Strictification of Computations on Trees

João Saraiva[*], Doaitse Swierstra and Matthijs Kuiper

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

An attribute grammars describes a computation over a recursive data structure (syntax tree). The corresponding evaluator can be directly encoded as a set of lazy evaluated functions. In this paper we show how this set may be converted into a larger set of strict functions and a collection of new data types. We call this process, which is based on a global data flow analysis, *strictification*. The resulting set of small functions and the new data types are amenable to further analysis and optimization. Especially the *elimination* transformation leads to very efficient programs, both in time and space, and which are much more suited for incremental (*i.e.* memoized) and parallel evaluation.

Introduction

Functional programs written in a lazy language can be surprisingly short and elegant; unfortunately this elegance comes at a price, since in general they are also expensive to evaluate, due to the overhead associated with the demand driven execution model and the associated dynamic scheduling of computations. A lot of research has been done on *strictness analysis*, in order to move some of the decisions to be taken about the evaluation order from the dynamic to the static realm.

In this paper we present techniques, some of which originate from the field of attribute grammars, which enables us to transform a specific class of functions into a larger set of mutually recursive functions, which however are all strict in their all arguments. In this way the inherently lazy part of the computation becomes explicitly represented in the program, and the resulting program can be evaluated in a data-driven instead of a demand driven way.

After this general transformation has been performed we are left with a large set of mutually recursive small strict functions, each of which usually does not perform much work. This resulting set can however, since the overall computation has been unravelled, be analysed for often occurring special cases,

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

and be subjected to further transformations in order to perform work which can be done statically, and to make the remaining run-time data structures as lean as possible.

The work described here originates from our work on constructing an incremental evaluator for higher-order attribute grammars [VSK91]; the traditional approaches for evaluating attribute grammars –*i.e.* the decoration of an abstract syntax tree by means of a generated tree-walking automaton– proved to be no longer applicable due to the fact that the abstract syntax tree (i.e. the argument controlling the overall computation) is no longer an invariant data structure in the computation but depends also on intermediate results of the overall computation [Pen94, CP96]. Since our new evaluation model only uses the generated strict functions we can rely on efficient function caching for achieving incremental evaluation.

We show that even when one is not aiming at incremental evaluation, the transformations performed are useful: since strictness is improved it becomes easier to perform parallel evaluation, and the transformed programs in general execute faster and in less space.

In this paper we show, by means of a running example, how the actual transformation is being performed, and what final optimisations become enabled as a result of this. The techniques shown have all been implemented and form part of the current Lrc-attribute grammar based program synthesizer.

1 The Class of Functions Considered

The class of functions we are considering are the so-called circular functional programs, as introduced in [Bir84]. Although such programs have proved to be surprisingly hard to explain to the average functional programmer, they actually become a lot simpler to understand and design if one considers them as the representation in a lazy functional language of an attribute grammar evaluator [KS87, Joh87]. More recently this class of functions became known as catamorphisms, although the context in which such catamorphisms got their name (*i.e.* the formal derivation of functional programs) usually restricts itself to recursive functions which take only one parameter of one specific data type as an argument [BdM96]. In attribute grammar terms one would say that one is dealing with a grammar which has only a single non-terminal.

Attribute grammars are usually thought to be implemented by using so-called tree-walk evaluators which evaluate and store attribute values in the nodes of a recursive data structure (the abstract syntax tree) and thus evaluate all the values in the data-flow graph induced y the rules for assigning values to its attributes. In its functional programming counterpart lazy evaluation is used to schedule the computations of values dynamically. In our bootstrapped Lrc-system it appears that a corresponding tree-walk evaluator may visit a specific node up to 13 times. The equivalent circular program would have many function calls with arguments depending on results of the same call.

As an example of such a circular program we present a analyser for a small language, called Block, which consists of programs of the following form:

```
blk main : (   use y;
               blk f : (dcl w; use y; use f)
               dcl x; dcl x; dcl y; use w
           )
```

Such programs describe the basic block-structure found in many formal languages, with the peculiarity however that declarations of identifiers may also occur after their first use. Furthermore, an identifier from a global scope is visible in a local scope only if is not hidden by an a declarations with a same identifier in a local scope. In a block an identifier may be declared at most once.

According to these rules the above program contains two errors: at the outer level the variable x has been declared twice and the use of the variable w has no binding occurrence at all.

The abstract grammar of the language is described by the following recursive data type definitions:

```
data Program   =   P     Items
data Items     =   NilItems
               |   Cons_Items   Item  Items
data Item      =   Block    Name  Items
               |   Decl    Name
               |   Use    Name
data Name      =   Ident    String
```

Figure 1 presents the lazy program for the Block language written in Haskell.

```
evalProgram   ( P  items)  = ( errors)
  where   ( dclo,  errors) =  evalItems    items  ( NilEnv) 0  dclo

evalItems    ( NilItems )  dcli  lev  env   = ( dcli , [])

evalItems    ( Cons_Items  item  items)  dcli  lev  env   = ( dclo2 ,  errors )
  where   ( dclo1,  errors1) =  evalItem    item   dcli  lev  env
          ( dclo2,  errors2) =  evalItems    items   dclo1   lev  env
          ( errors) =  errors1 ++  errors2

evalItem    ( Block  name  items)  dcli  lev  env   = ( dclo1 ,  errors1)
  where   ( error) = lookup_env  name  lev  dcli
          ( dclo1) = cons_env  name  lev  dcli
          ( dclo2,  errors2) =  evalItems    items    env ( lev + 1)  dclo2
          ( errors1) =  error ++  errors2

evalItem    ( Decl  name)  dcli  lev  env   = ( dclo ,  errors)
  where   ( errors) = lookup_env  name  lev  dcli
          ( dclo) = cons_env  name  lev  dcli

evalItem    ( Use  name)  dcli  lev  env   = ( dcli ,  errors)
  where   ( errors) = is_member_env  name  env
```

Figure 1: The lazy program which performs the static semantic analysis of a Block program.

The functions `cons_env`, `lookup_env` and `is_member_env` perform the usual operations in an environment. The environment is implemented as a binary search tree.

This program is a *circular* program: *one of the results of a function call is also one of its arguments.* Thus, lazy evaluation is essential.

Because we allow an *use-before-declare* discipline, a conventional imperative implementation of the required analysis naturally leads to a program which traverses the abstract syntax tree twice: once for processing the declarations of identifiers and constructing an environment and once for processing the uses of identifiers using the computed environment to check for the use of non-declared names. The uniqueness of names is detected in the first traversal: for each newly encountered declaration it is checked whether that identifier has already been declared at the same level. In this case an error message is computed. Of course that identifier might have been declared at a global level. Thus we need to distinguish between identifiers declared at different levels. We introduce a variable *lev* used to compute the level of a block. The environment is a partial function mapping an identifier (*name*) to its level of declaration (*lev*).

As a consequence, semantic errors resulting from duplicate definitions are computed during the first traversal and errors resulting from missing declarations in the second one. In order to make the problem more interesting, and to demonstrate our techniques, we require that the error messages produced in both traversals are to be merged in order to generate a list of errors which follows the sequential structure of the program.

Observe that the environment computed for a block and used for processing the use-occurrences is the global environment for its nested blocks. Thus, only during the second traversal of a block (*i.e.*, after collecting all its declarations) the program actually begins the traversals of its nested blocks; as a consequence the computations related to first and second traversals are intermingled.

Let us analyse in detail the most intricate function alternative of the above program: the function alternative applied to BLOCK nodes. Figure 2 shows the induced dependency graph, which follow from a flow analysis of the total program.

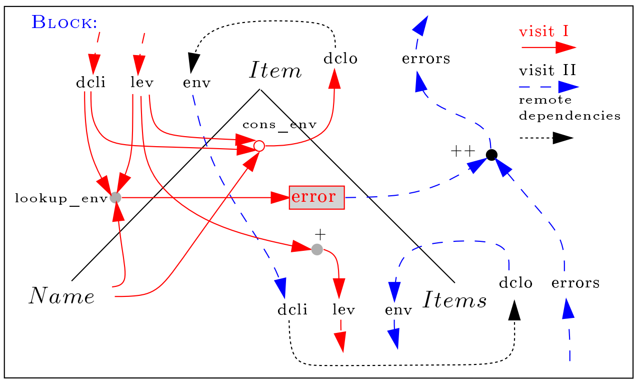


Figure 2: Dependency graph of function `evalItem` alternative BLOCK.

The local variable `error` is computed taking the environment which consists of the global environment and the local declarations processed thus far (`dcli`). This value is concatenated to the list of errors generated by processing the body of the block. Because of lazy evaluation the computation of the list of errors is suspended until it is really necessary. So, the computation of the local variable `error` is suspended too. As a consequence, the declarations collected in `dcli` must be kept in memory until the computation of the variable `error` is finally enforced! Thus a huge closure must be kept until finally the list of errors is required, *i.e.* until the end of the evaluation.

Figure 3 shows a profile of the use of heap memory of the lazy program processing a BLOCK source text. The profile was produced by the `nhc` heap profiler [RR96].

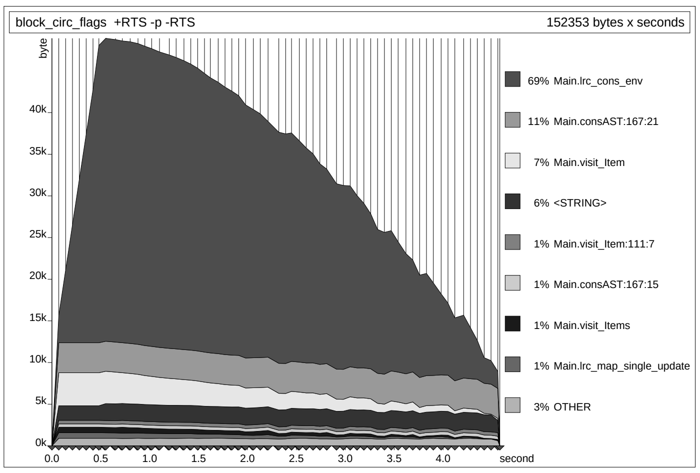


Figure 3: Lazy accumulation of suspended computations.

This profile shows a heap that grows quickly to a peak size of $\approx 50kbyte$ diminishing afterwards. The unnecessary accumulation of the environment is causing this huge memory consumption (*lrc_cons_env*, which constructs the environment, produces 70% of the total heap space). The heap grows linearly in the number of blocks. Although this profile was obtained using the HASKELL strict flag, the lazy accumulation is not avoided.

2 Strictification

In this section we present our technique to transform circular programs into strict ones. We use well known attribute grammar static analysis techniques [Alb91, Paa95] in order to break up the circularities into a large set of small multiple traversal functions.

The transformation is performed in two steps: based on a global analysis of the program we find a partial order in which the arguments, local values and results of a function can be computed. Then we derive a set of *visit sequences* for each constructor of the circular program. Such visit sequences statically associate a fixed sequence of moves and computations with each constructor of the abstract syntax tree (productions of the grammar). The number of visits to a data type is statically fixed too. A *visit subsequence* is associated to each of the visits. It specifies a subset of arguments/results of the original circular function which are needed/computed in a particular visit.

Secondly we transform these visit sequences into *visit functions*. To combine the effects of the individual traversals we introduce *visit trees*. These visist trees are a small adaption of our previously introduced *bindings* [PSV92]. Visit trees are needed to *pass* values that are computed in one traversal and that are used in a subsequent one[1]. We call such dependencies between traversals *inter-traversal dependencies*. In the circular program these dependencies are handle by the lazy evaluation machinery.

2.1 Computing Visit Sequences

In this section, we show how to derive *visit sequences* from the circular programs. We use Kastens' ordered scheduling algorithm to derive the visit sequences [Kas80]. Our presentation in this paper is necessarily short. Full definitions can be found in [Kas80] and [Pen94] (page 122).

We start by briefly describing visit sequences since they are the result of this step. A visit sequence describes the operations which must be performed by a tree walk automata when visiting a node: `eval` x computes value x, `visit` (n, v) descends to node n for the v^{th} time and `suspend` v returns from the v^{th} visit to the father node.

The visit sequences are obtained in four steps: first a *dependency graph* for all the circular function alternatives is computed. Once the dependencies are computed we can detect whether an argument in a call depends on a result of the same call. Such dependencies force additional traversals to the data types where such functions are applied. Figure 2 shows the dependency graph derived from function `eval`**Item** alternative Block. This graph illustrates the dependency forcing additional visits to data type $Items$: the argument env of function `eval`**Items** depends on its result $dclo$.

Secondly, a *linearization* of those dependency graphs is computed. This step computes a linear (total) order for all the circular functions in which the arguments and results must be computed. In our example the order computed for `eval`**Items** is $lev, dcli, dclo, env, errors$.

Next, a set of *partitions* $[\pi_1, \pi_2, \ldots, \pi_n]$ is computed for each data type. A partition $\pi_v(t) = ([arg], [res])$ defines which arguments $[arg]$ are needed to compute the results $[res]$ during visit v to data type t. Since we are converting a partial order into a total one we have here a considerable freedom, which may be exploited when computing the partitions. Here we use Kastens' algorithm which maximizes the size of the partitions and minimizes the number of visits. The resulting partitions for our example are:

$$\begin{array}{lcl}
\pi_1(\mathit{Program}) & = & ([], [errors]) \\
\pi_1(\mathit{Items}, \mathit{Item}) & = & ([lev, dcli], [dclo]) \\
\pi_2(\mathit{Items}, \mathit{Item}) & = & ([env], [errors])
\end{array}$$

Finally, the visit (sub)sequences are produced according to the previous partitions. For each partition $\pi_v(t) = ([arg], [res])$ and for each constructor of type t a visit subsequence is produced which defines the sequence of instructions that are needed in order to compute the results $[res]$ using the arguments $[arg]$.

[1]This happens for example when a circular program computes some information which is assigned to nodes in the tree and that is used later on.

Figure 4 shows the two visit subsequences derived for data constructor BLOCK. The data type is used to distinguish different occurrences of a variable in the visit subsequences.

```
$visit\ 1 : \pi_1 = ([dcli, lev], [dclo])$
    eval      $Items.lev$    $(= lev + 1)$
    eval      error          (= lookup_env ...)
    eval      $dclo$         (= cons_env ...)
    suspend   1

$visit\ 2 : \pi_2 = ([env], [errors])$
    eval      $Items.dcli$   $(= env)$
    visit     $(Items, 1)$
    eval      $Items.env$    $(= Items.dclo)$
    visit     $(Items, 2)$
    eval      $errors$       (= error ++ $Items.errors$)
    suspend   2
```

Figure 4: Visit sequences for data constructor BLOCK derived from the circular program

Observe that, the variable error is evaluated in the first visit subsequence and it is used in the second subsequence. Such dependencies are discussed in next section.

2.2 Computing Visit Functions

In imperative programming the implementation of visit sequences is straightforward: values needed in later visits are stored in the nodes of the original tree. Thus no problem arises when a later visit uses values computed in previous ones. In a purely functional setting values cannot be stored in the original tree. As a consequence, values needed in future traversals must be explicitly passed around.

We propose a new technique to transform visit sequences into pure functions. Our approach mimics the imperative approach: values needed later are stored in a new tree, called a *visit tree*. Such values have to be preserved from the traversal that creates them until the last traversal that uses them. Each traversal builds a new visit tree with the additional values stored in the nodes for the next traversal. We will show that as a result of optimizations this tree may be quite different from the original tree. The functions that perform the subsequent traversal find the values they need either in their arguments or in the tree nodes, exactly as in the imperative approach. A set of visit tree types is defined, one per traversal. Subtrees that are not needed in future traversals are *discarded* from the visit trees concerned. As result any data no longer needed is indeed no longer referenced.

This technique requires a static analysis of the visit sequences in order to determine the set of inter-traversal dependencies *itd*. A visit sequence of a data constructor C has an inter-traversal dependency between visit v and w, with $v < w$, if there is a value x that is computed in subsequence v of C which is used in subsequence w of C. We denote this by ($C^{v \rightarrow w}$,[x]).

It determines also which subtrees can be discarded from the subsequent visit trees. A new set of visit tree types is computed according to the inter-traversal dependencies and discarded subtrees determined previously. Finally, it computes the set of visit functions corresponding to these new visit tree types. Figure 5 shows the four stages of our technique.

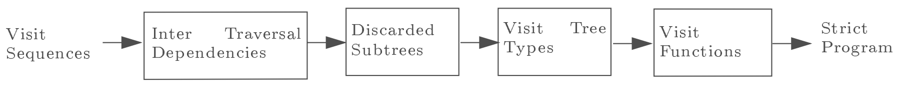


Figure 5: Stages need to produce the visit functions.

The *type of the visit tree* for the first traversal is the type of the original tree. The visit trees for the following traversals are defined as:

$$\begin{array}{lcl} \text{visit tree for traversal } n & = & \text{original tree} \\ & - & \text{discarded subtrees} \\ & + & \text{values needed in traversals} \geq n \end{array}$$

The mapping of the visit sequences to pure *visit functions* is as follows: for each data type T and for each visit $v \in [1..n]$, with n the number of visits of T, a visit function $\texttt{eval}_v\mathbf{T}$ is produced. Such function $\texttt{eval}_v\mathbf{T}$ has the visit tree T_v as its first parameter and the arguments of partition $\pi_v(T)$ as further parameters. The visit function returns the visit tree for next traversal $v+1$ and the results of the corresponding partition. The visit function of the last visit n of T does not return any new visit tree. The signature of these functions are:

$$\begin{array}{l} \texttt{eval}_v\mathbf{T} :: T_v \rightarrow arg_v \rightarrow (T_{v+1}, res_v) \\ \texttt{eval}_n\mathbf{T} :: T_n \rightarrow arg_n \rightarrow res_n \end{array}$$

with $1 \leq v < n$ and arg_v (res_v) is the set of argument (result) types of visit v. Each visit function $\texttt{eval}_v\mathbf{T}$ is defined by giving a function definition for each constructor of type T. These definitions are selected by using pattern matching on the respective data constructor. The body of a function definition is derived from the respective visit subsequence.

Let us analyse the visit sequences presented in figure 4 and derive the corresponding visit functions. We start by computing the inter-traversal dependencies. Observe that the local variable $error$ and the level lev are evaluated in the first visit subsequence. They are however needed in the second one: $error$ to evaluate $Items.errors$, *i.e.* the complete list of errors, and lev to compute the level of nested blocks (see figure 4). The set of inter-traversal dependencies derived is:

$$\begin{array}{ll} itd = \{ & (\textsc{Block}^{1\rightarrow 2}, [Items.lev, error]), \\ & (\textsc{Decl}^{1\rightarrow 2}, [error])\} \end{array}$$

Next the set of discarded subtrees is computed. Subtrees of type $Name$ of declarations and blocks are used in the first traversal only. They are used in constructing the environment and detecting duplicate declarations (see figure 2). Thus they can be discarded from the second traversal.

After that the set of visit tree types is derived. The type for the first visit is the type of the original tree:

$$\texttt{type}\ Items_1 \quad = \quad Items$$

The visit tree type for the second traversal is:

$$\begin{array}{lcl} \texttt{data}\ Item_2 & = & \textsc{Use}_2\ \ Name \\ & | & \textsc{Decl}_2\ \ Error \\ & | & \textsc{Block}_2\ \ Int\ \ Error\ Items_1 \\ \texttt{data}\ Items_2 & = & \textsc{NilItems}_2 \\ & | & \textsc{Cons\_Items}_2\ \ Item_2\ Items_2 \end{array}$$

Finally, the visit functions based on visit trees are derived according to visit sequences and the newly derived data types. The circularity of function $\texttt{eval}\textbf{Items}$ is broken into two traversal functions, both strict in their arguments. The first traversal function computes the collection of declarations used as the initial environment in the second traversal. Inter-traversal dependencies values are stored in the visit tree during the first traversal. In the second traversal such values are ready to be used.

$$
\begin{array}{l}
\texttt{eval}_1\textbf{Item}\quad (\ \textsc{Block}_1\ \ name\ \ \text{items\_1}\ )\ \ dcli\ \ lev\ = ((\ \textsc{Block}_2\ \ (\ lev+1)\ \ \texttt{error}\ \ \text{items\_1}\ ),\ dclo)\\
\quad \texttt{where}\quad (\ \texttt{error}\ ) = \texttt{lookup\_env}\ \ name\ \ lev\ \ dcli\\
\qquad\qquad\ \ (\ dclo) = \texttt{cons\_env}\ \ name\ \ lev\ \ dcli
\end{array}
$$

$$
\begin{array}{l}
\texttt{eval}_2\textbf{Item}\quad (\ \textsc{Block}_2\ \ lev\ \ \texttt{error}\ \ \text{items\_1}\ )\ \ env\ = (\ errors)\\
\quad \texttt{where}\quad (\boxed{\text{items\_2}},\ dclo_2) = \texttt{eval}_1\textbf{items}\ \ \text{items\_1}\ \ env\ \ lev\\
\qquad\qquad\ \ (\ errors_2) = \texttt{eval}_2\textbf{items}\ \ \boxed{\text{items\_2}}\ \ dclo_2\\
\qquad\qquad\ \ (\ errors) = \texttt{error}\ \texttt{++}\ \ errors_2
\end{array}
$$

lev , error are computed in first visit and used in the second one

The function $\texttt{eval}_2\textbf{Item}$ contains the two calls to the visit functions which evaluate the body of a block. The first call returns the visit tree $\boxed{\text{items\_2}}$ which is used as the first argument of the second call.

Observe that, the circular function call of the lazy program (see figure 1)

$$(\ dclo_2,\ \ errors_2) = \texttt{eval}\textbf{Items}\quad items\quad env\ (\ lev+1)\ \ dclo_2$$

is now unravelled and it is evaluated strictly by the two traversal functions called during the second visit to blocks as follows:

$$(\ errors_2) = \texttt{uncurry}\ \ \texttt{eval}_2\textbf{Items}\quad (\texttt{eval}_1\textbf{Items}\ \ \text{items\_1}\quad env\ \ lev)$$

the increment of the *lev* is performed in the first visit (see visit functions above).

3 Elimination of Redundant Computations

After strictification the resulting programs are constitute of a large set of small strict functions, each of them working on a particular data type. Such small functions provide ample opportunity to be further transformed in order to eliminate unnecessary steps in the computation.

A source of inefficiency in the resulting strict programs is the existence of *redundant data constructors* (nodes) in the visit trees. These nodes lost their semantic meaning on a specific traversal, *i.e.* no useful computation is being performed during a traversal. Such nodes can be eliminated, yielding smaller visit trees and consequently more efficient programs. We consider three kind of redundant nodes: *syntactic*, *copy* and *unit* nodes.

- *syntactic Nodes*: Auxiliary data types which are used when structuring the circular program. They are used only to improve comprehensibility and are not needed during computation. Those data types can be detected statically and unfolded when producing the strict program.

- *Copy Nodes*: Data constructors whose associated visit subsequence for a specific visit consists of *copy rules* only. Copy rules are equations of the form $\alpha \equiv id\,\beta$. These nodes can be detected and eliminated statically. The corresponding data constructor and visit function definition are eliminated from the strict program. An example of a dependency graph of such a visit (sub)sequences is given in figure 6.

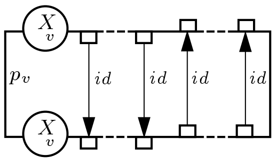


Figure 6: Copy Terms: Non crossing copy rules only

- *Unit Nodes*: Data constructors which only store values for subsequent traversals. Such terms are identified statically. Their elimination however is performed dynamically: if values stored in an instance of such constructors are the unit elements of the functions going to be applied to them then this node does not contribute to the program semantics. So, they are not included in the visit trees. Figure 7 characterizes such terms.

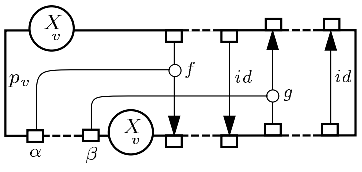


Figure 7: Unit Nodes: f and g are functions, α and β values computed in an earlier traversal and $f\ x\ \alpha\ \equiv\ x$ and $g\ y\ \beta\ \equiv\ y$.

Let us return to our example. Consider the data types that describe the abstract grammar of Block presented in section 1. The data type *Items* was only used to structure the circular program: neither the lazy nor the strict program computes any value when traversing nodes of that type. Arguments and results are just being passed on. A more efficient program is obtained if the definition of data type *Item* is unfolded:

```
data Items2  =  NilItems2
             |  Cons_Use2     Name Items2
             |  Cons_Decl2    Error Items2
             |  Cons_Block2   Int  Error Items1 Items2
```

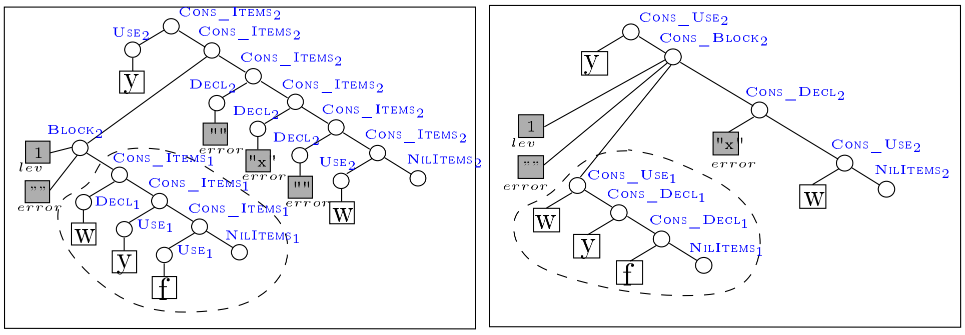
Consider now the declaration nodes. In the first visit an error is produced if the same identifier has been declared before at the same level. In the second visit that error is *concatenated* to the list of errors. However, nodes where no error occurred do not contribute to that list since they contain the unit element of list concatenation: the empty list. They are *unit nodes* and can be removed from the visit tree. The respective function definition is statically modified in order to identify if the *error* contains the unit element. In that case no node is dynamically created. The function definition applied to the unfolded data types and which perform the unit element detection is presented next.

```
eval₁Items    ( Cons_Decl₁  name  items_1 )  dcli  lev   = (vistree₂, dclo)
  where    ( error) = lookup_env  name  lev  dcli
           ( dcli₂) = cons_env  name  lev  dcli
           ( dclo ,  items_2 ) =  eval₁Items    items_1   dcli₂  lev
             vistree₂     | error == []     =  items_2
                          | otherwise       = ( Cons_Decl₂   error  items_2 )
```

4 Results of Strictification

This section presents the results of the strict program processing Block texts. We consider two strict programs: one without the elimination optimization and the other with elimination.

Figure 8(a) shows the visit tree computed in the first traversal (and used in the second traversal) of the strict program without the elimination optimization. The source text is the example sentence. Figure 8(b) shows the visit tree computed in the first traversal but now using the strict program obtained with the elimination. In both visit trees the block's bodies contain the original abstract syntax tree (marked with dotted circles). Remember that only in the second traversal the evaluator descends to the nested blocks.



a) Strict Program without Elimination   b) Strict Program with Elimination

Figure 8: Visit trees for the second traversal without elimination (a) and with elimination (b)

Using the elimination optimizations all the declaration nodes are removed from the tree during the first visit, except the one that really contains a semantic error. This node stores the only error detected in the first traversal that must be preserved for the second traversal. Elimination yields smaller trees and as a result decreases memory consumption and execution time.

Figure 9 presents a heap usage profile of both strict programs. The source Block text is the same used when producing the profile of the lazy program (figure 3).

The live heap remains constant during the computation, rarely exceeding $10kbyte$! The lazy accumulation of the circular program is completely vanished. The performance of the strict programs is better than the circular one. The profiles of both strict programs have a similar shape. This is expected since they traverse the tree only twice. In a program with more traversals we would have

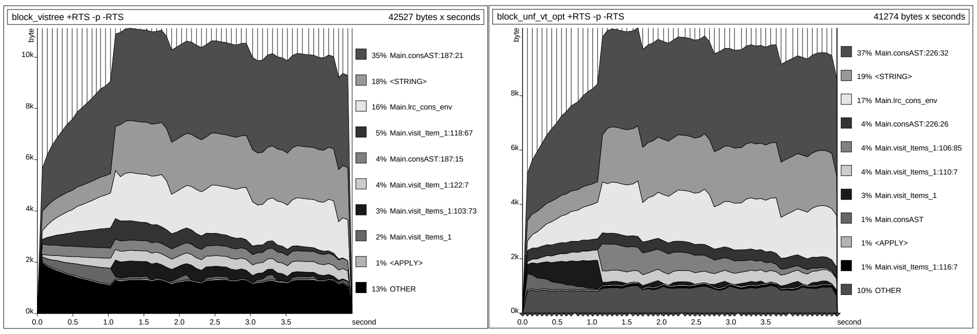


Figure 9: Heap profile of the strict programs: left graphic without elimination and right with elimination.

seen larger differences in performance. Nevertheless, already using this small example we can clearly see that elimination produces more efficient memory consumption programs.

5 Applications

In this section we consider three areas where our techniques can be applied: the incremental and parallel computation of circular programs and the implementation of *higher order attribute grammars*.

5.1 Incremental Computation

Function memoization is a efficient technique to implement incremental computations [PT89]. Circular programs relying on lazy evaluation cannot use standard memoization techniques to achieve such incremental behaviour, since comparing unevaluated arguments is not likely to work in our situation. After the strictification process however function memoization can be applied straightforwardly.

Furthermore, elimination improves the the evaluator's incremental behaviour since fewer redundant function are being memoized, thus avoiding filling up the memoing tables with useless entries, *i.e.* the redundant functions that would be applied if nodes not be eliminated. Moreover, the strict functions contains exactly the data going to be used in the call and the data is in a canonical form. Thus no non-used data which might disturb the memoization process is passed to a visit function.

The size of visit trees give a better estimative of the computations needed to be performed on a traversal than using a total tree. This more accurate information can be used dynamically to decide if a function call is to be memoized or not [Pug88].

5.2 Parallel Computation

Implicit parallelism exhibit by multiple traversal algorithms may be hidden by the circular definitions. In a circular program it may be impossible to discover which traversal functions may be computed in parallel. Consider for example the circular program presented in figure 1. Our transformation automatically extracts and identifies such potential parallelism. Moreover, since the argument of the visit functions are visit trees and not the complete tree communication overhead is decreased when in a distributed system.

Consider the visit function computed in the second visit to a block. The two visit functions that analyse the body of a block can be computed in parallel with the function that traverses the rest of the tree. Thus, the bodies of all the blocks can be computed in parallel. This parallelism was hidden in the circular program.

```
eval2Items   ( Cons_Block2  lev  error  items_ body items)  env  = ( errors)
  where   ( dclo,  items_2 ) =  eval1Items   Items_ body env  lev
          ( errors2) =  eval2Items   items_2    dclo
          ( error_aux) =  error  errors2
          ( errors3) =  eval2Items   Items3   env
          ( errors) =  error_aux  errors3
```

The static detection of independent visit functions per se can lead to a too fine grain of parallelism [Jou91]. The size of a visit tree can be used dynamically to decide whether two independent function calls may be computed in parallel or not.

5.3 Implementation of Attribute Grammars

Our work was developed in the context of incremental attribute evaluation of Higher Order Attribute Grammars. Traditional techniques for incremental attribute evaluations which are based on change propagation [RTD83] do not handle higher order attribute grammars efficiently [CP96]. An efficient and elegant incremental attribute evaluator can be implemented through visit function memoization [CP96].

After all the transformations have been performed the resulting programs show a striking resemblance to code which one would have written by hand. it is not an unlikely situation that during an analysis a data structure is constructed which, after the analysis has been completed, needs some small final touch-ups. In our model you find this phenomenon back by having a first traversal generating the fragments of the large subtree, in a small visit tree, which is then transformed into a final value by a small traversal of that tree.

We have implemented a system, called Lrc, which accepts higher order attribute grammars and produce pure functional attribute evaluators. Incrementallity is achieved through memoization. Information about a very experimental version of Lrc system is available on the internet at

`http://www.cs.ruu.nl/people/matthys/lrc_html`

6 Conclusions

This paper presented a technique to transform circular programs to strict ones. Visit trees were introduced to handle inter-traversal dependencies. Such visit trees can be further optimized in order to eliminate redundant computations and increase programs' efficiency. Elimination yields smaller trees, thus decreasing memory consumption and execution time of the strict programs. The first results show that strict programs are considerably more efficient than the lazy programs.

References

[Alb91] Henk Alblas. Introduction to attribute grammars. In H. Alblas and B. Melichar, editors, *International Summer School on Attribute Grammars, Applications and Systems*, volume 545 of *LNCS*, pages 1–15. Springer-Verlag, 1991.

[BdM96] Richard Bird and Oegerikus de Moor. *Algebra of programming*, volume 100 of *Prentice-Hall Inernational Series in Computer Science*. Prentice-Hall, 1996.

[Bir84] R. S. Bird. Using circular programs to eliminate multiple traversals of data. *Acta Informatica*, (21):239–250, January 1984.

[CP96] Alan Carle and Lori Pollock. On the optimality of change propagation for incremental evaluation of hierarchical attribute grammars. *ACM Transactions on Programming Languages and Systems*, 18(1):16–29, January 1996.

[Joh87] Thomas Johnsson. Attribute grammars as a functional programming paradigm. In G. Kahn, editor, *Functional Programming Languages and Computer Architecture*, volume 274 of *LNCS*, pages 154–173. Springer-Verlag, September 1987.

[Jou91] Martin Jourdan. A survey of parallel attribute evaluation methods. In H. Alblas and B. Melichar, editors, *International Summer School on Attribute Grammars, Applications and Systems*, volume 545 of *LNCS*, pages 234–255. Springer-Verlag, 1991.

[Kas80] Uwe Kastens. Ordered attribute grammars. *Acta Informatica*, 13:229–256, 1980.

[KS87] Matthijs Kuiper and Doaitse Swierstra. Using attribute grammars to derive efficient functional programs. In *Computing Science in the Netherlands CSN'87*, November 1987. `ftp://ftp.cs.ruu.nl/pub/RUU/CS/techreps/CS-1986/1986-16.ps.gz`.

[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.

[Pug88] William Pugh. An improved replacement strategy for function caching. In *ACM Conference on Lisp and Functional Programming*, volume 7, pages 269–276. ACM, july 1988.

[RR96] Colin Runciman and Niklas Röjemo. Heap profiling for space efficiency. In John Launchbury, Erik Meijer, and Tim Sheard, editors, *Second International School on Advanced Functional Programming*, volume 1129 of *LNCS*, pages 159–183. Springer-Verlag, 1996.

[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.

[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.