ruu.nl
phone: +31 30 2536761

Abstract

This paper presents an effective strategy for function cache management in the context of incremental attribute evaluation. The evaluators we study consist of pure functions which are memoized to achieve incremental evaluation. To prevent unbounded growth of the function cache a strategy for managing this cache is needed. Our strategy only keeps in the cache the functions used in the reevaluation of the current input. This strategy has the following properties: it efficiently handles Higher-order Attribute Grammars, it works well when a sequence of modifications on the input exhibits some locality of work and measurements show that for sequences of typical tree transformations the evaluator consumes only a fraction of memory with our strategy than with a complete memoization, and is even slightly faster.

1 Introduction

This paper presents an effective strategy for function cache management in the context of incremental attribute evaluation. The attribute evaluators we study consist of pure, recursive functions. The incremental behaviour is obtained by memoizing function calls. To prevent unbounded growth of the cache a strategy to manage this cache is needed. We propose a simple strategy which only uses dynamic information and is based on the assumption that the locations of most changes in the input are strongly correlated. Our strategy only keeps in the cache entries for functions calls used in the reevaluation of the current input. It efficiently handles Higher-order Attribute Grammars and it works well when a sequence of modifications on the input exhibits some locality of work.

Several programming environments repeatedly apply a software tool to a sequence of similar inputs. Examples include compilers, interpreters and text processors whose inputs are usually incrementally modified text files. This motivates the development of incremental versions of such tools, i.e., tools which can efficiently recompute the result of a function when the input has changed only slightly. Attribute grammars are a suitable formalism for specifying such tools [Kas91]. From an attribute grammar incremental attribute evaluators can be automatically derived which evaluate any particular input [RTD83, Pen94]. Incremental attribute evaluation is particularly suited to implement language-based editors where the performance of the evaluator is primordial, since the feedback it provides may guide the user through the editing process.

The main question in the implementation of a function memoization scheme is which cache management to employ. The *complete memoization* of functions quickly decreases the performance of the evaluator due to the fast growth of the cache. We also show that traditional cache management algorithms are not suited for incremental attribute evaluation.

*On leave from the Department of Computer Science, University of Minho, Braga, Portugal.

We implemented our strategy in the LRC system, a generator of incremental attribute evaluators. Measurements show that for typical program transformations the evaluator is 7% faster with our strategy than with the complete memoization strategy, using only a fraction of the memory. These results also prove that it is possible to efficiently perform incremental attribute evaluation without consuming too much memory.

The rest of this paper is structured as follows: section 2 briefly presents *incremental attribute evaluation*. Section 3 discusses several visit-function cache updating strategies. Results of *complete memoization* are presented and we show why traditional cache management strategies do not work in incremental attribute evaluation. Section 4 presents an effective cache management strategy. In section 5 results of our strategy are presented. Section 6 briefly compares our strategy with other approaches. Section 7 presents the conclusions.

2 Incremental Attribute Evaluation

Higher-order attribute grammars (HAG) [VSK89] extend the classical attribute grammar formalism [AM91, Paa95]. In a HAG an attribute itself can have attributes (which can have attributes, etc.). Such an attribute is called an *attributable attribute* (an ata). Traditional attribute evaluators have problems with the incremental evaluation of ata's. When an ata changes one must recompute the attributes of the ata, unless one can reuse the values of the ata's attributes that were computed when decorating the ata. To achieve this reuse one must somehow compare the new value of the ata with its old one and determine which parts of the ata have remained unchanged. This is in general an expensive computation, because ata's can be large trees.

Our approach to the incremental evaluation of HAGs is as follows. First, we use functional evaluators that consist of pure functions, called *visit-functions*. Each function take as parameter a tree and some inherited attributes of the root node of the tree and returns a subset of the synthesized attributes of the tree. Second, attribute values are not stored in the nodes of the tree, they only exist as function arguments and function results. Third, the calls to visit-functions are memoized in a function cache. Each entry in a function cache stores the arguments and results of one function call.

These evaluators efficiently handle the incremental evaluation of ata's. When an ata changes only slightly, many of the function calls to compute the attributes of the changed ata are found in the cache. Our evaluator method has an additional advantage. When several equal trees (or ata's) are decorated with the same inherited attributes then only one instance is decorated. All other decorations result in a cache hit.

3 Visit-Function Cache Updating

We now return to the problem of updating the visit-function cache. When considering different update strategies the correctness of the attribute evaluator is not influenced, since it is always possible to remove any element from the cache. Efficiency is the only concern. So, the main issues of the cache updating algorithm are: which algorithm to use and when to invoke the removal algorithm.

We distinguish between *Attribute Grammar Dependent* and *Attribute Grammar Independent* caching managements strategies. The former includes all the strategies that use some knowledge about the AG to define which visit-functions may or may not be cached. It is possible to use static analysis in order to determine which visit-function calls may not be cached. For example, visit-functions applied to subtrees that are instances of productions whose right-hand side consists of terminal symbols only may not be cached. The latter consists of strategies that do not have any knowledge about the AG. This includes all the probabilistic cache management strategies.

Incremental attribute evaluation is particularly suited to implement language-based editors, i.e., for performing incremental semantic analysis of programs [RTD83]. The editor maintains the abstract syntax tree of the input and each *modification/edit action* is described as a tree transformation. A *single tree transformation* is denoted by $t' = t[r \leftarrow r']$, where t' is the tree resulting from replacing r by r' in t. This tree transformation is represented in figure 1. A tree transformation may cause attributes to have inconsistent values, not only in the subtree where the transformation was applied, but also elsewhere in the tree. The set of attribute instances that require a new value is denoted by Δ.

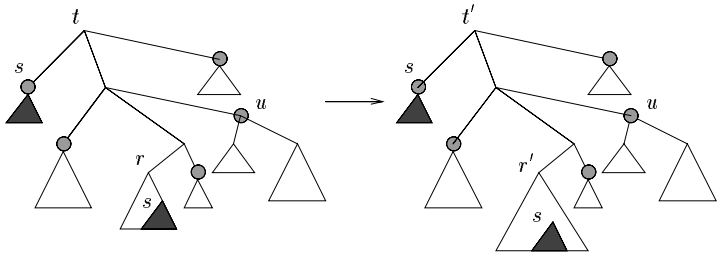

Figure 1: Tree transformation $t' = t[r \leftarrow r']$.

We define $A_{tree}(t)$ as the set of all attribute instances associated to the nodes of tree t. Each step of a *sequence of tree transformations* is a single tree transformation. We call a tree transformation $t' = t[r \leftarrow r']$ a *global tree transformation*, if $|A_{tree}(t')| \sim |\Delta|$, and a *local tree transformation* if $|A_{tree}(t')| \gg |\Delta|$. The assumption of the editing model is that almost all tree transformations are local.

3.1 Definition of Experiments

To measure the performance of incremental evaluators we used a PASCAL program and one of the incremental attribute evaluators produced by the LRC system. The incremental evaluator is a two-visit evaluator and performs semantic analysis, but no code generation. The PASCAL source program implements a text formatting program and has 19 procedures and consists of about 500 lines ($\approx$ 15000 characters). We have performed several modifications to this program. We consider two different sequences of *real program modifications/edit actions* in that program: one with 7 *global tree transformations* (GC) and another with 6 *local tree transformations* (LC). These two sequences of program modifications will be used throughout this text.

As global tree transformations we have considered the following tree changes: change the name of a global procedure and add a global variable (instruction `nvalX:integer;`). As local tree transformations we have considered three different changes in two different procedures. The two procedures are in different parts of the text, one is the second procedure in the source text (`procedure 2`) and the other is the nineth (`procedure 9`). We refer the procedures by the position they have in the text. The local changes were made in the following order: add a local variable `x:integer;` to the procedure , add a small statement `x := 0;` at the beginning of the procedure and add another statement `x := x + 1;` at the end of the procedure. These program modifications are sumarized in table 1.

Note that, since our approach allows multiple tree transformations, it is possible to perform all the edit actions before redecoration takes place. However, in order to analyse the incremental behaviour of our approach we perform a redecoration after each edit action.

3.2 Complete Memoization of Visit-Function Calls

	Global Changes		Local Changes
t	(pascal program)	t	(pascal program)
g^{i}	rename `procedure 17`	l^{i}	add `x:integer;` to `proc. 9`
g^{ii}	rename `procedure 19`	l^{ii}	add `x := 0;` to `proc. 9`
g^{iii}	rename `procedure 8`	l^{iii}	add `x := x + 1;` to `proc. 9`
g^{iv}	add global variable	l^{iv}	add `x:integer;` to `proc. 2`
g^{v}	add global variable	l^{v}	add `x := 0;` to `proc. 2`
g^{vi}	add global variable	l^{vi}	add `x := x + 1;` to `proc. 2`
g^{vii}	rename `procedure 17`		

Table 1: Global and Local Program Modifications.

With complete memoization of visit-function calls, each different visit-function call finds a place in the cache. The cache is organized as a hash array of collision lists. Locating a visit-function call v in this cache includes traversing a list comparing the function and the arguments of v within each element of the list. Thus, the cache overhead is strongly related too the size of these lists.

In table 2 we present results of processing the PASCAL program. We present the number of *misses* (i.e., the number of function calls computed) and *hits* (the number of function calls reused), the number of lookup operations performed (more precisely the number of visit-function call comparisons), the number of bytes used to store the visit-functions in the cache and the evaluator's runtime (in seconds).

Cache Strategy	Hash Array	Misses	Hits	Lookups	Cache Entries	Runtime
No Memoing (non incremental)	-	-	-	-	-	0.96
Full Memoing (I)	10007	8147	1542	5199	351516	1.07
Full Memoing (II)	1009	8147	1542	34931	351516	1.09
Δ Evaluation (memoing without hits)	1009	11391	0	55189	456552	1.25

Cache Strategy	Hash	Global Changes			Local Changes		
	Array	Misses	Hits	Runtime	Misses	Hits	Runtime
No Memoing (non incremental)	-	-	-	$0.96 + (7 * 0.96)$ $= 7.68$	-	-	$0.96 + (6 * 0.96)$ $= 6.72$
Full Memoing	1009	34523	12483	$1.09 + 5.02$ $= 6.11$	13686	2728	$1.09 + 0.74$ $= 1.83$

Table 2: Results of decoration from scratch and incremental evaluation processing the PASCAL program.

The first part of the table consists of processing the program from scratch using different configurations. In the first row we use a conventional attribute evaluator. In the second and third rows results using full memoing are presented. These two results were obtained using different configurations of the cache. The evaluator's performance is better in (I) since it performs fewer lookup operations, i.e., it performs five times fewer lookup operations and the running time is 2% better than in (II). In this text we will use and compare our results with the second configuration (II). This is only due to practical reasons, since we will use two sequences of small tree transformations as benchmarks and they would have a negligible impact in the growth of the cache in configuration (I). Nevertheless, the results will be the same in (I) if we perform more tree transformations, or when the size of the cache compared to the size of the tree becomes less favourable. The full memoing version is 11% slower than the exhaustive version (i.e., non incremental) due to the costs of memoization. The memoing version reuses 1542 visit-function calls ($\approx$ 20% of all the function calls)! This reuse is due to equivalent decorations of shared subtrees. For example, in a normal PASCAL program all occurrences of a variable in the same scope are evaluated in the same way. The same holds for statements like $i := i + 1$. The memoization overhead however is not compensated completely by this sharing, and still a 11% time increase is present. In the

fourth row we changed the system to "*really*" evaluate the Δ attribute instances, i.e., we do not reuse any visit-function and always store the new visit-function call in the cache. Obviously, the runtime increases and is 17% greater than the full memoing version. This is what we have gained with the sharing of equivalent subtrees.

In the second part of table 2 we present results of incremental evaluation. The runtimes presented are divided into two groups: the first one is the time obtained in the initial processing of t to reach the point where the transformations were applied and the time of the evaluation of the updates themselves. As expected, the incremental evaluator performs quite well with local tree transformations. We have a speedup of 7.8 considering only the incremental evaluation of the tree transformations (i.e., processing the modifications $l^i, \cdots, l^{vi}$) and a speedup of 3.7 with the decoration from scratch. As to be expected, global tree transformations have poor incremental behaviour (speedup of 1.3), since most of the attribute instances must be reevaluated and can not be reused. However, global tree transformations are intrinsically hard problems giving poor results in other approaches as well.

In figure 2 we present a graphical representation of the running times obtained for each tree transformation. The horizontal line EE represents the time the Exhaustive Evaluation takes when processing both sequences of tree transformations. Since the changes as such are minimal, all these times are almost equal.

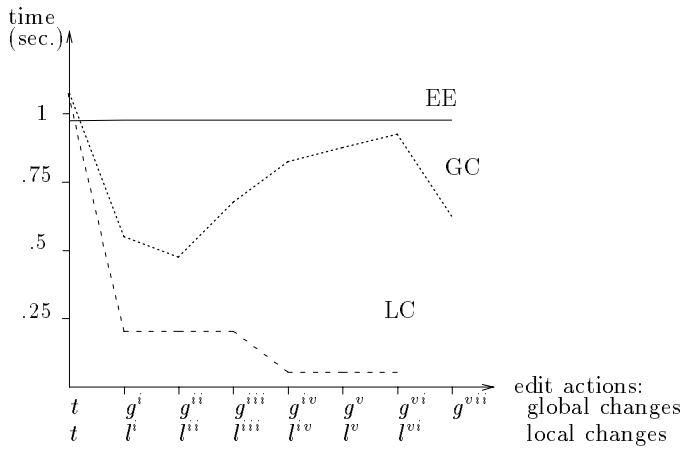

Figure 2: Exhaustive, global and local running times.

In the sequence of global changes the evaluator's performance decreases due to the growth of the visit-function cache. Each global tree transformation induces many new entries in the cache and consequently makes the lookup operations more and more expensive. Observe that, in the program modifications g^{iv}, g^v and g^{vi} the evaluator's performance decreases when processing similar transformations (add a global variable). Observe also that exactly the same modification (rename procedure 17) performed in g^i and g^{vii} is processed 10% slower in g^{vii} due to the growth of the cache.

In the sequence of local changes the cache's growth is much smaller, since each tree change only induces a few new entries in the cache. Its growth did not have (yet) any impact on the evaluator's running time. Note however, that the running time decreases after three edit actions. This happens because the modification l^{iv} is the first change in the second procedure and the path to root of this procedure is smaller, and fewer attribute instances must be reevaluated. The same change in the first procedure (l^i) is processed 3 times slower. Although the size of the visit-function cache did not influence the evaluator's performance in this sequence of local changes, its growth will eventually influence it if we perform more and more edit actions.

3.3 Why Traditional Cache Management Algorithms Do Not Work

One of the most commonly used cache replacement strategies is the *Least-Recently-Used* (LRU) algorithm. The algorithm is very simple and its update operation is fast. However, this simple algorithm *per se* is not suited for incremental attribute evaluation. We will explain why by using a simple example.

Without loss of generality, consider a two-pass attribute evaluator and assume that both passes require the computation of approximately the same number of visit-functions. Let us assume that the cache can only store 50% of the visit-functions needed to compute when processing a particular input from scratch.

After processing this input from scratch the cache stores the calls to visit-functions performed in the second pass of the evaluator. If we then change the input and call the evaluator it will have a poor incremental behaviour: the evaluator starts the decoration by performing its first pass and no visit-function call in this pass is found in the cache. Note that when the evaluator starts its second pass the cache holds the entries from the previous pass. The cache always contains the visit-functions applied in the "other" pass. The algorithm has been implemented in the LRC system and the results are presented in table 3. We used a fix bucket strategy, with 4 positions per bucket.

Evaluation Strategy	Global Changes			Local Changes		
	Misses	Hits	Runtime	Misses	Hits	Runtime
LRU	72085	6919	8.76	33012	2977	3.56

Table 3: Results of the LRU cache updating algorithm processing the changes of table 1.

As we can see the performance of the incremental evaluator decreases in both sequences of tree transformations. We can also observe that when processing the global transformations the incremental evaluator's performance is worse than the non-incremental one ($8.76 > 7.68$).

4 ULE Updating Strategy

Consider the single tree transformation $t' = t[r \leftarrow r']$ and let u be a subtree of t and t' that is not affected by the tree transformation, as represented in figure 1. The incremental evaluation of t' reuses the visit-function applied to the root of u. However, all the visit-function applied to the subtrees of u are not reused, since the hits occur at the top of the subtree and the evaluator skips the visits to u. More generally, all the visit-functions applied to subtrees not affected by a tree transformation are not reused in the incremental decoration of the resulting tree, with the exception being the visit-functions applied to their roots. In figure 1 the nodes which are parameters of the reused visit-functions are marked with a disk. Observe that, keeping all the other entries in the cache decreases its speed and the evaluator only reuses them if a future tree transformation affects those subtrees.

A possible cache strategy, which we called *Used in Last Evaluation* (ULE), keeps only in the cache the entries which were used in the redecoration of the current input. ULE assumes that if the current tree transformation did not affect a set of subtrees $\mathcal{U}$, then the following transformation will probably not affect $\mathcal{U}$, i.e., it assumes that a sequence of tree transformations exhibits locality. If the following tree transformation does not affect any subtree in $\mathcal{U}$, then this cache updating strategy is *optimal*. By *optimal* we mean that the cache contains the *minimum* number of visit-function entries that need to be applied to redecorate the subtrees in $\mathcal{U}$ (the visit-functions applied to the roots of the subtrees in $\mathcal{U}$) and ensures that the incremental evaluator runs in $\mathcal{O}(|\Delta \cup A_{path}(t', r')|)$ steps, where $A_{path}(t', r')$ is the set of attribute instances associated to the nodes in the path from the root of t' to the root of r', including the roots of t' and r'. The overhead due to having more visit-functions cached than needed in this case is zero.

Let $new(t', \mathcal{C})$ be the set of new visit-functions that need to be applied to compute the attribute instances in Δ using cache $\mathcal{C}$ and let $used(t', \mathcal{C})$ be the set of entries of $\mathcal{C}$ which are used when processing t'. Then the new cache obtained when processing t' is:

$$\mathcal{C}' = used(t', \mathcal{C}) \cup new(t', \mathcal{C})$$

The algorithm that determines $\mathcal{C}'$ is simple. When processing t', every time a hit in $\mathcal{C}$ occurs, the respective entry is copied from $\mathcal{C}$ to $\mathcal{C}'$, since it is an element of $used(t', \mathcal{C})$. When a miss occurs it means a new visit-function is being applied, so a new entry must be stored in $\mathcal{C}'$, since it is an element of $new(t', \mathcal{C})$. After decorating t' the cache $\mathcal{C}'$ contains the elements $used(t', \mathcal{C}) \cup new(t', \mathcal{C})$. In this algorithm, the size of the cache $\mathcal{C}'$ is always smaller than the cache obtained when processing the changed tree from scratch.

5 Performance of ULE Updating Strategy

We implemented the ULE cache updating strategy in the LRC system. Table 4 presents the results of processing the two sequences GC and LC of edit actions.

Cache Strategy	Global Changes			Local Changes		
	Misses	Hits	Runtime	Misses	Hits	Runtime
ULE-Cache	40523	12268	1.09 + 4.84 = 5.93	13848	2709	1.09 + 0.69 1.78

Table 4: Results of the ULE cache updating strategy.

Comparing these results with those obtained using the full memoing strategy (see table 2), the runtime decreases in both sequences of tree transformations, 4% and 3% in the global and local tree transformations respectively. The memory needed for caching the visit-functions, decreases in both strategies: after the sequence of global tree transformations, the size of the cache using the ULE algorithm is 7.2 times smaller than when using complete memoing. After processing the sequence of local tree transformations, the evaluator uses 8 times less memory! The total number of visit-function call comparisons performed during the lookup operations also decreases significantly, since the caches are smaller.

In figure 3 we present the running times (figure 3.*a*) and the memory used to store the visit-function calls (figure 3.*b*) during both sequences of tree transformations. The results obtained using the full memoing strategy are also presented (lines labelled with FM).

As we can see in the figure 3.*a*, in almost all the tree transformations the evaluator's performance using the ULE-cache is better than the full memoing version. As far as memory consumption is concerned, in both sequences of tree transformations the evaluator uses much less memory than when processing from scratch, and the memory used remains constant. Observe that the ULE-cache evaluator has the same runtime when processing g^i and g^{viii}, since the growth of the cache does not affect its performance, as in the full memoing strategy. In the sequence of local transformations the evaluator has also a better runtime when processing l^{ii}, l^{iii}, l^v and l^{vi} since the cache's size is smaller than the full memoing version and consequently the lookup operation is faster. However, the performance decreases when processing g^{iv} and l^{iv}. In these transformations the "user" jumped from one part of the tree to another. Using the ULE strategy, this transformation implies that the smallest subtree, which contains the changed subtree and whose visit-function calls applied to its root are stored in the cache, has to be redecorated from scratch. This problem, that we call *the jump edit action*, will be analysed in detail in the next section.

5.1 The Jump Edit Action

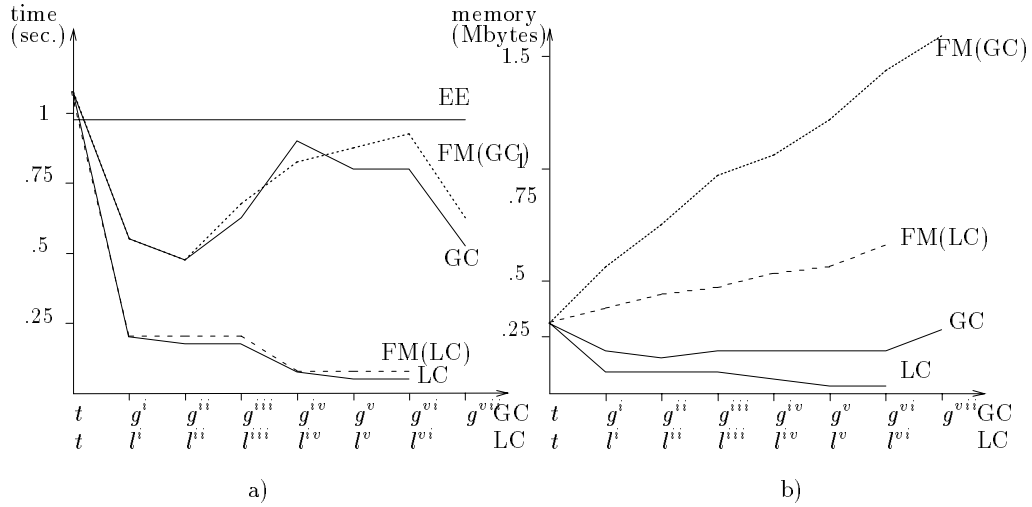

Figure 3: Running times and memory consumption of the ULE strategy.

We defined a *jump edit action* as two consecutive transformations in different parts of the tree. For example, when an user makes some changes in one procedure in a program and after that he changes another procedure. We analyse the behaviour of our approach in this situation. Consider the finite number of tree transformations:

$$t_0 \longrightarrow t_1 \longrightarrow \cdots \longrightarrow t_n$$

Suppose that each t_i is obtained from t_{i-1}, with $1 \leq i \leq n$, by a subtree replacement in the same subtree of t_0. Let r be this subtree. Suppose also that the only attribute instances affected by the tree transformations are $\Delta = A_{tree}(r)$ and those on the path to the root. Using the ULE cache strategy, the incremental evaluator runs in less than or equal to $\mathcal{O}(|\Delta \cup A_{path}(t_i, r)|)$ steps, i.e., with its best performance. The cache $\mathcal{C}_i$ has the minimal number of entries needed to guarantee that performance of the evaluator, i.e., it is *specialised* to perform incremental decorations in that part of the tree. So, the overhead of the cache is minimal.

Consider now that after those n tree transformations the user *jumps* to another part of the tree and replaces the subtree q', as represented in figure 4.

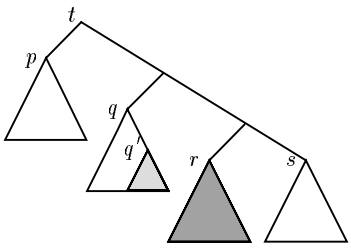

Figure 4: The jump edit action: after a sequence of transformations in r the "user" changes a subtree of q.

Let q be the smallest subtree of t which contains q' and whose visit-function calls that have these subtrees as arguments are stored in cache $\mathcal{C}$. Note that the subtree q was not affected by the previous tree transformations. So, only the visit-functions applied to its root are cached. In this case, the ULE strategy runs in less than or equal to $\mathcal{O}(|\Delta \cup A_{tree}(q) \cup A_{path}(t, q)|)$ steps. It needs to redecorate the entire subtree q from scratch. This happens when processing the change l^{iv} in LC (see figure 3).

Let us now assume that the user returns to subtree r. Then the subtrees r and s must be redecorate from scratch, since only the visit-function applied to their parent are in the cache. This happens when the user jumps to a deepest level of the tree. Note that in the sequence of local transformations if we perform exactly the same edit actions but starting by changing procedure 2 and after that jumping to procedure 9, then the impact in the evaluator's performance is larger. In this case a large subtree containing all the procedures defined after procedure number 2 must be redecorated from scratch. In table 5 we present the results obtained.

Cache Strategy	Local Changes		
	Misses	Hits	Runtime
ULE-Cache	18189	3183	2.53

Table 5: Results of the ULE strategy processing LC in reverse order.

As expected the runtime of the evaluator decreases significantly. Nevertheless, this performance of the evaluator when processing some jump edit actions is not surprising, since our strategy assumes that there exists some locality in the tree transformations. The jump edit action can be efficiently handled in our approach if we keep in the cache a subset of visit-functions applied in the subtrees not affected by the tree transformations. That is, if we memoize visit-functions applied to subtrees of p, q and s in the previous example. Determining which entries must remain always in the cache might be done by performing a static analysis of the AG. However, since we are only using dynamic information we can use the size of the subtree, where the visit-function is applied, to decide whether we cache the visit-function or not. When performing incremental attribute evaluation it is better to memoize *large* subtrees, since a hit in the respective entry represents that visits to *large* subtrees can be skipped. In table 6 we present the results obtained when processing the reverse of the sequence of local tree transformations and keeping in the cache the visit-functions applied to those subtrees which are greater at least 10% of the input tree.

Cache Strategy	Local Changes		
	Misses	Hits	Runtime
Ule-Cache	7389	1047	1.71

Table 6: Results of the ULE strategy keeping large subtrees in the cache.

Besides solving the problem of the jump edit action, this strategy also increases the performance of the evaluator. The running time of the evaluator decreased 33% when compared with the pure ULE-cache strategy (see table 5). This evaluator is also 7% faster than the full memoing evaluator processing the same edit actions. This improvement of the performance is due to have a smaller and consequently faster cache.

6 Brief Comparison with Other Approaches

There has been a lot of research on incremental attribute evaluation, ever since Reps [RTD83] first used attribute grammars for performing incremental semantic analysis of programs. The goals of the following approaches are very similar to ours, but they use a different methodology.

Change Propagation - Reps [RTD83] proposes a simple approach to incremental attribute evaluation which involves propagating changes through the attributed tree. It uses a dependency graph to ensure that an attribute instance is evaluated only after all the instances it depends on have been assigned their final values. It needs also to keep track of which attributes changed value. Furthermore, this approach only allows one single edit action before decoration takes place. Yeh and Kastens [YK88] propose an approach that eliminates the need of the dependency graph and that allows multiple tree transformations. Both approaches have an optimal running time processing OAG's. However, they consume too much memory and become extremely complicated and far from when processing HAG's [TC90].

Function Caching - Pugh [PT89] caches the semantic function calls in order to achieve incrementality. Our approach is more efficient since a cache hit for a visit-function call means that an entire visit to an arbitrarily large tree can be skipped. Observe also that a visit-function may return the results of several semantic functions at same time.

7 Conclusions

This report presented a strategy for effective function cache management in the context of attribute evaluation. The proposed function cache management strategy only keeps in the cache entries for function calls used in the reevaluation of the input. This strategy solves the problem of the infinite growth of the function cache. Moreover, it also improves the performance of the incremental evaluator, since the resulting cache is smaller and consequently faster. We also presented several optimisations to solve the problem of handling jump edit actions.

For experimental purposes, those strategies have been implemented in the Lrc system. Measurements show that for a small sequence of tree transformations our evaluator is 7% faster than the full memoing evaluator. As far as memory consumption is concerned, our evaluator consumes only a fraction of the memory and its consumption remains constant during the sequences of tree transformations performed.

References

[AM91] H. Alblas and B. Melichar, editors. *International Summer School on Attribute Grammars, Applications and Systems*, volume 545 of *LNCS*. Springer-Verlag, 1991.

[Kas91] Uwe Kastens. Attribute grammars as a specification method. In H. Alblas and B. Melichar, editors, *International Summer School on Attribute Grammars, Applications and Systems*, volume 545 of *LNCS*, pages 16–47. Springer-Verlag, 1991.

[Paa95] Jukka Paakki. Attribute grammar paradigms - a high-level methodology in language implementation. *ACM Computing Surveys*, 27(2):196–255, June 1995.

[Pen94] Maarten Pennings. *Generating Incremental Evaluators*. PhD thesis, Utrecht University, November 1994. `ftp://ftp.cs.ruu.nl/pub/RUU/CS/phdtheses/Pennings/`.

[PSV92] Maarten Pennings, Doaitse Swierstra, and Harald Vogt. Using cached functions and constructors for incremental attribute evaluation. In M. Bruynooghe and M. Wirsing, editors, *Programming Language Implementation and Logic Programming*, volume 631 of *LNCS*, pages 130–144. Springer-Verlag, 1992.

[PT89] William Pugh and Tim Teitelbaum. Incremental computation via function caching. In *16th Annual ACM Symposium on Principles of Programming Languages*, volume 1, pages 315–328. ACM, January 1989.

[RTD83] Thomas Reps, Tim Teitelbaum, and Alan Demers. Incremental context-dependent analysis for language-based editors. *ACM Transactions on Programming Languages and Systems*, 5(3):449–477, July 1983.

[TC90] Tim Teitelbaum and Richard Chapman. Higher-order attribute grammars and editing environments. In *ACM SIGPLAN'90 Conference on Principles of Programming Languages*, volume 25, pages 197–208. ACM, June 1990.

[VSK89] Harald Vogt, Doaitse Swierstra, and Matthijs Kuiper. Higher order attribute grammars. In *ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*, volume 24, pages 131–145. ACM, July 1989.

[VSK91] Harald Vogt, Doaitse Swierstra, and Matthijs Kuiper. Efficient incremental evaluation of higher order attribute grammars. In J. Maluszynki and M. Wirsing, editors, *Programming Language Implementation and Logic Programming*, volume 528 of *LNCS*, pages 231–242. Springer-Verlag, 1991.

[YK88] Dashing Yeh and Uwe Kastens. Improvements of an incremental evaluation algorithm for ordered attribute grammars. *ACM - SIGPLAN Notices*, 23(12):45–50, December 1988.