Strictification of Lazy Functions

João Saraiva*, Doaitse Swierstra, Matthijs Kuiper and Maarten Pennings

Department of Computer Science, University of Utrecht
P.O. Box 80.089, 3508 TB Utrecht, The Netherlands
emails: {saraiva,kuiper,swierstra}@cs.ruu.nl
penningm@natlab.research.philips.com

Abstract

This papers describes a transformation from lazy functions into efficient non-lazy ones. The functions we study perform multiple traversals over a data structure. Our transformation performs a global analysis of the calling structure of a set of mutually recursive lazy-functions in order to transform them into sets of functions which must be called in sequence. Many of the resulting functions can be eliminated by the optimizations presented in this paper. We present measurements that show that transformed and optimized functions allow efficient incremental execution. The paper contains examples that were automatically constructed with a generator of incremental functional programs.

1 Introduction

One of the more intricate parts of the world of functional programming deals with the construction of so-called circular programs, with the program *repmin* of Bird acting as the canonical representative of this class [Bir84]. For almost everyone, when first introduced to such programs, it takes a while before (s)he actually is convinced that such a program may work indeed as claimed by their authors. It is the use of lazy evaluation which does the trick.

For those who have not seen such programs before, we present here again the example of [Bir84]. The goal of this program is to compute a binary tree with integers in the leaves, which has the same shape as the argument tree, but with all leaves replaced by the minimal value in the original tree.

$$
\begin{array}{l}
repmin\ t = \ r \\
\quad \texttt{where}\ \ (r, m) = repmin'\ t\ m \\
\qquad\qquad repmin'\ (Fork\ l\ r)\ m \ = \ (Fork\ lr\ rr), lm\ \texttt{min}\ rm) \\
\qquad\qquad\quad \texttt{where}\ \ (lr, lm) \ = \ repmin'\ lt \\
\qquad\qquad\qquad\qquad\ \ (rr, rm) \ = \ repmin'\ rt \\
\qquad\qquad repmin'\ (Leaf\ v)\ m \ = \ (Leaf\ m, v)
\end{array}
$$

The curious thing here is that part of the result of the initial call to $repmin'$, i.e. m, is also passed as an argument to $repmin'$.

For those who are well acquainted with attribute grammars such dependencies come as no surprise, they are well used to thinking in terms of setting up equations between attributes,

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

and letting the system worry about the order in which the computations are actually scheduled. It has furthermore been noticed by [KS87, Joh87] that there exists a direct translation from attribute grammars into this class of circular programs. Those acquainted with catamorphisms will furthermore recognise a catamorphims which returns a higher order type in the above program [MFP91].

In recent years we have been interested in the incremental evaluation of (higher order) attribute grammars [VSK91, SV91, PSV92, Pen94]. The main aspect of the method being used is that attribute grammars are transformed into large sets of mutually recursive *strict* functions, the calls to which are being cached in order to avoid unnecessary reevaluations. The fact that the arguments to these functions can be evaluated before the call, without changing the termination properties, makes the resulting evaluators efficient and simple, and thus there is no need anymore to implement the much more complicated, and much less efficient, caching of lazy functions [Hug85].

In this paper we present the techniques we present our techniques in the setting of functional languages and transformations thereof. One might foresee that these methods find their way into a compiler, as has happened with techniques like deforestation and virtual data structures [Wad90, SdM93].

It is relatively easy to see how the program $repmin'$ might be splitted into two functions: one that computes the minimal value of all the leaves, and one that constructs the resulting tree. In this way the circular dependency is broken. In section 2 we introduce an example program that can not be splitted so easily because the circular dependencies not only occur at the top level call, but also inside the recursive calls.

In section 3 we show how, after analysing the dependencies between the function arguments, the functions can be transformed into sequences of functions, which have to be called one after the other for each subtree. For this transformation to work we introduce so-called *bindings*: a kind of explicitly constructed environment part of the closure that would otherwise have been constructed.

When looking at the set of functions constructed in this way we will see that many functions actually do very little work. Thus we show in section 4 how by unfolding some definitions, several of the generated functions become superfluous, and may be removed from the program.

In section 5 we finally show that certain nodes of the trees have lost their semantic meaning for certain calls of functions; in order to get a better caching behaviour the program may again be transformed so that such nodes will no longer have to occur in the trees. Finally we have reached the situation in which only relevant calls remain, thus guaranteeing an optimal use of the function cache, in which we have a canonical representation for each subcomputation. We demonstrate the effectiveness of our approach by presenting a few results of re-evaluations of functions generated by our system.

In section 6 we finally present some conclusions.

2 The Example

Our approach will be introduced through an example. We present a program that generates code for a simple language called BLOCK. This language deals with the *scope* of variables in a *block structured language*. A variable from a global scope is visible in a local scope only if is not hidden by a variable with the same name in the local scope. A program in BLOCK is a list of *declarations* (such as `dcl` a), *statements* (such as `use` a) and *blocks*, where a block is also a list of declarations, statements and nested blocks. A concrete sentence in this language looks as follows:

```
(use x; dcl x; dcl w;
  (use z; use y ; dcl x; dcl z; use x);
dcl y; use y)
```

where blocks are surrounded by parenthesis. The local usage of x refers to the local declaration and the global usage of x refers to the global declaration. The usage of y and z refer to their only declaration (global and local respectively).

The structure of the language leads naturally to a two pass compiler: the first pass collects all the declarations of blocks, and the second one actually uses the constructed environment.

The objective of the program is to generate code for a BLOCK program. The code consists of a sequence of three types of instructions: `Enter`(d) which enters a block with d local declarations; `Leave`(d) which leaves a block with d local declarations and `Access`(l, d) which accesses the d'th variable (*displacement*) of the l'th nested block (*level*). The code for example above is:

```
Enter 3, Access 1 0,
  Enter 2, Access 2 1, Access 1 2, Access 2 0, Leave 2,
Access 1 2, Leave 3
```

The abstract syntax of the language BLOCK is defined by the following recursive data definitions:

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

and the data type for the environment is:

`data` *Env*	=	**Consenv** *Name Level Displacement Env*
	|	**Emptyenv**

The abstract syntax of the generated code is defined by:

`type` *Code*	=	[*Instr*]	`data` *Instr*	=	**Access** *Level Displacement*
`type` *Level*	=	*Int*		|	**Enter** *Int*
`type` *Displacement*	=	*Int*		|	**Leave** *Int*

Implementing the compiler in a functional language (*e.g.* GOFER) is "straightforward". The complete program is presented in figure 1.

Where the function `access` takes as arguments the environment and the variable being accessed and returns the appropriate instruction.

Observe that this program is a *circular* program (in the sense of [Bir84]), that is, one of the results of a function call is also one of its arguments. This occurs twice in the program: in function **travProgram** with argument/result *dclo* and in the alternative **Block** of function

```
travProgram :: Program → Code
travProgram (P items) = code
  where (code, dclo, nodo) = travItems items 0 0 Emptyenv dclo

travItems :: Items → Level → Displacement → Env → Env → (Code, Env, Displacement)
travItems (Cons_Items item items) lev nodi dcli env = (code1++code2, dclo2, nodo2)
  where (code1, dclo1, nodo1) = travItem item lev nodi dcli env
        (code2, dclo2, nodo2) = travItems items lev nodo1 dclo1 env

travItems (NilItems) lev nodi dcli env = ([], dcli, nodi)


travItem :: Item → Level → Displacement → Env → Env → (Code, Env, Displacement)
travItem (Decl name) lev nodi dcli env = ([], Consenv name lev nodi dcli, nodi + 1)

travItem (Use name) lev nodi dcli env = (code, dcli, nodi)
  where code = [access env name]

travItem (Block items) lev nodi dcli env = ([Enter nodo] ++ code1 ++ [Leave nodo], dcli, nodi)
  where (code1, dclo, nodo) = travItems items (lev + 1) 0 env dclo
```

Figure 1: Circular Program.

travItem . The dependency graph induced by the latter is presented in figure 2.

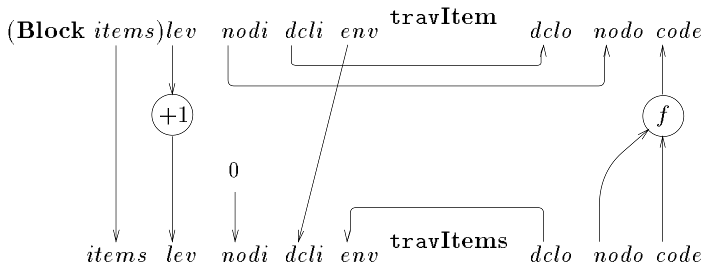

Figure 2: Dependency graph induced by the function applied to **Block**.

Circular programs require lazy evaluation in order to handle such *circular* dependencies. In the next section we present our approach which avoids the need for this lazy evaluation.

3 The strict version

This section presents a strict version of the example program and explains how this version was obtained.

Each function of the set of circular functions is translated into one or more functions and some additional data types. We call the functions in the translated version sub-traversals because each one corresponds to a part of the original function. The basic idea in obtaining the *sub*-traversals is to partition the results of an original function into groups and to associate a new function with each group of results. In our example function **travItem** is split into two functions. The first function computes *dclo* and *nod* and the second computes *code*.

The construction of the sub-traversals exploits the particular structure of the functions in the example program. The example is special in that all functions have as first parameter a recursive data type and the recursion structure of the functions "follows" the structure of their first argument. Each function has several definitions that are distinguished by the constructor used in the first argument. The sub-traversals in the transformed program have as first parameter the same recursive data type as the original traversal.

The strict version is obtained in 5 steps.

1. analysis of dependencies among arguments and results of traversal functions
2. linearization of dependencies
3. definition of interfaces of sub-traversals
4. dealing with interdependencies between sub-traversals
5. generation of GOFER code

The *first* step determines how arguments and results of functions may depend upon each other, and is based on well known attribute grammar analysis techniques [Kas80]. This step builds two kinds of dependencies: those among the arguments and results of one function and those among all arguments and results of all calls occurring in a function body. Once the dependencies are known we can see whether an argument in a call depends on a result in the same call. This is illustrated in figure 2 by the dependency from the result *dclo* of **travItems** to the argument *env*.

The *second* step linearizes the dependencies. This step works on all dependency graphs of all function definitions. This step computes an order in which the arguments and results must be computed. The computed order is compatible with the dependencies from the first step, meaning that if $x \rightarrow y$ is a dependency from step 1 then x comes before y in the order computed in this step. In our example the order computed for **travItems** is *lev*, *nodi*, *dcli*, *dclo*, *env*, *nodo*, *code*.

The *third* step determines the interfaces of the sub-traversals. Arguments and results of a function are grouped into a sequence of ([argument],[result]) pairs. Each pair will give rise to one sub-traversal function, that computes the [result] from [argument]. For example, for function **travItems** , the constructed sequence consists of two pairs: first ([*lev*, *nodi*, *dcli*], [*dclo*]) and then ([*env*],[*nodo*, *code*]).

The *fourth* step deals with interdependencies among sub-traversals. An example of such an interdependency is the case where an argument of a function is used in more than one sub-traversal. For example consider the parameter *lev* in figure 2. This parameter is needed in first and second sub-traversal of that function. Since for all calls of **travItem** the interface should be the same, the parameter *lev* is passed to first sub-traversal corresponding to **travItem**: it may be that the alternative **Decl** has to be dealt with, in which *lev* is used to extend the list of declarations. If the alternative however is a **Block** the parameter *lev* is actually only used in the second sub-traversal of **travItem**, since that is the one which contains the call to the first sub-traversal for the **Block**.

To handle such interdependencies among sub-traversals this step introduces *bindings*. Bindings contain precisely those values that should be passed from one sub-traversal to the next. They are *constructed* in one traversal —values that should be passed are included— and *destructed* in the next so that the values are available for use. Bindings are terms with a structure much like the tree that is being traversed.

Suppose we have n different, mutually recursive, data types T_i ($1 \leq i \leq n$) and that we have t_i traversals on data type T_i. These traversals have the following types, where $1 \leq i \leq n$ and $1 \leq v \leq t_i$

$$\texttt{trav}_v T_i :: T_i \ldots inputs \ldots \rightarrow \langle \ldots outputs \ldots \rangle$$

These functions are augmented with bindings, a change that is reflected in their types. The function for sub-traversal v for data type T_i may return binding information for all subsequent sub-traversals, that is to say for sub-traversals w with $v+1 \leq w \leq t_i$. Besides that, this function may also be passed bindings from all its predecessors, i.e. traversals w with $1 \leq w \leq v-1$. In other words, the types are changed to:

$$\begin{array}{lll} \texttt{trav}_v T_i & :: & T_i \ldots inputs \ldots \rightarrow T_i^{1\rightarrow v} \rightarrow T_i^{2\rightarrow v} \rightarrow \ldots \rightarrow T_i^{v-1\rightarrow v} \\ & \rightarrow & \langle \ldots outputs \ldots, T_i^{v\rightarrow v+1}, T_i^{v\rightarrow v+2}, \cdots, T_i^{v\rightarrow t_i} \rangle \end{array} \quad ,$$

where $T_i^{v\rightarrow w}$ is the type of the binding computed during traversal v to T_i and used during traversal w to T_i.

We must now determine the constructors for the bindings $T_i^{v\rightarrow w}$. Suppose that there are n_i constructors $\mathbf{con}_{i,k}$ (where $1 \leq k \leq n_i$) on type T_i. With each of these constructors we associate a set of so called binding constructors $\mathbf{con}_{i,k}^{v\rightarrow w}$ on $T_i^{v\rightarrow w}$ with defining traversal v ($1 \leq v \leq t_i - 1$) and using traversal w ($v+1 \leq w \leq t_i$). In other words, for any type T_i, $1 \leq i \leq n$, we have $\frac{1}{2} t_i(t_i - 1)$ associated binding types $T_i^{v\rightarrow w}$, $1 \leq v < w \leq t_i$, each with n_i binding constructors:

$$\begin{array}{lll} \texttt{data}\ T_i^{v\rightarrow w} & = & \mathbf{con}_{i,1}^{v\rightarrow w} \ldots \\ & | & \mathbf{con}_{i,2}^{v\rightarrow w} \ldots \\ & \vdots & \\ & | & \mathbf{con}_{i,n_i}^{v\rightarrow w} \ldots \quad . \end{array}$$

Now that we have set up a framework for bindings, we are finally able to discuss the *shape* of the binding constructors. A binding constructor $\mathbf{con}_{i,k}^{v\rightarrow w}$ binds objects that are computed in traversal v of an instance of constructor $\mathbf{con}_{i,k}$ and that are used in traversal w of that same node. Binding constructors bind two kinds of objects namely local results and arguments to be used later and *bindings for sons*. In our example, a **Block** node puts a *lev* in a binding, and a **Cons_Items** node puts the *binding* for each son in a binding.

Bindings may be empty, and most of them probably are. That is to say, the mutual recursive definitions of the bindings is such that for a particular binding, the (infinite) set of all producible terms contains no term that binds a (non-binding) value. The fourth step determines which bindings are guaranteed to be empty and these are not added to the sub-traversals. The bindings added in this step require the definition of extra data types. The bindings induced by our example are:

$$\begin{array}{llll} \texttt{data}\ \mathrm{Item}^{1\rightarrow 2} & = & \mathbf{Use}^{1\rightarrow 2} & \\ & | & \mathbf{Decl}^{1\rightarrow 2} & \\ & | & \mathbf{Block}^{1\rightarrow 2} \quad Int & \texttt{-- type(lev)} \\ \texttt{data}\ \mathrm{Items}^{1\rightarrow 2} & = & \mathbf{NilItems}^{1\rightarrow 2} \quad Int & \texttt{-- type(nodo)} \\ & | & \mathbf{Cons\_Items}^{1\rightarrow 2} \quad \mathrm{Item}^{1\rightarrow 2}\ \mathrm{Items}^{1\rightarrow 2} & \end{array}$$

The *fifth* and final step constructs the GOFER code for the sub-traversals. The code of the transformed program is presented in figure 3.

$$
\begin{array}{l}
\texttt{trav}_1\textbf{Program}\ (\textbf{P}\ \mathit{Items} = (\mathit{code})\\
\quad\texttt{where}\quad \begin{array}{lcl}(\mathit{dclo}, \mathrm{Items}^{1\to 2}) & = & \texttt{trav}_1\textbf{Items}\ \mathit{Items}\ \textbf{Emptyenv}\ 0\ 0\\ (\mathit{code}, \mathit{nodo}) & = & \texttt{trav}_2\textbf{Items}\ \mathit{Items}\ \mathit{dclo}\ \mathrm{Items}^{1\to 2}\end{array}\\[2ex]
\texttt{trav}_1\textbf{Items}\ (\textbf{Cons\_Items}\ \mathit{Item}\ \mathit{Items})\ \mathit{dcli}\ \mathit{nodi}\ \mathit{lev} = (\mathit{dclo}_2, \textbf{Cons\_Items}^{1\to 2}\ \mathrm{Item}^{1\to 2}\ \mathrm{Items}^{1\to 2})\\
\quad\texttt{where}\quad \begin{array}{lcl}(\mathit{dclo}_1, \mathit{nodo}, \mathrm{Item}^{1\to 2}) & = & \texttt{trav}_1\textbf{Item}\ \mathit{Item}\ \mathit{dcli}\ \mathit{nodi}\ \mathit{lev}\\ (\mathit{dclo}_2, \mathrm{Items}^{1\to 2}) & = & \texttt{trav}_1\textbf{Items}\ \mathit{Items}\ \mathit{dclo}_1\ \mathit{nodo}\ \mathit{lev}\end{array}\\[2ex]
\texttt{trav}_1\textbf{Items}\ (\textbf{NilItems})\ \mathit{dcli}\ \mathit{nodi}\ \mathit{lev} = (\mathit{dcli}, \textbf{NilItems}^{1\to 2}\ \mathit{nodi})\\[2ex]
\texttt{trav}_2\textbf{Items}\ (\textbf{Cons\_Items}\ \mathit{Item}\ \mathit{Items})\ \mathit{env}\ (\textbf{Cons\_Items}^{1\to 2}\ \mathrm{Item}^{1\to 2}\ \mathrm{Items}^{1\to 2}) = (\mathit{code}_1\texttt{++}\mathit{code}_2, \mathit{nodo}_2)\\
\quad\texttt{where}\quad \begin{array}{lcl}\mathit{code}_1 & = & \texttt{trav}_2\textbf{Item}\ \mathit{Item}\ \mathit{env}\ \mathrm{Item}^{1\to 2}\\ (\mathit{code}_2, \mathit{nodo}_2) & = & \texttt{trav}_2\textbf{Items}\ \mathit{Items}\ \mathit{env}\ \mathrm{Items}^{1\to 2}\end{array}\\[2ex]
\texttt{trav}_2\textbf{Items}\ (\textbf{NilItems})\ \mathit{env}\ (\textbf{NilItems}^{1\to 2}\ \mathit{nodo}) = ([], \mathit{nodo})\\[2ex]
\texttt{trav}_1\textbf{Item}\ (\textbf{Decl}\ \mathit{name})\ \mathit{dcli}\ \mathit{nodi}\ \mathit{lev} = (\textbf{Consenv}\ \mathit{name}\ \mathit{lev}\ \mathit{nodi}\ \mathit{dcli}, \mathit{nodi}+1, \textbf{Decl}^{1\to 2}\ )\\[2ex]
\texttt{trav}_1\textbf{Item}\ (\textbf{Use}\ \mathit{Name})\ \mathit{dcli}\ \mathit{nodi}\ \mathit{lev} = (\mathit{dcli}, \mathit{nodi}, \textbf{Use}^{1\to 2}\ )\\[2ex]
\texttt{trav}_1\textbf{Item}\ (\textbf{Block}\ \mathit{Items})\mathit{dcli}\ \mathit{nodi}\ \mathit{lev} = (\mathit{dcli}, \mathit{nodi}, , \textbf{Block}^{1\to 2}\ (\mathit{lev}+1))\\[2ex]
\texttt{trav}_2\textbf{Item}\ (\textbf{Decl}\ \mathit{name})\ \mathit{env}\ \textbf{Decl}^{1\to 2}\ = ([])\\[2ex]
\texttt{trav}_2\textbf{Item}\ (\textbf{Use}\ \mathit{name})\ \mathit{env}\ \textbf{Use}^{1\to 2}\ = (\mathit{code})\\
\quad\texttt{where}\quad \mathit{code}\ =\ [\texttt{access}\ \mathit{env}\ \mathit{name}]\\[2ex]
\texttt{trav}_2\textbf{Item}\ (\textbf{Block}\ \mathit{Items})\ \mathit{env}\ (\textbf{Block}^{1\to 2}\ \mathit{lev}) = ([\textbf{Enter}\ \mathit{nodo}]\ \texttt{++}\ \mathit{code}_1\ \texttt{++}\ [\textbf{Leave}\ \mathit{nodo}])\\
\quad\texttt{where}\quad \begin{array}{lcl}(\mathit{dclo}, \mathrm{Items}^{1\to 2}) & = & \texttt{trav}_1\textbf{Items}\ \mathit{Items}\ \mathit{env}\ 0\ \mathit{lev}\\ (\mathit{code}_1, \mathit{nodo}) & = & \texttt{trav}_2\textbf{Items}\ \mathit{Items}\ \mathit{dclo}\ \mathrm{Items}^{1\to 2}\end{array}
\end{array}
$$

Figure 3: Strict Compiler.

4 Further Optimizations

When looking at the functions of figure 3 we see that some of them do very little work. In this section we show how, after unfolding some definitions, some functions may be removed from the program.

4.1 Unfolding Redundant Data Types

Observe that neither the lazy nor the strict version of our compiler do compute any value when traversing nodes of type *Items*. They only pass arguments and results.We will now statically transform the program, preserving its semantics, in order to avoid these unnecessary steps in the computation. More efficient programs are obtained if we unfold the definition of data type *Item*:

$$
\begin{array}{llll}
\texttt{data}\ \mathit{Program} & = & \mathbf{P} & \mathit{Items} \\
\texttt{data}\ \mathit{Items} & = & \mathbf{Cons\_Item\_Block} & \mathit{Items}\ \mathit{Items} \\
 & \mid & \mathbf{Cons\_Item\_Use} & \mathit{Name}\ \mathit{Items} \\
 & \mid & \mathbf{Cons\_Item\_Decl} & \mathit{Name}\ \mathit{Items} \\
 & \mid & \mathbf{NilItems} & \\
\texttt{data}\ \mathit{Name} & = & \mathbf{Ident} & \mathit{STR}
\end{array}
$$

and then write the corresponding functions. The bindings for the strict program are also simplified:

$$
\begin{array}{lllll}
\texttt{data}\ \mathrm{Items}^{1\to 2} & = & \mathbf{NilItems}^{1\to 2} & \mathit{Int} & \texttt{-- type(nodo)} \\
 & \mid & \mathbf{Cons\_Item\_Block}^{1\to 2} & \mathit{Int}\ \mathrm{Items}^{1\to 2} & \texttt{-- type(lev)}
\end{array}
$$

Consider the functions $\texttt{trav}_1\mathbf{Item}$ and $\texttt{trav}_2\mathbf{Item}$ applied to **Use** and **Decl** respectively. The compiler does not perform any computation when traversing **Use** nodes, since only in the second traversal the `uses` are processed. The same holds for **Decl** nodes in the second traversal. In this traversal the program is only computing the code and no declarations are collected in the environment anymore. So, instances of such nodes can be removed from the tree for the first and second traversal respectively. This can be achieved if we use different representations of the tree for different traversals, i.e., if we *split* the tree.

4.2 Splitting

A *split tree* T is a tuple $[T_1, \ldots T_n]$ of terms, where n is the number of traversals performed on T. Term T_v includes only that part of T that is actually inspected during traversal $\texttt{trav}_v\mathbf{T}$, with $1 \leq v \leq n$.

In our example nodes of type *Items* are traversed twice, inducing the *split data type* $Items_s = (Items_1, Items_2)$. The complete splitted data types are:

$$
\begin{array}{lll}
\texttt{data}\ \mathit{Program}_1 & = & \mathbf{P}_1\ \mathit{Items}_1\ \mathit{Items}_2 \\
\texttt{data}\ \mathit{Items}_1 & = & \mathbf{NilItems}_1 \\
 & \mid & \mathbf{Cons\_Item\_Decl}_1\ \mathit{Name}\ \mathit{Items}_1 \\
 & \mid & \mathbf{Cons\_Item\_Block}_1\ \mathit{Items}_1 \\
 & \mid & \mathbf{Cons\_Item\_Use}_1\ \mathit{Items}_1
\end{array}
\qquad
\begin{array}{lll}
\texttt{data}\ \mathit{Items}_2 & = & \mathbf{NilItems}_2 \\
 & \mid & \mathbf{Cons\_Item\_Decl}_2\ \mathit{Items}_2 \\
 & \mid & \mathbf{Cons\_Item\_Block}_2\ \mathit{Items}_2\ \mathit{Items}_1\ \mathit{Items}_2 \\
 & \mid & \mathbf{Cons\_Item\_Use}_2\ \mathit{Name}\ \mathit{Items}_2
\end{array}
$$

Note that the data type $Name$ is only included in the type constructor $\mathbf{Cons\_Item\_Decl}_1$ for the first traversal and in the $\mathbf{Cons\_Item\_Use}_2$ for the second one. In the other traversals it is not needed. The constructor $\mathbf{Cons\_Item\_Block}_2$ has three children: the first one defines the tree for second traversal (which contains the block) and the other two define the two traversals to the body of the block.

Next we present the functions that split the tree according to the previous data types.

$$
\begin{array}{l}
\texttt{split}\mathbf{Program}\ (\mathbf{P}\ \mathit{Items}) = (\mathbf{P}_1\ \mathit{Items}_1\ \mathit{Items}_2) \\
\quad \texttt{where}\ \ (\mathit{Items}_1, \mathit{Items}_2) \ = \ \texttt{split}\mathbf{Items}\ \mathit{Items} \\
\texttt{split}\mathbf{Items}\ (\mathbf{Cons\_Item\_Decl}\ \mathit{Name}\ \mathit{Items}^2) = (\mathbf{Cons\_Item\_Decl}_1\ \mathit{Name}\ \mathit{Items}^2_1, \mathbf{Cons\_Item\_Decl}_2\ \mathit{Items}^2_2) \\
\quad \texttt{where}\ \ (\mathit{Items}^2_1, \mathit{Items}^2_2) \ = \ \texttt{split}\mathbf{Items}\ \mathit{Items}^2 \\
\texttt{split}\mathbf{Items}\ (\mathbf{Cons\_Item\_Use}\ \mathit{Name}\ \mathit{Items}^2) = (\mathbf{Cons\_Item\_Use}_1\ \mathit{Items}^2_1, \mathbf{Cons\_Item\_Use}_2\ \mathit{Name}\ \mathit{Items}^2_2) \\
\quad \texttt{where}\ \ (\mathit{Items}^2_1, \mathit{Items}^2_2) \ = \ \texttt{split}\mathbf{Items}\ \mathit{Items}^2
\end{array}
$$

The traversal functions must be changed in order to use the split data types. A pattern $\mathbf{p}(\cdots)$ selecting an alternative function is changed to the pattern $\mathbf{p}_v(\cdots)$. A recursive call to a complete tree $\texttt{trav}_v\mathbf{T}\ T$ is mapped into a recursive call to a split tree $\texttt{trav}_v\mathbf{T}\ T_v$. For example, the split version of the function applied on $\mathbf{P}_1$ is:

$$
\begin{array}{lll}
\texttt{trav}_1\textbf{Program}\ (\mathbf{P}_1\ \mathit{Items}_1\ \mathit{Items}_2 = (\mathit{code}) & & \\
\ \texttt{where}\ \ (\mathit{dclo}, \mathrm{Items}^{1\to 2}) & = & \texttt{trav}_1\textbf{Items}\ \mathit{Items}_1\ \textbf{Emptyenv}\ 0\ 0 \\
\qquad\quad (\mathit{code}, \mathit{nodo}) & = & \texttt{trav}_2\textbf{Items}\ \mathit{Items}_2\ \mathit{dclo}\ \mathrm{Items}^{1\to 2}
\end{array}
$$

$$
\begin{array}{lll}
\texttt{trav}_1\textbf{Items}\ (\textbf{Cons\_Item\_Use}_1\ \mathit{Items}_1)\ \mathit{dcli}\ \mathit{nodi}\ \mathit{lev} = (\mathit{dclo}, \mathrm{Items}^{1\to 2}) & & \\
\ \texttt{where}\ \ (\mathit{dclo}, \mathrm{Items}^{1\to 2}) & = & \texttt{trav}_1\textbf{Items}\ \mathit{Items}_1\ \mathit{dcli}\ \mathit{nodi}\ \mathit{lev}
\end{array}
$$

4.3 Elimination

As a result of splitting an important optimization can be performed: the elimination of some data types and some *redundant* functions consisting of copy rules only. Consider the function $\texttt{trav}_1\textbf{Items}$ presented above, this function does not perform any usefull computation. It directly passes arguments and results to a recursive call to itself. The split version of the strict program contains two redundant functions: the alternatives applied to the split data type constructors $\textbf{Cons\_Item\_Decl}_1$ and $\textbf{Cons\_Item\_Use}_2$. Thus, these alternatives can be eliminated. Nodes that are instances of those type constructors can be eliminated too. Elimination requires three steps:

1. First the redundant type constructors are eliminated;

2. Second the split functions are transformed in order to deal with the fact that nodes that are instances of such constructors have disappeared.

   In the running example the transformed split functions are:

$$
\begin{array}{lll}
\texttt{splitItems}\ (\textbf{Cons\_Item\_Decl}\ \mathit{Name}\ \mathit{Items}^2) = (\textbf{Cons\_Item\_Decl}_1\ \mathit{Name}\ \mathit{Items}_1^2, \mathit{Items}_2^2) & & \\
\ \texttt{where}\ \ (\mathit{Items}_1^2, \mathit{Items}_2^2) & = & \texttt{splitItems}\ \mathit{Items}^2 \\
\texttt{splitItems}\ (\textbf{Cons\_Item\_Use}\ \mathit{Name}\ \mathit{Items}^2) = (\mathit{Items}_1^2, \textbf{Cons\_Item\_Use}_2\ \mathit{Name}\ \mathit{Items}_2^2) & & \\
\ \texttt{where}\ \ (\mathit{Items}_1^2, \mathit{Items}_2^2) & = & \texttt{splitItems}\ \mathit{Items}^2
\end{array}
$$

3. Finally, the redundant functions are eliminated.

Observe that, without the unfolding of the *Item* data type the copy operations were hidden, that is, the program would need to pattern-match nodes of type *Item* in order to know which particular instance it was.

Although we have presented the splitting and elimination as transformations of the program and the associated tree structures, our system actually generates code which directly constructs the splitted and contracted trees.

5 Attribute Grammars

We have developed a system which performs the global analysis described in section 3 and the splitting and elimination optimizations. The programs are specified by an attribute grammar and the strict functional programs are automatically generated. The bindings, the split data types and the split functions are induced by the attribute grammar too.

We have implemented a generator for the construction of incremental evaluation of attribute grammars. This generator uses the techniques from this paper. Experience shows that large functional programs can be efficiently implemented with our techniques. The generator is bootstrapped: it generates itself from a rather large specification. The original program has 90 traversal functions that work on data types which together have 307 different constructors.

The transformed program consists of 866 sub-traversal functions which require 2188 binding constructors. Some traversal functions are transformed in as many as 12 sub-traversals.

5.1 Attribute Evaluator

We have incorporated a new back-end to the LRC system [Pen94] in order to produce GOFER based evaluators. The code presented in figure 4 has been automatically generated by our system, starting from the attribute-grammar-equivalent of the initial circular program. In order to stick with our GOFER-based presentation we have only replaced some semantic functions by their GOFER equivalents.

$$\begin{array}{lll}
\multicolumn{3}{l}{\mathtt{trav}_1\mathbf{Program}\ (\mathbf{P}_1\ \mathit{Items}_1\ \mathit{Items}_2) = (\mathit{Program.code})} \\
\mathtt{where}\ \ (\mathit{Items.dclo}, \mathrm{Items}^{1\to 2}) & = & \mathtt{trav}_1\mathbf{Items}\ \mathit{Items}_1\ \mathbf{Emptyenv}\ 0\ 0 \\
\phantom{\mathtt{where}}\ \ (\mathit{Program.code}, \mathit{Items.nodo}) & = & \mathtt{trav}_2\mathbf{Items}\ \mathit{Items}_2\ \mathit{Items.dclo}\ \mathrm{Items}^{1\to 2}
\end{array}$$

$$\mathtt{trav}_1\mathbf{Items}\ (\mathbf{NilItems}_1\ )\ \mathit{Items.dcli}\ \mathit{Items.nodi}\ \mathit{Items.lev} = (\mathit{Items.dcli}, \mathbf{NilItems}^{1\to 2}\ \mathit{Items.nodi})$$

$$\begin{array}{lll}
\multicolumn{3}{l}{\mathtt{trav}_1\mathbf{Items}\ (\mathbf{Cons_Item_Decl}_1\ \mathit{Name}\ \mathit{Items}^2_1)\ \mathit{Items.dcli}\ \mathit{Items.nodi}\ \mathit{Items.lev} = (\mathit{Items.dclo}, \mathrm{Items}^{1\to 2})} \\
\mathtt{where}\ \ (\mathit{Items}^2.\mathit{dclo}, \mathrm{Items}^{1\to 2}) & = & \mathtt{trav}_1\mathbf{Items}\ \mathit{Items}^2_1\ \mathit{Items.dcli}\ (\mathit{Items.nodi} + 1)\ \mathit{Items.lev} \\
\phantom{\mathtt{where}}\ \ \mathit{Items.dclo} & = & \mathbf{Consenv}\ \mathit{Name}\ \mathit{Items.lev}\ \mathit{Items.nodi}\ \mathit{Items}^2.\mathit{dclo}
\end{array}$$

$$\begin{array}{lll}
\multicolumn{3}{l}{\mathtt{trav}_1\mathbf{Items}\ (\mathbf{Cons_Item_Block}_1\ \mathit{Items}^3_1)\ \mathit{Items.dcli}\ \mathit{Items.nodi}\ \mathit{Items.lev} =} \\
\multicolumn{3}{l}{(\mathit{Items.dclo}, \mathbf{Cons_Item_Block}^{1\to 2}\ (\mathit{Items.lev} + 1)\ \mathrm{Items}^{1\to 2})} \\
\mathtt{where}\ \ (\mathit{Items.dclo}, \mathrm{Items}^{1\to 2}) & = & \mathtt{trav}_1\mathbf{Items}\ \mathit{Items}^3_1\ \mathit{Items.dcli}\ \mathit{Items.nodi}\ \mathit{Items.lev}
\end{array}$$

$$\mathtt{trav}_2\mathbf{Items}\ (\mathbf{NilItems}_2\ )\ \mathit{Items.env}\ (\mathbf{NilItems}^{1\to 2}\ \mathit{Items.nodo}) = ([], \mathit{Items.nodo})$$

$$\begin{array}{lll}
\multicolumn{3}{l}{\mathtt{trav}_2\mathbf{Items}\ (\mathbf{Cons_Item_Use}_2\ \mathit{Name}\ \mathit{Items}^2_2)\ \mathit{Items.env}\ \mathrm{Items}^{1\to 2} = (\mathit{Items.code}, \mathit{Items.nod})} \\
\quad (\mathit{Items}^2.\mathit{code}, \mathit{Items.nodo}) & = & \mathtt{trav}_2\mathbf{Items}\ \mathit{Items}^2_2\ \mathit{Items.env}\ \mathrm{Items}^{1\to 2} \\
\quad \mathit{Items.code} & = & [\mathtt{access}\ \mathit{Items.env}\ \mathit{Name}]\ \mathtt{++}\ \mathit{Items}^2.\mathit{code}
\end{array}$$

$$\begin{array}{lll}
\multicolumn{3}{l}{\mathtt{trav}_2\mathbf{Items}\ (\mathbf{Cons_Item_Block}_2\ \mathit{Items}^3_2\ \mathit{Items}^2_1\ \mathit{Items}^2_2)\ \mathit{Items.env}\ (\mathbf{Cons_Item_Block}^{1\to 2}\ \mathit{Items}^2.\mathit{lev}\ \mathrm{Items}^{1\to 2}) =} \\
\multicolumn{3}{l}{(\mathit{Items.code}, \mathit{Items.nodo})} \\
\mathtt{where}\ \ (\mathit{Items}^3.\mathit{code}, \mathit{Items.nodo}) & = & \mathtt{trav}_2\mathbf{Items}\ \mathit{Items}^3_2\ \mathit{Items.env}\ \mathrm{Items}^{1\to 2} \\
\phantom{\mathtt{where}}\ \ (\mathit{Items}^2.\mathit{dclo}, \mathrm{Items}^{1\to 2}) & = & \mathtt{trav}_1\mathbf{Items}\ \mathit{Items}^2_1\ \mathit{Items.env}\ 0\ \mathit{Items}^2.\mathit{lev} \\
\phantom{\mathtt{where}}\ \ (\mathit{Items}^2.\mathit{code}, \mathit{Items}^2.\mathit{nodo}) & = & \mathtt{trav}_2\mathbf{Items}\ \mathit{Items}^2_2\ \mathit{Items}^2.\mathit{dclo}\ \mathrm{Items}^{1\to 2} \\
\phantom{\mathtt{where}}\ \ \mathit{Items.code} & = & [\mathbf{Enter}\ \mathit{Items}^2.\mathit{nodo}]\ \mathtt{++}\ \mathit{Items}^2.\mathit{code}\ \mathtt{++}\ [\mathbf{Leave}\ \mathit{Items}^2.\mathit{nodo}]\ \mathtt{++}\ \mathit{Items}^3.\mathit{code}
\end{array}$$

Figure 4: Splitted Compiler.

5.2 Incremental Behaviour

In next table we present results of two different incremental reevaluations of the example sentence (see section 2): one modification of the last `use` statement in the global block into `use` w and one modification of the declaration `decl` w into `decl` u. Both reevaluations started in a state where all functions applied when processing the initial sentence were stored in the cache.

$$
\begin{array}{lll}
Items & = & \textbf{Cons\_Item\_Decl}\ Name\ Items^2 \\
 & & \quad Items.used \; = \; \textbf{Consuse}(Name, Items^2.used) \\
 & | & \textbf{Cons\_Item\_Use}\ Name\ Items^2 \\
 & & \quad Items.used \; = \; Items^2.used \\
 & | & \textbf{Cons\_Item\_Block}\ Items^2\ Items^3 \\
 & & \quad Items^2.dcli \; = \; \texttt{project}(Items.env, Items^2.used) \\
 & & \quad Items.used \; = \; Items^3.used \\
 & | & \textbf{NilItems} \\
 & & \quad Items.used \; = \; \textbf{Emptyuse}()
\end{array}
$$

Figure 5: Attribute Grammar notation.

	Strict Compiler of fig. 3		Splitted Compiler of fig. 4	
Modification	*Hits*	*Misses*	*Hits*	*Misses*
Change global `use`:	6	15	6	5
Change global `decl`:	3	25	3	15

When processing the first modification the *splitted* compiler has a better behaviour since it does not need to reevaluate the declarations. Modifying a global variable usually has a poor incremental behaviour, since all the environment change. Both evaluators only reuse 3 functions and have to recompute most of the functions. The difference in the number of misses are due to the absence of calls to eliminated functions.

In our example however the nested block does not use that changed global variable at all. Nevertheless, since its total environment has changed, all the functions applied to process that block are recomputed. We can easily write an AG dealing with this problem by letting each block synthesize a list of those variables which actually occur in a **Use** constructor and project the environment on that list. This is easily achieved by adding the following attribution rules to the AG as presented in figure 5.

Where the type of the attribute *used* is a list of $Name$s and `project` is a simple semantic function which implements the projection.

In the table below we present the results of the *splitted* evaluator using such a projection. The resulting splitted evaluator is a 3 visit-evaluator.

Modification	*Hits*	*Misses*
Change global `use`:	8	5
Change global `decl`:	7	10

When reevaluating the second modification the number of misses decreases and the number of hits increases. The nested block is reevaluated without recomputing any visit-function. It reuses all three functions: the one that synthesizes the list of used variables, the one that synthesizes the declarations and the one that synthesizes the code.

Note that this efficient program transformation was performed without changing any recursive function. The attribution rules presented above were added to the AG using its natural structural decomposition.

6 Conclusions

This paper presented a transformation from lazy into non-lazy ones. A function is transformed into one or more functions that must be called in sequence. Extra data types and arguments are added to the non-lazy functions when values computed in one function are needed in another. Measurements show that the transformed and optimized functions can be executed incrementally with a function cache. The techniques can be used to combine several catamorphisms on mutual recursive data types.

References

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

[Hug85] John Hughes. Lazy memo-functions. In Jean-Pierre Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, volume 201 of *LNCS*, pages 129–146. Springer-Verlag, September 1985.

[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.

[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`.

[MFP91] Erik Meyer, Maarten Fokkinga, and Ross Patterson. Functional programming with bananas, lenses and barbed wire. In *Functional Programming Languages and Computer Architecture*, 1991.

[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.

[SdM93] S. Doaitse Swierstra and O. de Moor. Virtual data structures. In Bernhard Möller, Helmut Partsch, and Steve Schuman, editors, *Formal Program Development*, volume 755 of *LNCS*, pages 355–371, 1993.

[SV91] Doaitse Swierstra and Harald Vogt. Higher order attribute grammars. In H. Alblas and B. Melichar, editors, *International Summer School on Attribute Grammars, Applications and Systems*, volume 545 of *LNCS*, pages 48–113. Springer-Verlag, 1991.

[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.

[Wad90] Philip Wadler. Deforestation: transforming programs to eliminate trees. *Theoretical Computer Science*, 73:231–248, 1990.