Tools and Libraries to Model and Manipulate Circular Programs

João Paulo Fernandes * † João Saraiva †

Department of Computer Science, University of Minho, Portugal
{jpaulo,jas}@di.uminho.pt

Abstract

This paper presents techniques to model circular lazy programs in a strict, purely functional setting. Circular lazy programs model any algorithm based on multiple traversals over a recursive data structure as a single traversal function. Such elegant and concise circular programs are defined in a (strict or lazy) functional language and they are transformed into efficient strict and deforested, multiple traversal programs by using attribute grammars-based techniques. Moreover, we use standard slicing techniques to slice such circular lazy programs.

We have expressed these transformations as an *Haskell* library and two tools have been constructed: the *HaCirc* tool that refactors *Haskell* lazy circular programs into strict ones, and the *OCirc* tool that extends *Ocaml* with circular definitions allowing programmers to write circular programs in *Ocaml* notation, which are transformed into strict *Ocaml* programs before they are executed. The first benchmarks of the different implementations are presented and show that for algorithms relying on a large number of traversals the resulting strict, deforested programs are more efficient than the lazy ones, both in terms of runtime and memory consumption.

Categories and Subject Descriptors D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming; D.2.2 [*Software Engineering*]: Design Tools and Techniques; D.3.3 [*Programming Languages*]: Language Constructs and Features—lazy evaluation, eager/strict evaluation

General Terms Algorithms, Design, Languages

Keywords Multiple Traversal Algorithms, Circular Programming, Intermediate Data Structures, Traversal Scheduling

1. Introduction

Circular lazy programs, as introduced by Richard Bird [2], are a famous example that demonstrates the power of a lazy evaluation mechanism. Bird's work showed that any multiple traversal algorithm can be expressed in a lazy language as a single traversal *circular function*, being the *repmin* program the reference example in this case. Such a (virtual) circular function may contain a *circular definition*, that is, an argument of a function call that is also a result of that same call. Although circular definitions induce non-termination under a strict evaluation mechanism, they can be immediately evaluated using a lazy evaluation strategy. The lazy engine is able to compute the right evaluation order, if that order exists. Indeed, using this style of circular programming, the programmer does not have to concern him/herself with the definition and the scheduling of the different traversal functions, since a single (traversal) function has to be defined. Moreover, because there is a single traversal function, the programmer does not have to define intermediate gluing data structures to convey values computed in one traversal and needed in following ones, either.

On the contrary, defining multiple traversal programs within a strict, purely functional setting can be a complex task: additional data structures have to be defined and constructed/destructed to explicitly pass values computed in one traversal and needed in following ones. Furthermore, there are algorithms that rely on a large number of traversals whose scheduling is not a trivial one. As a result, expressing such algorithms in a strict setting leads to longer solutions which are harder to write, understand and maintain.

The purpose of this paper is three-fold. Firstly, we present techniques to model and transform circular lazy programs into strict multiple traversal (equivalent) ones. This refactoring of circular programs is expressed in terms of attribute grammar techniques [14]. We have informally presented this approach in [24]. In this paper, we give its formal definition and its implementation as an *Haskell* library: the CircLib library. Two tools have been constructed to transform *Haskell* and *Ocaml* based circular programs into their strict counterparts. In this way, we make this concise and elegant style of expressing multiple traversal algorithms also available to non-lazy functional programmers. Moreover, we use partial evaluation techniques to derive deforested versions of the strict programs. Secondly, because our techniques break up circular definitions into several strict functions, we can directly apply standard slicing techniques to slice circular lazy programs. That is, given a circular program we derive a program that performs the computations needed to produce some of its results (backward slicing), or the computations that use some of its arguments (forward slicing). Thirdly, we conduct the first systematic benchmarking of circular, strict and deforested programs. The results show that for algorithms relying on large number of traversals the strict, deforested programs are more efficient than the lazy ones, both in terms of runtime and memory consumption.

This paper is organized as follows: Section 2 presents circular programs, the notation used, and it introduces the running example used throughout the paper. Section 3 presents the derivation of strict programs from circular ones. Section 4 presents the slicing of circular programs. In Section 5 we discuss the class of circular

* Supported by Fundação para a Ciência e Tecnologia (FCT), grant No. SFRH/BD/19186/2004

† Partially funded by Fundação para a Ciência e Tecnologia (FCT), grant No. POSI/CHS/44304/2002

programs considered. Section 6 presents the tools developed and Section 7 shows benchmark results. Section 8 shows our conclusions. Finally, in appendices, we include the API library.

2. Circular Programs

Circular programs were first proposed by Bird [2] as an elegant and efficient technique to eliminate multiple traversals of data structures. As the names suggests, circular programs are characterized by having what appears to be a circular definition: arguments in a function call depend on results of that same call. That is, they contain definitions of the form: `(...,x,...) = f ... x ...`

In order to motivate the use of circular programs, Bird introduces the following programming problem, widely known as the *repmin* problem: consider the problem of transforming a binary leaf tree into a second tree, identical in shape to the original one, but with all the tip values replaced by the minimum tip value.

In a strict and purely functional setting, solving this problem would require a two traversal strategy: the first traversal would compute the original tree's minimum value, and the second traversal would replace all the tip values by the minimum value, therefore producing the desired tree result. However, a two traversal strategy is not essential to solve the *repmin* problem. An alternative solution can, on a single traversal, compute the minimum tip value and, at the same time, replace all tip values by that minimum value. Bird showed how the single traversal, presented next, may be obtained by transforming the original program using the following techniques: tupling, fold-unfold and circular programming.

```
repmin (Tip n      , m) = (Tip m        , n)
repmin (Fork (l, r), m) = (Fork (t1, t2), min m1 m2)
     where (t1, m1) = repmin (l, m)
           (t2, m2) = repmin (r, m)
transform t = nt
     where (nt, m)  = repmin (t, m)
```

Bird's work showed the power of circular programming, not only as an optimization technique to eliminate multiple traversal of data, but also as a powerful, elegant and concise technique to express multiple traversal algorithms. Circular programs are also used in the construction of Haskell compilers [17, 9], to express pretty printing algorithms [25], breadth-first traversal strategies [18], type systems [1] and aspect-oriented compilers [5]. As an optimization technique, circular programs are used, for example, in the deforestation of accumulating parameters [29]. Circular programs can also be obtained through partial evaluation [16] and continuations [3]. As Johnsson [11] and Swierstra and Kuiper [15] originally showed, circular programs are the natural representation of attribute grammars in a lazy setting [27, 4, 22, 6].

2.1 Notation

To demonstrate our techniques, we use the language given in Fig. 1. A program is a sequence of definitions. The language natively incorporates integers (`0,1,...`), with the usual operators, characters (`'a','b',...,'z'`) and strings (character sequences). It also makes use of lists, the empty list being represented by `[]`, the insertion of an element `x` in the head of a list `l` being represented by `x:l` and the concatenation of two lists, `l1` and `l2`, being represented by `l1 ++ l2`. The semantics of the language is that of standard lazy functional languages.

2.2 The Table Formatter Program

The *repmin* problem is a famous example which nicely exploits and demonstrates the power of circular programming. However, when defining more realistic multiple traversal problems, like for example the four traversal pretty printing algorithm presented in [25], the programmer has to define additional gluing data structures to pass

Expressions			
e	::=	v	variables
	|	n	constants
	|	$(e_1, ..., e_n)$	tuples
	|	$C(v_1, ..., v_n)$	constructors
Attributions			
a	::=	$v_1 = v_2$	variable copying
	|	$v_1 = C(v_2, ..., v_n)$	contructor value
	|	$v_1 = f\ e$	function application
	|	$v\ = n$	constant value
	|	$(v_1, ..., v_n) = v_m\ e$	recursive calls
Function and Data-Types definitions			
$Decl$	::=	$v\ e_1 = e_2$	function definition,
	|	$v\ e_1 = e_2\ where\ a_1\ ...\ a_n$	with a where clause
	|	$T\ =\ C_1\ t_1 \mid ... \mid C_n\ t_n$	type definition
t	::=	$() \mid (t_1, ..., t_n) \mid Int \mid Char \mid String \mid T$	

Figure 1. Abstract syntax

values to future traversals. Furthermore, the scheduling of traversals can be a complex task, as well.

To show more clearly the properties of circular programming we will use a more realistic example. Let us consider that we want to define a program that formats HTML style tables. Fig. 2 shows an example of a possible input (left) and correspondent output (right).

The straightforward solution to construct such a program is to compute the *heights* and *widths* of each element in the table, before we define the formatting. They can be computed as follows: the height of an element is the height of a data element (*i.e.*, a string with height `1`) or the height of a nested table. The height of a row is the maximum height of its elements. And, the height of a table is the sum of the heights of its rows plus the line separators. The width of an element is the length of the data element,or the width of the nested table. Like for the height of a column, the width of a column is the maximum width of the elements in that column, and the width of a table is the sum of the widths of its columns (plus the column separators). In the input HTML example, we have annotated tag TD with the height of the element (superscript) and its width (subscript).

Having defined the heights and widths of the elements in a table, the next step is to do the formatting. Obviously, we will need to add some vertical and horizontal glue (spaces) so that we can obtain the desired output. In our example, in the first column of the second row we need to add `2` spaces of horizontal glue (the element has width `14` whilst the nested table has `12`: see associated subscripts). Such two spaces have to be used `7` times as vertical glue since that column has that height.

The immediate implementation of this algorithm would rely on a two traversal strategy. First we traverse the HTML tree to compute the correct heights and widths of each element, and in a second traversal we produce the formatting using those values. Note, however, that in order to compute the width of our outermost table, we need to compute the width of each column first. Thus, we need to know the width of the nested table. According to this approach that table has to be traversed twice as well. As a result, in the first traversal of an outermost table we need to perform the two traversals to its nested tables. So, the computations related to the first and second traversals are intermingled. Moreover, the values of the height and width of the nested table have to be passed to the second traversal of the outermost table: they are needed to defined the necessary vertical and horizontal glue. That is to say that in a straightforward implementation of this program an intermediate data structure has to be defined and constructed to pass explicitly the height and width of a nested table from the first to the second traversal.

Next, we present the elegant and concise *Table* circular program that relies on a single traversal. Note that to construct such a program the programmer did not have to define and construct/destruct

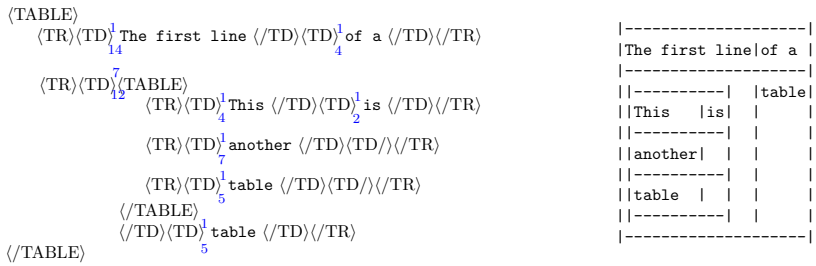

Figure 2. HTML Table Formatting

gluing data structures nor to schedule the different traversals. Such data structures and the scheduling of computations will be defined by the static analysis and transformations we present in Section 3.

HTML like tables are defined by the following recursive data type definitions:

```
Table = RootTable Rows      Elems = EmptyElems
Rows  = EmptyRows                 | ConsElems(Elem,Elems)
      | ConsRows(Row,Rows)  Elem  = OneStr String
Row   = OneRow Elems              | OneTable Table
```

Next, we present the single traversal circular program. As referred before, for each table the program computes the desirable format (*lines*), its height (*mh*) and width (*mw*). The function that processes the rows returns three things: the format of the rows, the height of those rows and the list of widths of the columns (in our example, this list will be $[14, 5]$). Thus, the width of the table is the sum of those widths plus the separators (22 in our example). Each row needs to know the available width of each column, to add glue in the format, if necessary. Thus, this function receives as argument the list of available widths of the columns. This list is the computed list of widths. As we can see below, a circular dependency is defined.

```
evalTable :: Table -> ([String],Int,Int)
evalTable (RootTable rows) = (lines,mh,mw)
  where (lines1,mh,mws) = evalRows (rows,mws)
        mw     = (sum mws)+(length mws)+1
        lines  = sepLine (mws, lines1)
```

When processing the rows, we accumulate the heights of each row (*mh*), and we *zip* the widths of the columns with the maximum values of the rows. In our example, the two rows produce the following two lists of widths $[14, 4]$ (first) and $[12, 5]$. The result of *zipwith_max* is $[14, 5]$, that is, the maximum width of each column.

```
evalRows (ConsRows(row,rows),aws) = (lines,mh,mws)
  where (lines1,mh1,mws1) = evalRow  (row ,aws)
        (lines2,mh2,mws2) = evalRows (rows,aws)
        mh     = mh1 + mh2 + 1    -- (+1 is for the separator)
        mws    = zipwith_max (mws1, mws2)
        lines = addSep (aws,lines1,lines2)
evalRows (EmptyRows,aws) = ([],0,[])
```

For each individual row, we receive as argument the available widths of its columns, and we have to compute its format, height and the widths (that will be used to compute the widths of the table elements). One result of the function *evalElems* is the maximum height (*mh*) of the elements in the row. We need to pass it to those same elements, in order to add vertical glue. Once again we use a circular definition: the height computed is the height passed as argument.

```
evalRow (OneRow els,aws) = (lines, mh, mws)
  where (lines1,mh,mws)  = evalElems (els,mh,aws)
        lines  = addBorder lines1
```

The elements of one row receive as argument the available height of the row and the list of maximum widths. It returns the format, the height of the row and the widths.

```
evalElems(ConsElems(el,els),ah,aws)=(lines,mh,mws)
  where aws2  = tail aws
        (lines1,mh1,mw1)  = evalElem  el
        (lines2,mh2,mws2) = evalElems(els,ah,aws2)
        mws   = mw1:mws2
        mh    = max (mh1, mh2)
        lines = glue(aws,mw1,ah,mh1,lines1,lines2)
evalElems (EmptyElems,ah,aws) = ([],0,[])
```

Finally, the function that processes individual elements, returns their format, height and width.

```
evalElem (OneStr str) = ([str],1,length str)
evalElem (OneTable table) = (lines1,mh,mw1)
  where (lines1,mh1,mw1)  = evalTable table
        mh = mh1 + 1
```

The functions `addSep`, `sepLine`, `addBorder` and `glue`, add line separators, horizontal and vertical borders, and glue table lines, respectively.

This table formatter is a *circular* program: circular definitions occur twice as we can see in the program. These programs can be immediately evaluated under a lazy evaluation mechanism. The lazy engine will be able to schedule the computations and convey values between different traversal functions at execution time. Under a strict evaluation setting, however, such programs induce non-termination. Next, we will show how to transform this circular program into a strict and deforested multiple traversal program.

3. From Circular to Strict Programs

In this section we will describe a program transformation technique to derive a strict program from its lazy circular definition. A strict evaluation setting is attractive not only because we obtain implementations that are not restricted to a lazy semantics execution model, but also because we obtain very efficient implementations in terms of memory and time consumption. The resulting program can be correctly executed under both a strict and a lazy execution model.

3.1 Detection of Circular Definitions

Let us analyze in detail one of the most intricate function alternatives of the above program: the function `evalTable` applied at the node `RootTable`, where a circular definition occurs. Figure 3 shows the induced dependency relation (represented as a graph), which follows from a flow analysis of the total program.

For each alternative function definition a dependency graph is induced. Such graphs are labeled with the data type constructor that the alternative definition refers to. Furthermore, in these graphs we use undirected (solid) lines to connect the types involved in a tree-like structure: result type on top and arguments at the bottom. The variable names representing formal arguments(results) of the function definition are displayed at the left(right) of the resulting type. Such variable names are displayed in all occurrences of that data type in the different induced graphs. Notice, for example, that

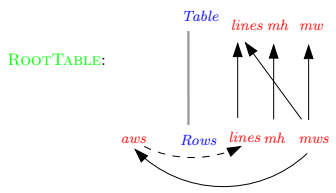

Figure 3. Dependency graph of function `evalTable`

the results produced by `evalTable`: `lines`, `mh` and `mw`, are drawn to the right of `Table`'s position. Arrows are used in the graphs to represent dependencies between variables. For example, the arrow with origin in the variable `mws` and destination in the variable `aws` represents that `mws` is used to compute `aws`. We use black lines to represent direct dependencies and dashed-black lines to represent indirect dependencies. Later, we will present the formal process to calculate these dependencies.

As we can easily see in Figure 3, there is an evaluation order to evaluate the so-called circular definition, since no value depends directly nor indirectly on itself.

Dependencies from a result to an argument, however, induce additional traversals to the tree.

The detection of such circular definitions in the abstract syntax tree of the programs under consideration is a straightforward function. Thus, we omit its definition here.

3.2 Partitionable Circular Programs

This section discusses the class of circular programs for which strict programs can be derived. That is, circular programs whose circularity may be eliminated, by statically analyzing the dependencies induced by them. These dependencies are established in the program's functions, between function arguments and function results, and the static analysis consists in determining an alternative evaluation order for them.

The algorithms that compute the alternative evaluation order establish the number of visits and an *interface* for every data-type *X* of the circular program. We denote the interface of data-type *X* by *Interface*(*X*). *Interface*(*X*), as computed by these algorithms, usually has the following shape:

$$\mathit{Interface}(X) = [(args_1, results_1), \ldots, (args_n, results_n)]$$

with $args_i$ = {*arguments of the ith function defined over elements of type X*}
$results_i$ = {*results of the ith function defined over elements of type X*}

Thus, by computing *Interface*(*X*), for every data-type *X*, the scheduling algorithms specify, for every visit to *X*, which arguments are used and which results are computed. Roughly speaking, *Interface*(*X*) fixes the types for every one of the traversal functions for type *X*. Interface *Interface*(*X*) induces a partial order on the arguments and results of the functions defined over X.

The largest class of circular programs for which strict multiple traversal programs can be derived is the class of *partitionable circular programs*. Informally, a circular program is partitioned if for each data-type there is an interface, such that in any function defined over the data-type, its results are computable in an order which is included in the partial order induced by the interface.

For every constructor C of a circular program, let $DP(C)$ be the relation of direct dependencies, between variable occurrences, defined in the function of the circular program that traverses elements built using C (defined `in` C, for short). Formally, let $DP(C)$ be the relation

$$DP(C) = \{Var_1 \rightarrow Var_2 \mid Var_2 \texttt{ depends on } Var_1 \texttt{ in } C\}$$

A program variable (directly) depends on another if the latter is used to compute the former (whether this computation requires complex processing of the latter, or simply be the copy of its value). These dependencies are easily inferred from the program, in the program sentences that match our functional language's three first attribution rules: in the first rule ($v_1 = v_2$), the variable v_1 depends on the variable v_2, in the second rule ($v_1 = C(v_2, ..., v_n)$), v_1 depends on the variables $v_2 ... v_n$ and in the third ($v_1 = f\ e$), v_1 depends on all the variables that occurr in e. We present such dependencies in Figure 4 (black lines were used to represent this type of dependencies). Next, we also present the derived DP relation, for the constructor $RootTable$ of the *Table* program.

$$\begin{aligned} DP(RootTable) = \{&(RootTable, 1, lines) \rightarrow (RootTable, 0, lines),\\ &(RootTable, 1, mh) \rightarrow (RootTable, 0, mh),\\ &(RootTable, 1, mws) \rightarrow (RootTable, 0, mw),\\ &(RootTable, 1, mws) \rightarrow (RootTable, 1, aws),\\ &(RootTable, 1, mws) \rightarrow (RootTable, 0, lines)\} \end{aligned}$$

Each dependency is established between two program variables, each of which is represented by a tuple with three components: the first component represents the constructor, say C, where the dependency is detected and the third component represents the variable name. The second component contains an integer value, say i; this value represents the data-type X_i, in $C : X_1\ X_2\ \ldots\ X_n \rightarrow X_0$, that is an argument of the traversal function that induces the dependency.

For example, we have $RootTable$: `Rows` $\rightarrow$ `Table`, and the variable $(RootTable, 1, lines)$ states the occurrence of a variable, named $lines$, computed by traversing an element of type `Rows`, which is the first argument of the constructor $RootTable$.

Furthermore, the dependency
$(RootTable, 1, lines) \rightarrow (RootTable, 0, lines)$ states that, in the definition of the function that traverses elements built using the constructor $RootTable$ (let such an element be $(RootTable\ x)$), the result value $lines$ is computed by traversing x (i.e., using the $lines$ value computed by traversing a value of type `Rows`). In other words, the result value $lines$, represented by $(RootTable, 0, lines)$, depends on the $lines$ value produced by traversing the first argument of $RootTable$, being this value represented by $(RootTable, 1, lines)$.

Having defined the relation $DP(C)$, we are now ready to give the definition of *partitionable circular program*.

Definition (Partitionable Circular Program). Let $PO(X)$ be the partial order induced by *Interface*(X). A circular program is a *partitionable circular program* if for every constructor $C : X_1\ X_2\ \ldots\ X_n \rightarrow X_0$, the relation
$DP(C) \cup \bigcup_{i=0}^{n} PO(\langle C, i\rangle)$, where $\langle C, i\rangle = X_i$,
is non-circular. In this case we say that the interfaces are *compatible*.•

A non-circular relation of dependencies between variables is a relation that does not include, at the same time, a dependency between a variable a and a variable b, and a dependency between the variable b and the variable a, i.e., by a non-circular relation we mean a cycle-free relation.

The concept of *partitionable circular programs* is inspired in the similar concept for attribute grammars. In [7], Engelfried and Filè proved that deciding whether an attribute grammar is partitionable or not is a NP-complete problem. Kastens [13] defined a subclass of partitionable attribute grammars, the so-called ordered attribute grammars, that can be checked by an algorithm that depends polynomially in time on the size of the attribute grammar. We define a slightly different class of circular programs, that we shall call *L-ordered circular programs*.

Definition (L-Ordered Circular Program). A circular program is a *L-ordered circular program* if there exist total orders $TO(X)$ for every data-type *X* such that for every constructor C that defines values of type *X*, $C : X_1\ X_2\ \ldots\ X_n \rightarrow X_0$, the relation
$DP(C) \cup \bigcup_{i=0}^{n} TO(\langle C, i\rangle)$ is cycle free.•

The total orders $TO(X)$ are easily converted into interfaces: cut them into maximal segments of function arguments and function results.

3.3 Ordered Circular Programs

In this section we present an adaptation of Kastens' attribute scheduling algorithm [13, 21, 20] to circular programs. The basic idea of this algorithm is the following: for each data-type *X* defined in the program, a partial order $DS(X)$ over the program variables that occur in the function defined on *X* is computed. It determines an evaluation order for values in *X*, applicable in any context where *X* may occur. As a result, an element $X.a \rightarrow X.b \in DS(X)$ indicates that a must be computed before b in any node that is an instance of X.

The existence of such an order is a sufficient but not necessary condition for the well-definedness of circular programs. Note that Kastens' ordering algorithm makes a worst case assumption by merging all (indirect) dependencies on variables of a data-type, in any context the data-type may occur, into a single dependency graph. This pessimistic approach, however, is crucial for L-ordered programs: it must always be possible to compute the variables of *X* in the order specified by $DS(X)$, irrespective of the actual context of *X*.

Step 1: $DP = \bigcup_{C \in Constructors} DP(C)$, where $Constructors$ is the set of the program's constructors, is computed; this is the relation of direct dependencies between variable occurrences in the program.

The circular program is not ordered if DP is cyclic.

Step 2: $IDP = \bigcup_{C \in Constructors} IDP(C)$ is computed; this is the relation of induced dependencies between variable occurrences. IDP projects indirect dependencies into dependencies between variable occurrences as follows: every dependency between variables of one occurrence of a symbol, say *X*, induces a dependency between corresponding variables of all occurrences of *X*. Formally it is defined as follows:

$$\begin{aligned} IDP(C) = DP(C) \quad \cup \quad & \{ \ \langle C,i,a\rangle \rightarrow \langle C,i,b\rangle \\ & \mid \langle C',j,a\rangle \rightarrow \langle C',j,b\rangle \in IDP^+ \\ & \wedge \langle C,i\rangle \ = \ \langle C',j\rangle \} \end{aligned}$$

The circular program is not ordered if IDP is cyclic.

Figure 4 shows the IDP relation (black and dashed lines were used to represent it) induced by the *Table* circular program (in fact, for simplicity and readability, Figure 4 omits the representation of the dependencies established, in IDP, between two argument variables and between two result variables, e.g., the dependency $(RootTable, 1, mw) \rightarrow (RootTable, 1, lines)$ is omitted).

The relation $IDS = \bigcup_{X \in Data-Types} IDS(X)$, where $Data-Types$ is the set of the program's data-types, defines the *Induced Dependencies* among variables:

$$\begin{aligned} IDS(X) = \{ \ X.a \rightarrow X.b \quad & \mid \quad (C,i,a) \rightarrow (C,i,b) \in IDP \\ & \wedge \quad \langle C,i\rangle = X \ \} \end{aligned}$$

The IDS relation, for the *Table* program, is presented next.

$IDS(Table) = \{\}$
$IDS(Rows) = \{Rows.aws \rightarrow Rows.lines,\ Rows.mws \rightarrow Rows.aws,$
$\quad Rows.mws \rightarrow Rows.lines\}$
$IDS(Row) = \{Row.aws \rightarrow Row.lines,\ Row.mws \rightarrow Row.aws,$
$\quad Row.mws \rightarrow Row.lines\}$
$IDS(Elems) = \{Elems.ah \rightarrow Elems.lines,\ Elems.aws \rightarrow Elems.lines,$
$\quad Elems.mh \rightarrow Elems.ah,\ Elems.mh \rightarrow Elems.lines,$
$\quad Elems.mws \rightarrow Elems.aws,\ Elems.mws \rightarrow Elems.lines\}$
$IDS(Elem) = \{\}$

Step 3: the "interfaces" for the data-type symbols are determined. That is, the algorithm statically establishes the number of visits to a data-type *X* and for each of those visits it defines which arguments are used to compute which results. Several orders are possible. Kastens' algorithm maximizes the size of the interfaces so that the number of visits is minimized. In order to compute such interfaces we define successively

$$\begin{aligned} A_{X,1} &= Results(X) - \{X.a \mid X.a \rightarrow X.b \in IDS^+\} \\ A_{X,2n} &= \{ \ X.a \quad \mid \quad X.a \in Arguments(X) \\ &\qquad \wedge \quad \forall X.b : X.a \rightarrow X.b \in IDS^+ \\ &\qquad\qquad \Rightarrow \exists m < 2n : X.b \in A_{X,m}\} \\ &\quad - \textstyle\bigcup_{k=1}^{2n-1} A_{X,k} \\ A_{X,2n+1} &= \{ \ X.a \quad \mid \quad X.a \in Results(X) \\ &\qquad \wedge \quad \forall X.b : X.a \rightarrow X.b \in IDS^+ \\ &\qquad\qquad \Rightarrow \exists m < 2n+1 : X.b \in A_{X,m}\} \\ &\quad - \textstyle\bigcup_{k=1}^{2n} A_{X,k} \end{aligned}$$

where $Arguments(X)$ is the set of argument variables of the function defined over X, and $Results(X)$ is the set of result variables of that same function. The sets $A_{X,k}$, with $1 \leq k \leq m$ form a disjoint partition of $Arguments(X) \cup Results(X)$. The algorithm uses a "backward" sort, hence, the evaluation order corresponds to a decreasing order of index k. Thus, the subsets are in such a way that $A_{X,k}$ contains the arguments which contribute directly to the computation of results in $A_{X,k-1}$.

Having computed the disjoint partitions of $Arguments(X) \cup Results(X)$ for each data-type *X*, the graphs $DS(X)$ are defined as follows:

$$\begin{aligned} DS(X) = \quad & IDS(X) \\ \cup \quad & \{X.a \rightarrow X.b \quad \mid \quad X.a \in A_{X,k} \\ & \qquad\qquad \wedge \quad X.b \in A_{X,k-1} \ \wedge \ 2 \leq k \leq m\} \end{aligned}$$

We are now ready to give the definition of *ordered circular program*.

Definition (Ordered Circular Program). A circular program is an *ordered circular program* if the relation

$$\begin{aligned} EDP = \quad & \textstyle\bigcup_{C \in Constructors} DP(C) \\ & \textstyle\bigcup \{((C,i,a) \rightarrow (C,i,b)) \quad \mid \quad X.a \rightarrow X.b \in DS \\ & \qquad\qquad\qquad\qquad\qquad \wedge \quad < C,i >= X\} \end{aligned}$$

is cycle free. •

If the constructed relation is circular, the program is rejected, although circularities also arise for some programs that are not truly circular. We will return to this subject on Section 5. On the contrary, if the constructed relation is not circular, it can be topologically sorted in order to determine a total order on the variable occurrences of a constructor. That is, on the variables that occur in the program's part that specifies how to compute results when input matches a constructor. This order can be interpreted as a sequence of *abstract computations* to be performed on that constructor. Moreover, the fact that a circular program is ordered also proves that it always terminates for all possible finite inputs[1].

A circularity can originate from two sources. Either the program is not L−ordered (i.e., it is indeed not possible to determine an alternative evaluation order for the circular program) and no interface exist, or it is L−ordered (therefore it would be possible to transform the circular program into a strict one), but **Step 3** selected a *non-compatible interface*. In this case, one could try to enforce a different disjoint partition of $Arguments(X) \cup Results(X)$ by adding artificial dependencies. If a circular program is *ordered*, it is always possible to transform it into a strict, multiple traversal one. The scheduling algorithm defines the interfaces of data-type *X* as follows:

$$Interface(X) \quad = \quad [(A_{X,m}, A_{X,m-1}), \ldots, (A_{X,2}, A_{X,1})]$$

This is the crucial step of Kastens' algorithm and it is this that makes the algorithm polynomial. Many partial orders comply with a IDS relation, but **Step 3** fixes a particular choice: the one that maximizes the interfaces.

Let us now prove that the *Table* program is an ordered circular program. First, we define the sets $A_{X,k}$ of disjoint partitions of variables for all data-type symbols *X* of *Table*. We obtain

[1] provided that the auxiliary functions used in the program also terminate.

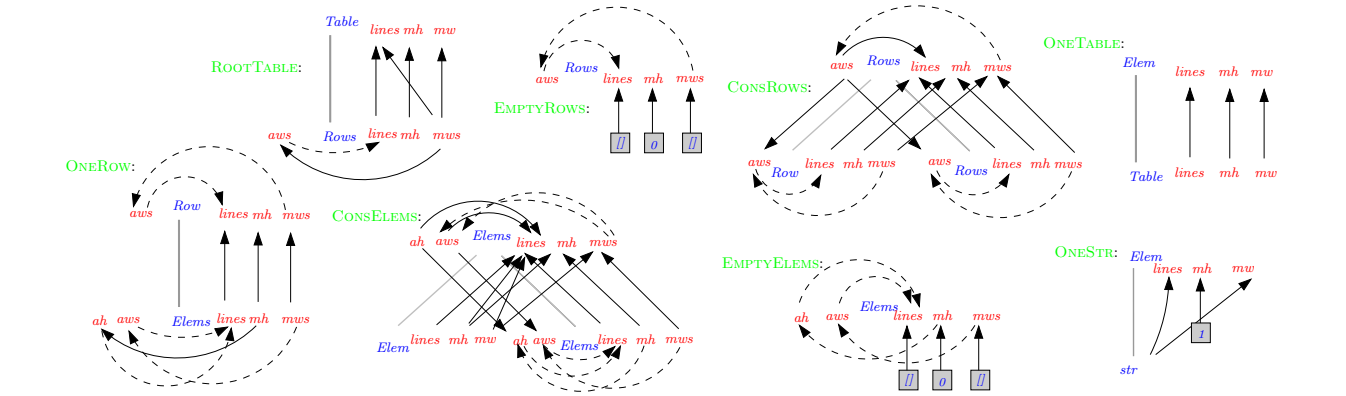

Figure 4. Dependency graph DP (black lines), IDP (black and dashed lines)

$A_{Table,1} = \{Table.lines, Table.mh, Table.mw\}$
$A_{Table,2} = \{\}$
$A_{Rows,1} = \{Rows.lines, Rows.mh\}$
$A_{Rows,2} = \{Rows.aws\}$
$A_{Rows,3} = \{Rows.mws\}$
$A_{Rows,4} = \{\}$
$A_{Row,1} = \{Row.lines, Row.mh\}$
$A_{Row,2} = \{Row.aws\}$
$A_{Row,3} = \{Row.mws\}$
$A_{Row,4} = \{\}$
$A_{Elems,1} = \{Elems.lines\}$
$A_{Elems,2} = \{Elems.ah, Elems.aws\}$
$A_{Elems,3} = \{Elems.mh, Elems.mws\}$
$A_{Elems,4} = \{\}$
$A_{Elem,1} = \{Elem.lines, Elem.mh, Elem.mw\}$
$A_{Elem,2} = \{\}$

Next, we compute the partial orders $DS(X)$ over the variables of $Arguments(X) \cup Results(X)$. As a result we have

$DS(Table) = \{\}$
$DS(Rows) = \{Rows.aws \rightarrow Rows.lines,\ Rows.mws \rightarrow Rows.aws,\ Rows.mws \rightarrow Rows.lines,\ Rows.aws \rightarrow Rows.mh\}$
$DS(Row) = \{Row.aws \rightarrow Row.lines,\ Row.mws \rightarrow Row.aws,\ Row.mws \rightarrow Row.lines,\ Row.aws \rightarrow Row.mh\}$
$DS(Elems) = \{Elems.ah \rightarrow Elems.lines,\ Elems.aws \rightarrow Elems.lines,\ Elems.mh \rightarrow Elems.ah,\ Elems.mh \rightarrow Elems.lines,\ Elems.mws \rightarrow Elems.aws,\ Elems.mws \rightarrow Elems.lines\ Elems.mh \rightarrow Elems.aws,\ Elems.mws \rightarrow Elems.ah\}$
$DS(Elem) = \{\}$

As we can easily notice, all the DS dependency relations are cycle free. Furthermore, we can observe the graphs shown in Figure 4 to notice that the dependency relations DP of the constructors are also cycle free. So, the *Table* program is ordered. We have the following partitions for the data-types symbols:

$Interface(Table) = [(\{\}, \{Table.lines, Table.mh, Table.mw\})]$
$Interface(Rows) = [(\{\}, \{Rows.mws\}), (\{Rows.aws\},\ \{Rows.lines, Rows.mh\})]$
$Interface(Row) = [(\{\}, \{Row.mws\}), (\{Row.aws\},\ \{Row.lines, Row.mh\})]$
$Interface(Elems) = [(\{\}, \{Elems.mh, Elems.mws\}), (\{Elems.ah, Elems.aws\},\ \{Elems.lines\})]$
$Interface(Elem) = [(\{\}, \{Elem.lines, Elem.mh, Elem.mw\})]$

It is worthwhile to note that the scheduling algorithm just broke up the circular definitions of the *Table* circular program into two partitions (or traversals). That is the case of `evalRows`' circular invocation, inside function `evalTable`: the algorithm schedules a two traversal strategy, where the first traversal computes the minimum widths of the table rows (`mws`) and the second traversal computes the table's height (`mh`) and, using the `mws` information (passed to the `aws` argument of the second traversal function), the formatted table lines (`lines`).

3.4 The Visit-Sequence Paradigm

The result of the circular program scheduling algorithm is a set of *interfaces*, that can be interpreted as a sequence of *abstract computations* that have to be performed by a multiple traversal program. In the context of attribute grammars, such abstract computations are usually called *visit-sequences*. They are constructed according to the following idea: for every constructor C a fixed sequence of abstract computations is associated. They abstractly describe which computations have to be performed in every visit of the program to a particular type of nodes in the tree. Such nodes are the instances of C.

Two kinds of abstract computations or instructions are used: `eval` (x) that computes variable x and `visit` (X, v) that visits data-type X for the vth time. In a visit-sequence program, the number of visits to a data-type X is fixed: it corresponds to the number of elements in $Interface(X)$. We denote the number of visits of data-type X by $\mathtt{v}(X)$. Furthermore, each visit v to X, with $1 \leq v \leq \mathtt{v}(X)$, has a fixed *interface*: the element in position v of sequence $Interface(X)$. This *interface* consists of a set of argument variables that may be used during the visit v and another set of result variables that are guaranteed to be computed by the visit v to X. We denote these two sets by $Args_v(X)$ and $Res_v(X)$, where $Args_v(X) = A_{X,2*(\mathtt{v}(X)-v+1)}$, and $Res_v\ (X) = A_{X,2*(\mathtt{v}(X)-v)+1}$.

The visit-sequence of a constructor is usually presented as a list of the two basic instructions. Visit-sequences, however, are the *input* of our techniques to derive purely functional programs. Thus, they are divided into *visit-sub-sequences* $vss(C, v)$, delimited by **begin** v and **end** v, containing the instructions to be performed on visit v to the constructor C, where C is a constructor of X, and $1 \leq v \leq \mathtt{v}(X)$. In order to simplify the presentation, visit-sub-sequences are also annotated with *define* and *usage* variable directives. Every visit-sub-sequence $vss(C, v)$ is annotated with the *interface* of visit v to X. Therefore $vss(C, v)$ is annotated with `arg`$(Args_v(X))$ and `res`$(Res_v(X))$. Every instruction `eval` (x) is annotated with the directive `uses` (bs) that specifies the list of variable occurrences used to evaluate x, *i.e.*, the occurrences that x depends on. The instruction `visit` (X_i, v) causes child i of constructor C, where $C : X_1\ X_2\ \ldots\ X_n \rightarrow X_0$, to be visited for the vth time. The visit uses the variable occurrences of $Args_v(X_i)$ as arguments and returns the variable occurrences of $Res_v(X_i)$. Thus `visit` (X_i, v) is annotated with `inp` and `out` where `inp` is the list of the elements of $Args_v(X_i)$ and `out` is the list of elements of $Res_v(X_i)$.

Figure 5 presents the annotated visit-sub-sequences derived from the *Table* circular program. The boxed variables correspond to values that are defined in one visit-sub-sequence and used in a different one. An implementation of this visit-sequences has to have a special mechanism to handle such occurrences: they induce values that have to be passed between different traversals of the evaluator.

As we have discussed in Section 2, in the multiple traversal evaluator of the table fomatter, the height, the width and the formatted

lines of the nested tables have to be passed from the first to the second traversal of its outer one. This can be seen in the visit-sub-sequences of `ConsElems`: those values are computed in the first sub-sequence and used in the second one.

3.5 Computing Strict 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.

The rules to transform visit-sequences into pure strict functions are described in [22]. Such strict functions 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. Thus, each traversal builds a new visit tree containing in its nodes the values needed in future visits. The functions that represent 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. Next, we present a fragment of the program that is obtained by applying such rules.

The type for the first visit of the strict program is the type of the original tree. The tree type for the second traversal is:

```
data Rows2  = ConsRows2 (Row2, Rows2)
            | EmptyRows2
data Row2   = OneRow2 (Int, Elems2)
data Elems2 = ConsElems2 (Int, Int, String, Elems2)
            | EmptyElems2
```

Note, for example, that type of `ConsElems2` constructor includes now references to the values that have to be passed from the first to its second traversal: the height, the width and the formatted string of the element (string or nested table). There is no reference to the `Elem` type because it induces a single traversal subtree. Next, we show a fragment of the strict, multiple traversal program. We include only the functions where circular definitions occurred in its circular program counterpart.

```
visit_Table1 :: Table -> ([String], Int, Int)
visit_Table1 (RootTable rows) = (lines, mw, mh)
  where (rows2, mws) = visit_Rows1 rows
        mw = (sum mws) + (length mws) + 1
        (lines1, mh) = visit_Rows2 (rows2, mws)
        lines = sepLine (mw, lines1)

visit_Rows1 (ConsRows (row,rows))
 = (ConsRow2 (row2, rows2), mws)
  where (rows2,mws2) = visit_Rows1 rows
        (row2 ,mws1) = visit_Row1  row
        mws = zipwith_max (mws1, mws2)
visit_Rows1 EmptyRows = (NoRow2 , [])

visit_Rows2 (ConsRows2 (row,rows),aws) = (lines,mh)
  where (lines1, mh1) = visit_Row2  (row, aws)
        (lines2, mh2) = visit_Rows2 (rows, aws)
        lines = addSep (aws, lines1, lines2)
        mh = mh1 + mh2 + 1
visit_Rows2 (EmptyRows2, aws) = ([], 0)

visit_Row1 (OneRow elems)
 = (OneRow2 (mh1, elems2), mws1)
  where (elems2, mws1, mh1) = visit_Elems1 elems

visit_Row2 (OneRow2 (mh1,elems),aws) = (lines,mh1)
  where lines1 = visit_Elems2 (elems, mh1, aws)
        lines  = addBorder lines1
```

Observe that the two circular definitions of the original circular program are broken into two traversal functions, both strict in their arguments.

3.5.1 Deforestation by Partial Evaluation

The strict program derived in the previous section relies on (possibly) large number of gluing intermediate data structures to convey information between different traversals. Such redundent structures can, however, be eliminated by using partial evaluation techniques [12]. Indeed, they are static parameters (*i.e.*, known at compile time) of the visit-functions. Thus, we can specialize the functions with these arguments. As a result, we obtain a complete data structure free program [23]. Such programs consist of a set of partially parameterized functions, each performing the computations scheduled for the traversal they represent. The functions return, as one of their results, the function for the next traversal. The main idea is that for each visit-sub-sequence we contruct a function, that besides computing the expected results, also returns the function that defines the following traversal. Any state information (like values inducing inter traversal dependencies) needed in future visits is passed on by partially parameterizing a more general function. Next, we show a fragment of the strict, deforested *Table* program obtained by partial evaluation of the strict one.

```
lambda_RootTable1 :: ([Int] -> (Int,[String]),[Int])
                     -> (Int,Int,[String])
lambda_RootTable1 rows = (lines,mw,mh1)
  where (rows2,mws1) = rows
        mw = (sum mws1)+(length mws)+1
        (lines1, mh1) = rows2 mws1
        lines = sepLine (mw, lines1)

lambda_ConsRows1 (row,rows)
 = (lambda_ConsRow2 (row2,rows2), mws)
  where (rows2,mws2) = rows
        (row2 ,mws1) = row
        mws = zipwith_max (mws1, mws2)
lambda_EmptyRows1 = (lambda_NoRow2,[])

lambda_ConsRows2 (row,rows,aws) = (lines , mh)
  where (lines1,mh1) = row aws
        (lines2,mh2) = rows aws
        lines = addSep (aws, lines1, lines2)
        mh    = mh1 + mh2 + 1
lambda_EmptyRows2 aws = ([],0)

lambda_OneRow1 elems
 = (lambda_OneRow2 (mh1,elems2), mws1)
  where (elems2,mws1,mh1) = elems

lambda_OneRow2 (mh1,elems,aws) = (lines,mh1)
  where lines1 = elems (mh1,aws)
        lines  = addBorder lines1
```

Due to the page limit of this paper we did not show the complete strict and deforested programs. However, the reader can obtain their *Haskell* or *Ocaml* versions using the tools described in Section 6.

Although we have used a first-order circular program as the running example, the techniques introduced by the higher-order extension to attribute grammars [26] directly apply to the transformation of higher-order circular functions, as well. Circular programs modelling algorithms that rely on a large number of traversals tend to have functions with a large number of arguments and results. Such programs, however, can be easily expressed in *Haskell* as a first class attribute grammar [4]. Our techniques directly apply to such *Haskell*-definitions.

The transformation presented in this section constructs *standard* strict multiple traversal programs. These programs can be now further transformed using other well-known techniques. For example, we can use the Hylo system [19] to refactor the derived strict program (which uses explicit recursion) into an hylomorphism. That is

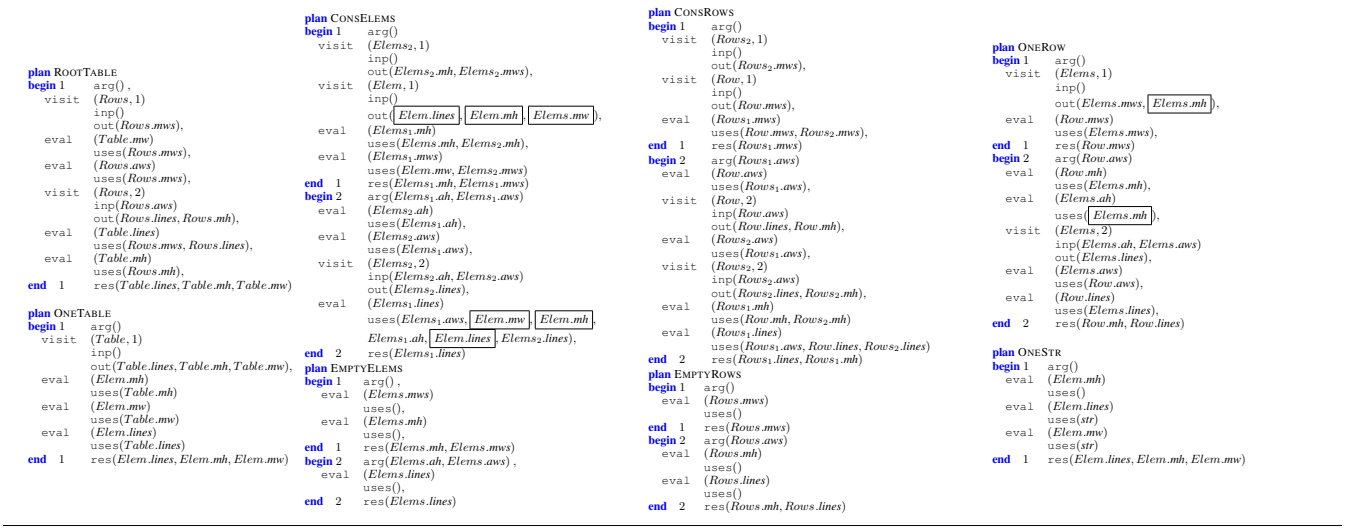

Figure 5. The visit-sub-sequences induced by the *Table* circular program.

to say that we can express a circular program as an hylomorphism. In the next section we present the use of program slicing techniques to slice circular programs.

4. Slicing Circular Programs

Although the programming language community has done a considerable amount of work on program slicing [10, 28], there is little work done on slicing of lazy functional languages. In this section, we use use *standard* slicing techniques to perform static slicing of circular lazy programs. Note that, the standard techniques for static slicing do not directly handle circular definitions due to potential copy-back conflicts as explicitly mentioned in [10].

The transformations presented in the previous section rely heavily in the construction of program dependency graphs and the scheduling of the computations they induce. Such dependency graphs are also the building blocks of program slicing techniques. Having broken the circular dependencies, we can use slicing techniques to perform slicing of circular programs. In fact, in the CircLib library we have expressed the dependency graphs in a generic relation library[2], which implements forward, backward slicing and chopping. Therefore, we can easily define slicing of circular programs by applying such slicing functions to the scheduled dependencies. Next, we present the result of a backward slicing of the circular table formatter. In this slicing we are interested in computing the width of the given table. That is to say that the slicing criteria is the width (result *mw*) of the table.

```
visit_Table1 :: Table -> Int
visit_Table1 (RootTable rows) = mw
  where mws1 = visit_Rows1 rows
        mw   = (sum mws1) + (length mws1) + 1

visit_Rows1 (ConsRows (row,rows)) = mws
  where mws2 = visit_Rows1 rows
        mws1 = visit_Row1 row
        mws  = zipwith_max mws1 mws2
visit_Rows1 EmptyRows = []

visit_Row1 (OneRow els) = mws
  where mws = visit_Elems1 els
```

[2] See the Uminho Haskell Library available at `http://wiki.di.uminho.pt/wiki/bin/view/PURe/PUReSoftware`

```
visit_Elems1 (ConsElems (el,els)) = mw1:mws2
  where mws2 = visit_Elems1 els
        mw1  = visit_Elem1  el
visit_Elems1 EmptyElems = []

visit_Elem1 (OneStr str) = length str
visit_Elem1 (OneTable table) = mw1
  where mw1 = visit_Table1 table
```

The result of the backward slicing is the sub-program that includes the definitions of the original one that contribute to compute the width of the table. All other definitions are *sliced-out*.

In this simple example, the resulting program performs a single tree traversal. For more complicated programs, however, the result of a slice may be a program that performs multiple tree traversals. In this case we can generate one of the three implementations presented in the paper, that is circular, strict or deforested programs. This is the case if we consider, in our example, as the slicing criteria the result that computes the table (*lines*). The resulting programs are very similar to the ones we have presented, with the exception that the top function returns one result only: the formatted table.

5. Class of Programs Considered

In the previous section we have studied the *Table* language and processor in great detail. It should be noticed that this running example is just a *simple* two traversal program. Things get much more complicated if we consider more practical examples. For example, in [25] we have presented an optimal pretty printing algorithm that performs four traversals over the abstract syntax tree describing the program to print. As a consequence, the strict version of that program needs four gluing intermediate data structures to convey information between the different traversals. Moreover, the scheduling of the four traversals is not trivial at all. Like in the *Table* example, it has several subtrees that have to be traversed in different visits to the parents. Indeed, we believe that would be extremely difficult to hand-write such a program in a strict setting. In that paper, however, we have expressed the pretty printing as an attribute grammar and we have derived its strict implementation.

Although we can derive strict implementations from circular definitions, our techniques do not consider all possible *well-formed circular programs*. By well-formed circular programs we mean the set of circular program that can be evaluated without inducing non-termination. It is well-known that AG scheduling algorithm per-

forms an approximation on the dependencies to compute the evaluation order. As a consequence, there are programs that are considered circular by the scheduling algorithm, although no circularity really exists. Moreover, there are other circular programs that do rely on dynamic scheduling (lazy evaluation) to compute the evaluation order. One example of such circular programs is the breadth-first numbering algorithm presented in [18].

Nevertheless, most of algorithms needed in practical examples belong to the class of ordered circular programs. Thus, they can be analyzed and transformed by our techniques. The single example we found in the literature that can not be (directly) considered is the breadth-first numbering. However, the tricky example presented by Okasaki can be slightly modified and expressed as an ordered circular program[3].

6. Tools and Libraries for Circular Programming

The techniques presented in this paper have been implemented: a library has been written and two tools constructed.

The CircLib Library: It is a library written in *Haskell* to manipulate circular programs (its API is given in appendix A). This library introduces two data types to model circular programs and visit sequences in *Haskell*, and it defines functions that implement all the formal definitions and techniques presented in this paper. It also includes slicing functions. CircLib is a reusable library that can be used to break-up circular dependencies. It can be used not only to transform circular lazy programs into strict ones, but also to express circular programs as hylomorphisms, to implement attribute grammar systems, to express circular XML transformations, etc. This library is the building block of the two tools described next.

The HaCirc Tool: It is an *Haskell* refactor. It refactors an *Haskell* circular program into its strict counterparts; it can be seen as a strictification tool. The *HaCirc* tool is also able to slice circular programs.

The OCirc Tool: In order to allow *Ocaml* programmers to express their multiple traversal programs in this elegant style of circular programming we have a similar tool for *Ocaml*. This tool transforms circular programs written in the *Ocaml* notation, into correct strict *Ocaml* programs.

There are two versions of the *HaCirc* and *OCirc* tools: a batch version that given as input a circular *Haskell*(*Ocaml*) program generates its strict/deforested *Haskell*(*Ocaml*) program, and a web-based interactive tool(s) that allows the tool(s) to be used online[4]. The reader may use these tools to produce, for example, the *Ocaml* or *Haskell* strict programs of the *repmin* and the *Table* processor from the circular definitions presented in this paper (and available there). The slicing of circular programs can also be performed using the online tools.

7. Benchmarks

In order to benchmark the different implementations of circular programs, we conducted several experiments. We show results of two circular programs: the *Table* formatter example and the processor of the *MicroC* language. *MicroC* is a tiny subset of the *C* language. The former induces a simple two traversal strict program, while the later induces a six traversal program. We consider the three implementations presented in this paper, *i.e.*, lazy, strict and deforested programs. The results presented next were obtained in an Intel Centrino 1.4 GHz with 512 MB of RAM memory, under a Linux Mandrake 10.0 OS. We have used the `ghc 6.4` compiler.

The Table Formatter: The three *Table* formatters were tested with three different input tables: a table with depth 150 (a typical 3x3 matrix, with one nested table, with depth 149), one with depth 250 and another with depth 350. The results obtained are presented in Table 1.

		Circular		Strict		Deforested	
	Table depth	Mem (Kb)	Time (sec)	Mem (Kb)	Time (sec)	Mem (Kb)	Time (sec)
Haskell	150	260	72.85	140	71.6	130	68.55
	250	450	266.69	240	260.00	220	255.65
	350	600	677.04	320	646.95	300	642.93

Table 1. Performance results of the three different Table formatters.

The results show that the three implementations have similar running times, although the deforested program is always slightly faster than the others. In terms of memory consumption, the deforested consumes half of the memory needed by the circular program. A two traversal program, however, does not forces the lazy mechanism to keep a large set of suspended computations. Next, we consider a more complex example, that relies on a six traversal strategy.

The *MicroC* Processor: The *MicroC* language processor generates assembly for a simple stack-based machine and it includes the advanced pretty-printing algorithm that performs four traversals to compute its prettiest representation [25]. As input we consider typical *MicroC* programs, with 1360, 2720 and 4080 lines. The runtimes (in seconds) are the accumulation of 10 executions. The memory consumption refers to the memory used in one run, and it was obtained with the built-in ghc memory profiler.

		Circular		Strict		Deforested	
	Input size	Mem (Kb)	Time (sec)	Mem (Kb)	Time (sec)	Mem (Kb)	Time (sec)
Haskell	1360	1600	17.63	3400	16.41	900	5.9
	2720	2800	36.06	6100	32.44	1600	12.21
	4080	4400	54.48	12000	47.75	3000	18.49

Table 2. Performance results of the three *MicroC* processors.

The above results show that the deforested *Haskell* program has the best running time of the different implementations of the *MicroC* processor: it is 2.8 times faster than the lazy program. The deforested implementation is also always more efficient than the strict one: 2.6 times faster. One would expect, however, that the aggressive optimizations performed by this advanced compiler would be able to perform the deforestation automatically, by using techniques like the cata-build rule. In fact, the strict implementation builds (intermediate) trees that are later consumed. However, as one can see in the definition of the strict *Table* program, the function that builds the intermediate structure also returns additional results. Thus, the cata-build rule does not apply [8] and the compilers are not able to perform such optimizations. This can also be seen in the results of the memory usage of the programs.

8. Conclusion

This paper presented techniques and tools to model and manipulate circular programs. These techniques transform circular programs into strict, purely functional programs. Partial evaluation and slicing techniques are used to improve the performance of the evaluators and to slice circular lazy programs, respectively. These techniques have been implemented to build the *Haskell* library *CircLib* which has been used to construct two tools to model and manipulate circular programs in *Haskell* and *Ocaml*. As a result, we can model in a strict or lazy setting a multiple traversal algorithm as a single traversal circular function without the need of additional redundant intermediate data structures and having to define complex

[3] In fact, the definition of breadth-first numbering in a strict setting was proposed by Okasaki as an exercise in one IFIP WG 2.8 meeting.

[4] The tools are available online at `http://wiki.di.uminho.pt/wiki/bin/view/Joao/CircularProgramming`

traversal scheduling strategies. Circular definitions are well-known and heavily used in the AG community. With this work we make this powerful style of programming available to other programming paradigms, namely the non-lazy functional one. Moreover, the presented slicing techniques allows the programmer to extract different aspects of a circular program. Finally, the first experimental results show that the strict deforested *Haskell* programs are more efficient than the *Haskell* lazy circular programs.

A. The CircLib *Haskell* library

In this section we present the API of the *Haskell* library that implements the re-schedulling of the circular definitions. We start by defining a data-type CP, to represent circular programs, and the functions that manipulate it[5]:

```
data CP
  = CP {constrs    :: [Constr],
        types      :: [DT],
        prods      :: Map Constr [DT],
        args       :: Map DT [VarName],
        results    :: Map DT [VarName],
        deps       :: Map Constr [Dep]
        semantics  :: Map Constr (Map VarName Function)}
type Var = (Constr, Int, String)
type Dep = ((Int, Name), (Int, Name))
```

where $Constr$, DT, $VarName$ and $Function$ are of type $String$.

```
dp          :: CP → Rel Var Var
idp         :: CP → Rel Var Var
ids         :: CP → Rel (DT, Name) (DT, Name)
a           :: CP → DT → Int → Set (DT, Name)
ds          :: CP → DT → Rel (DT, Name) (DT, Name)
edp         :: CP → Rel Var Var
isOrdered   :: CP → Bool
interface   :: CP → DT → Interface
type Interface = [(Set (DT, Name), Set (DT, Name))]
```

We model visit-sequences we the following data-structures and function.

```
data VisitSequences = VS (Map Constr [VisitSubSequence])
data VisitSubSequence = VSS { n             :: Int,
                              prod          :: [DT],
                              arg           :: [VarName],
                              res           :: [VarName],
                              instructions  :: [Instruction] }
data Instruction = Eval  { variable  :: Var,
                           uses      :: [Var]}
                 | Visit {visit      :: (Int, Int),
                          inp        :: [Name],
                          out        :: [Name] }
```

visit_sequences :: $CP \rightarrow VisitSequences$

The *slicing* of circular programs is perfomed by the functions:

```
backward_slice :: CP → Criteria → VisitSequences
forward_slice  :: CP → Criteria → VisitSequences
type Criteria = [VarName]
```

References

[1] D. S. Atze Dijkstra. Typing haskell with an attribute grammar (part i). Technical Report UU-CS-2004-037, Institute of Information and Computing Sciences, Utrecht University, 2004.

[2] R. S. Bird. Using circular programs to eliminate multiple traversals of data. *Acta Inf*, 21:239–250, 1984.

[3] O. Danvy and M. Goldberg. There and back again. In *ICFP '02: Seventh ACM SIGPLAN international conference on Functional programming*, pages 230–234, New York, USA, 2002. ACM Press.

[4] O. de Moor, K. Backhouse, and S. D. Swierstra. First-class attribute grammars. *Informatica (Slovenia)*, 24(3), 2000.

[5] O. de Moor, S. Peyton-Jones, and E. Van Wyk. Aspect-oriented compilers. *Lecture Notes in Computer Science*, 1799, 2000.

[6] A. Dijkstra. *Stepping through Haskell*. PhD thesis, Computer Science Depart., Utrecht University, The Netherlands, November 2005.

[7] J. Engelfriet and G. Filé. Simple multi-visit Attribute Grammars. *Journal of Computer and System Sciences*, 24(3):283–314, 1982.

[8] J. Fernandes and J. Saraiva. Calculating circular programs. *(in preparation)*, 2006.

[9] R. Hinze and J. Jeuring. Generic Haskell: Practice and theory. In *Summer School on Generic Programming*, 2002.

[10] S. Horwits and T. Reps. The Use of Program Dependence Graphs in Software Engineering. In *14th International Conference on Software Engineering*, pages 392–411, Melbourne, Australia, may 1992. ACM.

[11] T. Johnsson. Attribute grammars as a functional programming paradigm. In *Functional Programming Languages and Computer Architecture*, pages 154–173, 1987.

[12] N. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall Inernational Series in Computer Science. Prentice-Hall, 1993.

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

[14] D. E. Knuth. Semantics of Context-free Languages. *Mathematical Systems Theory*, 2(2):127–145, June 1968. Correction: *Mathematical Systems Theory* 5, 1, pp. 95-96 (March 1971).

[15] M. Kuiper and D. Swierstra. Using attribute grammars to derive efficient functional programs. In *Computing Science in the Netherlands CSN'87*, November 1987.

[16] J. L. Lawall. Implementing Circularity Using Partial Evaluation. In *Proceedings of the Second Symposium on Programs as Data Objects PADO II*, volume 2053 of *LNCS*, May 2001.

[17] S. Marlow and S. P. Jones. The new GHC/Hugs Runtime System. 1999.

[18] C. Okasaki. Breadth-first numbering: lessons from a small exercise in algorithm design. *ACM SIGPLAN Notices*, 35(9):131–136, 2000.

[19] Y. Onoue, Z. Hu, H. Iwasaki, and M. Takeichi. A calculational fusion system HYLO. In *Algorithmic Languages and Calculi*, pages 76–106, 1997.

[20] M. Pennings. *Generating Incremental Evaluators*. PhD thesis, Depart. of Comp. Science, Utrecht Univ., The Netherlands, 1994.

[21] T. Reps and T. Teitelbaum. *The Synthesizer Generator*. Springer, 1989.

[22] J. Saraiva. *Purely Functional Implementation of Attribute Grammars*. PhD thesis, Department of Computer Science, Utrecht University, The Netherlands, December 1999.

[23] J. Saraiva and D. Swierstra. Data Structure Free Compilation. In Stefan Jähnichen, editor, *8th International Conference on Compiler Construction, CC/ETAPS'99*, volume 1575 of *LNCS*, pages 1–16. Springer-Verlag, March 1999.

[24] J. Saraiva, D. Swierstra, and M. Kuiper. Strictification of Computations on Trees. In R. Linz, editor, *Third Latin-American Conference on Functional Programming - CLAPF'99 - Recife*, Mar. 1999.

[25] D. Swierstra, P. Azero, and J. Saraiva. Designing and Implementing Combinator Languages. In D. Swierstra, P. Henriques, and J. Oliveira, editors, *Third Summer School on Advanced Functional Programming*, volume 1608 of *LNCS Tutorial*, pages 150–206, September 1999.

[26] D. Swierstra and H. 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.

[27] D. Swierstra and P. Azero. Attribute grammars in a functional style. In *Systems Implementation 2000*, Berlin, 1998. Chapman & Hall.

[28] F. Tip. A Survey of Program Slicing Techniques. Technical report CS-R9438, CWI - Computer Science, Department of Software Technology, Amsterdam, February 1994.

[29] J. Voigtländer. Using circular programs to deforest in accumulating parameters. *Higher-Order and Symbolic Computation*, 17:129–163, 2004.

[5] These functions correspond to the *Haskell* versions of the formal definitions presented in Section 3.3.