View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Universidade do Minho: RepositoriUM

Universidade do Minho
Escola de Engenharia

João Paulo de Sousa Ferreira Fernandes

Design, Implementation and Calculation of Circular Programs

Setembro 2008

João Paulo de Sousa Ferreira Fernandes **Design, Implementation and Calculation of Circular Programs**

UMinho | 2008

Universidade do Minho
Escola de Engenharia

João Paulo de Sousa Ferreira Fernandes

Design, Implementation and Calculation of Circular Programs

Doutoramento em Informática
Ramo Fundamentos da Computação

Trabalho efectuado sob a orientação do
Professor Doutor João Alexandre Saraiva

Setembro 2008

É AUTORIZADA A REPRODUÇÃO INTEGRAL DESTA TESE APENAS PARA EFEITOS DE INVESTIGAÇÃO, MEDIANTE DECLARAÇÃO ESCRITA DO INTERESSADO, QUE A TAL SE COMPROMETE;

Universidade do Minho, ___/___/_____

Assinatura: _____

Acknowledgements

This thesis would not be real without the support of my friends, family and supervisors.

I sincerely thank all for their encouragement and commitment.

Several institutions contributed to this thesis in different ways. The research was mainly supported by Fundação para a Ciência e Tecnologia (FCT), grant No. SFRH/BD/19186/2004. The Department of Informatics of the University of Minho provided financial support for me to participate in the ACM SIGPLAN 2007 Workshop on Partial Evaluation and Program Manipulation, the ACM SIGPLAN Professional Activities Committee provided financial support for me to participate in the ACM SIGPLAN Haskell Workshop 2007 and the NATO Science for Peace and Security Programme provided financial support for me to participate in the Summer School Marktoberdorf 2007.

Abstract

Circular programming is a powerful technique to express multiple traversal algorithms as a single traversal function in a lazy setting. Such a (virtual) circular program may contain *circular definitions*, that is, arguments of function calls that are also results of that same calls. Although circular definitions always induce non-termination under a strict evaluation mechanism, they can sometimes 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.

In this Thesis, we present our studies on the design, implementation and calculation of circular programs. We start by developing techniques to transform circular programs into strict ones. Then, we introduce calculation rules to obtain circular programs from strict equivalents, both in the context of pure and monadic programming. Because we use calculation techniques we guarantee that the resulting circular programs are equivalent to the strict ones we start with. In this Thesis, we also perform a series of benchmarks comparing the running performances of circular programs and the programs we are able to derive from circular programs.

Abstract

A utilização de programas circulares na implementao de algoritmos de programação é uma técnica poderosa que permite, num paradigma *lazy*, implementar soluções que efectuam múltiplas travessias sobre uma ou mais estruturas de dados como um programa que efectua apenas uma travessia sobre uma única estrutura de dados. Num programa (virtualmente) circular podem ocorrer *definições circulares*, isto é, invocações de funções onde um argumento da invocação é, ao mesmo tempo, um resultado da mesma invocação. Embora este tipo de definições induza não terminação num paradigma estrito, a verdade é que, num paradigma *lazy*, elas podem ser desde logo executadas utilizando uma estratégia baseada em *lazy evaluation*: a máquina *lazy* é capaz de determinar o escalonamento correcto das computações, se ele existir. Na verdade, utilizando este método de programação, o(a) programador(a) não tem de definir nem de escalonar as diferentes travessias, uma vez que apenas uma função necessita de ser implementada. Para além disso, porque existe apenas função, e uma vez que essa função realiza apenas uma travessia, o(a) programador(a) também não é forçado a definir estruturas de dados intermédias para *colar* as diferentes travessias.

Nesta Tese são apresentados os nossos estudos relativos ao desenho, implementação e cálculo de programas circulares. Começamos por desenvolver técnicas de transformação de programas circulares em programas estritos. Depois apresentamos regras de cálculo que permitem obter programas circulares a partir de estritos, equivalentes, tanto no contexto de funções puras como no contexto de funções monádicas. Uma vez que, neste trabalho, utilizamos técnicas de cálculo de programas, é possível garantir a correção da transformação que propomos. Por fim, realizamos uma bateria de testes que permitem comparar a *performance* de programas circulares com a dos programas que derivamos a partir deles.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Circular programs were first proposed by Bird (1984) 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 leaf values replaced by the minimum leaf 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 leaf values by the minimum value, therefore producing the desired tree result. This straightforward solution is as follows.

$$\begin{aligned} &\textbf{data}\ \mathit{LeafTree} = \mathit{Leaf}\ \ \mathit{Int} \\ &\qquad\qquad\qquad\quad \mid\ \mathit{Fork}\ (\mathit{LeafTree}, \mathit{LeafTree}) \end{aligned}$$

$$\mathit{transform} :: \mathit{LeafTree} \rightarrow \mathit{LeafTree}$$

$$
\begin{aligned}
&transform\ t = replace\ (t, tmin\ t)\\
&\\
&tmin :: LeafTree \rightarrow Int\\
&tmin\ (Leaf\ n) \quad\ = n\\
&tmin\ (Fork\ (l, r)) = min\ (tmin\ l)\ (tmin\ r)\\
&\\
&replace :: (LeafTree, Int) \rightarrow LeafTree\\
&replace\ (Leaf\ \_, m) \quad\ = Leaf\ m\\
&replace\ (Fork\ (l, r), m) = Fork\ (replace\ (l, m), replace\ (r, m))
\end{aligned}
$$

However, a two traversal strategy is not essential to solve the *repmin* problem. An alternative solution can, on a single traversal, compute the minimum leaf value and, at the same time, replace all values by that minimum value. Bird showed how the single traversal program, presented next, may be obtained by transforming the original program using the following techniques: tupling, fold-unfold and circular programming[1].

$$
\begin{aligned}
&repmin\ (Tip\ n \quad\ , m) = (Tip\ m, \quad\ n)\\
&repmin\ (Fork\ (l, r), m) = (Fork\ (t_1, t_2), min\ m_1\ m_2)\\
&\quad \textbf{where}\ (t_1, m_1) = repmin\ (l, m)\\
&\qquad\qquad (t_2, m_2) = repmin\ (r, m)\\
&\\
&transform\ t = nt\\
&\quad \textbf{where}\ (nt, m) = repmin\ (t, m)
\end{aligned}
$$

Notice the circularity in the above program: m is both an argument and a result of the *repmin* call, in the *transform* function. Although this circular definition seems to induce both a cycle and non-termination of this program, the fact is that using a *lazy* language, the *lazy* evaluation machinery is able to determine, at runtime, the right order to evaluate such circular definition.

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. Indeed, using this style of circular programming, the programmer does not have to concern him/herself with the definition and the schedul-

[1] We review Bird's transformation in detail in Chapter 3.

ing 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.

Bird's approach, however, has a severe drawback since it preserves partial correctness only. The circular programs derived using Bird's method are not guaranteed to terminate.

Circular programs are used in the construction of Haskell compilers (Marlow and Jones 1999; Hinze and Jeuring 2002), to express pretty printing algorithms (Swierstra et al. 1999), breadth-first traversal strategies (Okasaki 2000), type systems (Dijkstra and Swierstra 2004) and aspect-oriented compilers (de Moor et al. 2000). As an optimization technique, circular programs are used, for example, in the deforestation of accumulating parameters (Voigtländer 2004). Circular programs can also be obtained through partial evaluation (Lawall 2001) and continuations (Danvy and Goldberg 2002). As Johnsson (1987) and Swierstra and Kuiper (Kuiper and Swierstra 1987) originally showed, circular programs are the natural representation of attribute grammars in a lazy setting (Swierstra and Azero 1998; de Moor et al. 2000; Saraiva 1999; Dijkstra 2005).

1.1 Contributions

The main contributions of this Thesis are:

- a program calculation rule, in the style of the shortcut deforestation rule, to obtain circular programs from strict ones;
- the formal proof that such rule is correct;
- the study of a program calculation rule, developed in the same setting as the above one, but in the monadic context;
- such monadic rule was also proved correct;

- a strictification technique, based on well-known Attribute Grammars techniques, that we have developed and applied to transform circular programs into strict and strict deforested equivalents;
- a sistematic benchmark comparing the performances of circular, strict and strict deforested programs.

1.2 Structure of the Thesis

This Thesis is organized as follows: in Chapter 2 we present techniques to model circular lazy programs in a strict, purely functional setting. A motivating example, that will guide the presentation for the Chapter, is also presented. The circular solution to such example is then transformed, by applying the Attribute Grammar based techniques that we propose, into a strict and a strict and deforested equivalent programs. In Chapter 3 we present a shortcut deforestation technique to calculate circular programs. The technique we propose takes as input the composition of two functions, such that the first builds an intermediate structure and some additional context information which are then processed by the second one, to produce the final result. Our transformation into circular programs achieves intermediate structure deforestation and multiple traversal elimination. Furthermore, the calculated programs preserve the termination properties of the original ones. In Chapter 4, we propose an extension to the new form of fusion presented in Chapter 3, but in the context of monadic programming. Our extension is also provided in terms of generic calculation rules, that can be uniformly defined for a wide class of data types and monads. In Chapter 5, we present the implementation of the techniques formally introduced in Chapter 2 as a *Haskell* library: the *CircLib* library. Using this library, we have constructed two tools to transform *Haskell* and *Ocaml* based circular programs into their strict counterparts. Furthermore, we also conduct the first systematic benchmarking of circular, strict and deforested programs. Finally, in Chapter 6, we draw some conclusions concerning the present techniques.

Chapter 2

Strictification of Circular Programs

This Chapter 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.

2.1 Introduction

Circular lazy programs, as introduced by Bird (1984), 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 us-

ing 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.

In this Chapter 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 (Knuth 1968). Moreover, we use partial evaluation techniques to derive deforested versions of the strict programs. Furthermore, 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).

This Chapter is organized as follows: Section 2.2 presents the notation and the running example used throughout the Chapter. Section 2.3 presents the derivation of strict programs from circular ones. Section 2.4 presents the slicing of circular programs. In Section 2.5 we discuss the class of circular programs considered. Section 2.6 shows our conclusions.

2.2 Circular Programs

2.2.1 Notation

To demonstrate our techniques, we use the language given in Fig. 2.1. 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.

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

Figure 2.1: Abstract syntax

2.2.2 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 re-

alistic multiple traversal problems, like for example the four traversal pretty printing algorithm presented in (Swierstra et al. 1999), the programmer has to define additional gluing data structures to pass 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. 2.2 shows an example of a possible input (left) and correspondent output (right).

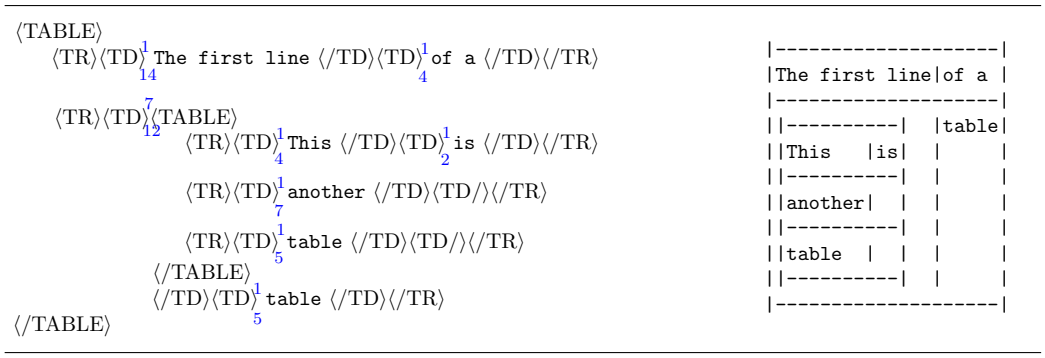

Figure 2.2: HTML Table Formatting

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 define 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 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 2.3.

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

$$
\begin{aligned}
&\textbf{data}\ \mathit{Table} &&= \mathit{RootTable}\ \mathit{Rows}\\
&\textbf{data}\ \mathit{Rows} &&= \mathit{EmptyRows}\\
&&&\mid \mathit{ConsRows}\ (\mathit{Row}, \mathit{Rows})\\
&\textbf{data}\ \mathit{Row} &&= \mathit{OneRow}\ \mathit{Elems}
\end{aligned}
$$

$$
\begin{aligned}
&\textbf{data}\ \mathit{Elems} = \mathit{EmptyElems} \\
&\qquad\qquad \mid \mathit{ConsElems}\ (\mathit{Elem}, \mathit{Elems}) \\
&\textbf{data}\ \mathit{Elem}\ \ = \mathit{OneStr}\ \mathit{String} \\
&\qquad\qquad \mid \mathit{OneTable}\ \mathit{Table}
\end{aligned}
$$

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.

$$
\begin{aligned}
&\mathit{evalTable} :: \mathit{Table} \rightarrow ([\mathit{String}], \mathit{Int}, \mathit{Int}) \\
&\mathit{evalTable}\ (\mathit{RootTable}\ \mathit{rows}) = (\mathit{lines}, \mathit{mh}, \mathit{mw}) \\
&\quad \textbf{where}\ (\mathit{lines}_1, \mathit{mh}_1, \mathit{mws}) = \mathit{evalRows}\ (\mathit{rows}, \mathit{mws}) \\
&\qquad\quad \mathit{mh}\ \ = \mathit{mh}_1 + 2 \\
&\qquad\quad \mathit{mw}\ \ = (\mathit{sum}\ \mathit{mws}) + (\mathit{length}\ \mathit{mws}) + 1 \\
&\qquad\quad \mathit{lines} = \mathit{sepLine}\ (\mathit{mws}, \mathit{lines}_1)
\end{aligned}
$$

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 $[12, 5]$ (second). The result of $\mathit{zipwith}_{Max}$ is $[14, 5]$, that is, the maximum width of each column.

$$
\begin{aligned}
&\mathit{evalRows}\ (\mathit{ConsRows}\ (\mathit{row}, \mathit{rows}), \mathit{aws}) = (\mathit{lines}, \mathit{mh}, \mathit{mws}) \\
&\quad \textbf{where}\ (\mathit{lines}_1, \mathit{mh}_1, \mathit{mws}_1) = \mathit{evalRow}\ (\mathit{row}, \mathit{aws}) \\
&\qquad\quad (\mathit{lines}_2, \mathit{mh}_2, \mathit{mws}_2) = \mathit{evalRows}\ (\mathit{rows}, \mathit{aws}) \\
&\qquad\quad \mathit{mh}\ \ = \mathit{mh}_1 + \mathit{mh}_2 + 1 \quad \text{-- } + 1 \text{ is for the separator} \\
&\qquad\quad \mathit{mws} = \mathit{zipwith}_{Max}\ (\mathit{mws}_1, \mathit{mws}_2)
\end{aligned}
$$

$$lines = addSep\ (aws, lines_1, lines_2)$$

$$evalRows\ (EmptyRows, aws) = ([\,], -1, [\,])$$

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.

$$\begin{aligned} &evalRow\ (OneRow\ elems, aws) = (lines, mh, mws) \\ &\quad \textbf{where}\ (lines_1, mh, mws) = evalElems\ (elems, mh, aws) \\ &\qquad\quad lines = addBorder\ lines_1 \end{aligned}$$

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.

$$\begin{aligned} &evalElems\ (ConsElems\ (elem, elems), ah, aws) = (lines, mh, mws) \\ &\quad \textbf{where}\ aws_2 = tail\ aws \\ &\qquad\quad (lines_1, mh_1, mw_1)\ \ = evalElem\ elem \\ &\qquad\quad (lines_2, mh_2, mws_2) = evalElems\ (elems, ah, aws_2) \\ &\qquad\quad mws\ = mw_1 : mws_2 \\ &\qquad\quad mh\ \ \ = max\ (mh_1, mh_2) \\ &\qquad\quad lines = glue\ (aws, mw_1, ah, mh_1, lines_1, lines_2) \end{aligned}$$

$$evalElems\ (EmptyElems, ah, aws) = ([\,], 0, [\,])$$

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

$$evalElem\ (OneStr\ str) = ([str], 1, length\ str)$$

$$\begin{aligned} &evalElem\ (OneTable\ table) = (lines_1, mh_1, mw_1) \\ &\quad \textbf{where}\ (lines_1, mh_1, mw_1) = evalTable\ table \end{aligned}$$

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.

2.3 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.

2.3.1 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 2.3 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

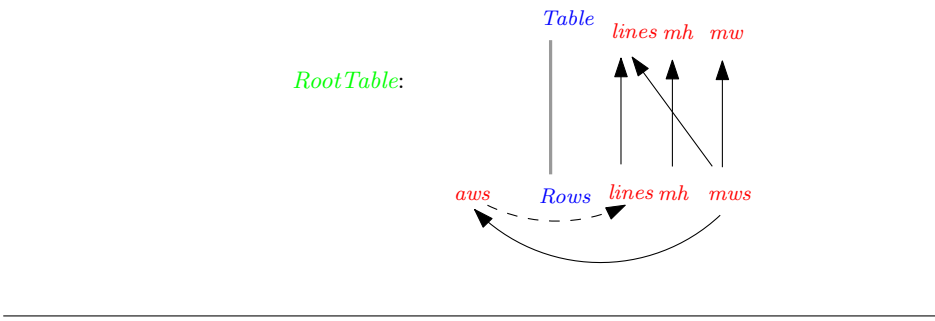

Figure 2.3: Dependency graph of function `evalTable`

that data type in the different induced graphs. Notice, for example, that 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 2.3, 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.

2.3.2 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:

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

$$\begin{aligned} \text{with} \quad args_i =& \{arguments\ of\ the\ ith\ function\ defined \\ & over\ elements\ of\ type\ X\} \\ results_i =& \{results\ of\ the\ ith\ function\ defined \\ & over\ elements\ of\ type\ X\} \end{aligned}$$

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 func-

tional 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 2.4 (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 defini-

tion 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

$$DP(C) \cup \bigcup_{i=0}^{n} PO(\langle C, i\rangle), \text{ where } \langle C, i\rangle = X_i,$$

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 (Engelfriet and Filé 1982), Engelfried and Filè proved that deciding whether an attribute grammar is partitionable or not is a NP-complete problem. Kastens (1980) 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.

2.3.3 Ordered Circular Programs

In this section we present an adaptation of Kastens' attribute scheduling algorithm (Kastens 1980; Reps and Teitelbaum 1989; Pennings 1994) 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:

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

The circular program is not ordered if IDP is cyclic.

Figure 2.4 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 2.4 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).

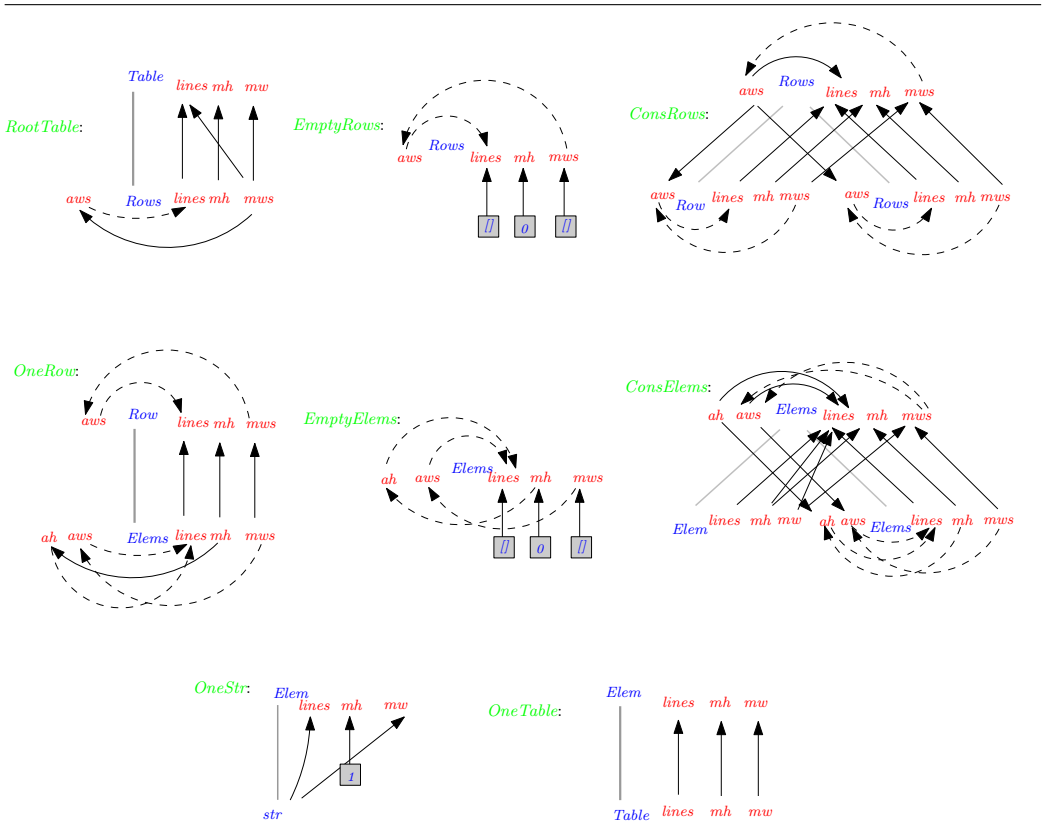

Figure 2.4: Dependency graph DP (black lines), IDP (black and dashed lines)

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:

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

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

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

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

$$A_{X,1} = Results(X) - \{X.a \mid X.a \rightarrow X.b \in IDS^+\}$$

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

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

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:

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

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

$$\begin{aligned} EDP = \; & \bigcup_{C \in Constructors} DP(C) \\ & \bigcup \{((C, i, a) \rightarrow (C, i, b)) \mid X.a \rightarrow X.b \in DS \wedge \langle C, i \rangle = X\} \end{aligned}$$

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 2.5. 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

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

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

$$\begin{array}{lll}
A_{Table,1} & = \{\text{Table}.lines, \text{Table}.mh, & A_{Row,3} = \{\text{Row}.mws\} \\
 & \quad \text{Table}.mw\} & A_{Row,4} = \{\} \\
A_{Table,2} & = \{\} & A_{Elems,1} = \{\text{Elems}.lines\} \\
A_{Rows,1} & = \{\text{Rows}.lines, \text{Rows}.mh\} & A_{Elems,2} = \{\text{Elems}.ah, \text{Elems}.aws\} \\
A_{Rows,2} & = \{\text{Rows}.aws\} & A_{Elems,3} = \{\text{Elems}.mh, \text{Elems}.mws\} \\
A_{Rows,3} & = \{\text{Rows}.mws\} & A_{Elems,4} = \{\} \\
A_{Rows,4} & = \{\} & A_{Elem,1} = \{\text{Elem}.lines, \text{Elem}.mh, \\
A_{Row,1} & = \{\text{Row}.lines, \text{Row}.mh\} & \quad \text{Elem}.mw\} \\
A_{Row,2} & = \{\text{Row}.aws\} & A_{Elem,2} = \{\}
\end{array}$$

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

$$
\begin{aligned}
DS(\mathit{Table}) &= \{\} \\
DS(\mathit{Rows}) &= \{\mathrm{Rows}.\mathit{aws} \rightarrow \mathrm{Rows}.\mathit{lines}, \mathrm{Rows}.\mathit{mws} \rightarrow \mathrm{Rows}.\mathit{aws}, \\
&\quad\ \mathrm{Rows}.\mathit{mws} \rightarrow \mathrm{Rows}.\mathit{lines}, \mathrm{Rows}.\mathit{aws} \rightarrow \mathrm{Rows}.\mathit{mh}\} \\
DS(\mathit{Row}) &= \{\mathrm{Row}.\mathit{aws} \rightarrow \mathrm{Row}.\mathit{lines}, \mathrm{Row}.\mathit{mws} \rightarrow \mathrm{Row}.\mathit{aws}, \\
&\quad\ \mathrm{Row}.\mathit{mws} \rightarrow \mathrm{Row}.\mathit{lines}, \mathrm{Row}.\mathit{aws} \rightarrow \mathrm{Row}.\mathit{mh}\} \\
DS(\mathit{Elems}) &= \{\mathrm{Elems}.\mathit{ah} \rightarrow \mathrm{Elems}.\mathit{lines}, \mathrm{Elems}.\mathit{aws} \rightarrow \mathrm{Elems}.\mathit{lines}, \\
&\quad\ \mathrm{Elems}.\mathit{mh} \rightarrow \mathrm{Elems}.\mathit{ah}, \mathrm{Elems}.\mathit{mh} \rightarrow \mathrm{Elems}.\mathit{lines}, \\
&\quad\ \mathrm{Elems}.\mathit{mws} \rightarrow \mathrm{Elems}.\mathit{aws}, \mathrm{Elems}.\mathit{mws} \rightarrow \mathrm{Elems}.\mathit{lines} \\
&\quad\ \mathrm{Elems}.\mathit{mh} \rightarrow \mathrm{Elems}.\mathit{aws}, \mathrm{Elems}.\mathit{mws} \rightarrow \mathrm{Elems}.\mathit{ah}\} \\
DS(\mathit{Elem}) &= \{\}
\end{aligned}
$$

As we can easily notice, all the DS dependency relations are cycle free. Furthermore, we can observe the graphs shown in Figure 2.4 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:

$$
\begin{aligned}
\mathit{Interface}(\mathit{Table}) &= [(\{\}, \{\mathit{Table}.\mathit{lines}, \mathit{Table}.\mathit{mh}, \mathit{Table}.\mathit{mw}\})] \\
\mathit{Interface}(\mathit{Rows}) &= [(\{\}, \{\mathit{Rows}.\mathit{mws}\}), \\
&\quad\ (\{\mathit{Rows}.\mathit{aws}\},\ \{\mathit{Rows}.\mathit{lines}, \mathit{Rows}.\mathit{mh}\})] \\
\mathit{Interface}(\mathit{Row}) &= [(\{\}, \{\mathit{Row}.\mathit{mws}\}), \\
&\quad\ (\{\mathit{Row}.\mathit{aws}\},\ \{\mathit{Row}.\mathit{lines}, \mathit{Row}.\mathit{mh}\})] \\
\mathit{Interface}(\mathit{Elems}) &= [(\{\}, \{\mathit{Elems}.\mathit{mh}, \mathit{Elems}.\mathit{mws}\}), \\
&\quad\ (\{\mathit{Elems}.\mathit{ah}, \mathit{Elems}.\mathit{aws}\}, \{\mathit{Elems}.\mathit{lines}\})] \\
\mathit{Interface}(\mathit{Elem}) &= [(\{\}, \{\mathit{Elem}.\mathit{lines}, \mathit{Elem}.\mathit{mh}, \mathit{Elem}.\mathit{mw}\})]
\end{aligned}
$$

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*).

2.3.4 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 $1 \leqslant v \leqslant \mathtt{v}(X)$, 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

$$Args_v(X) = A_{X,2*(\mathtt{v}(X)-v+1)}, \text{ and } Res_v\ (X) = A_{X,2*(\mathtt{v}(X)-v)+1}.$$

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 $1 \leqslant v \leqslant \mathtt{v}(X)$. 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 $\mathrm{arg}(Args_v(X))$ and $\mathtt{res}(Res_v(X_i))$.

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 2.5 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 2.2, 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.

2.3.5 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 (Saraiva 1999). 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

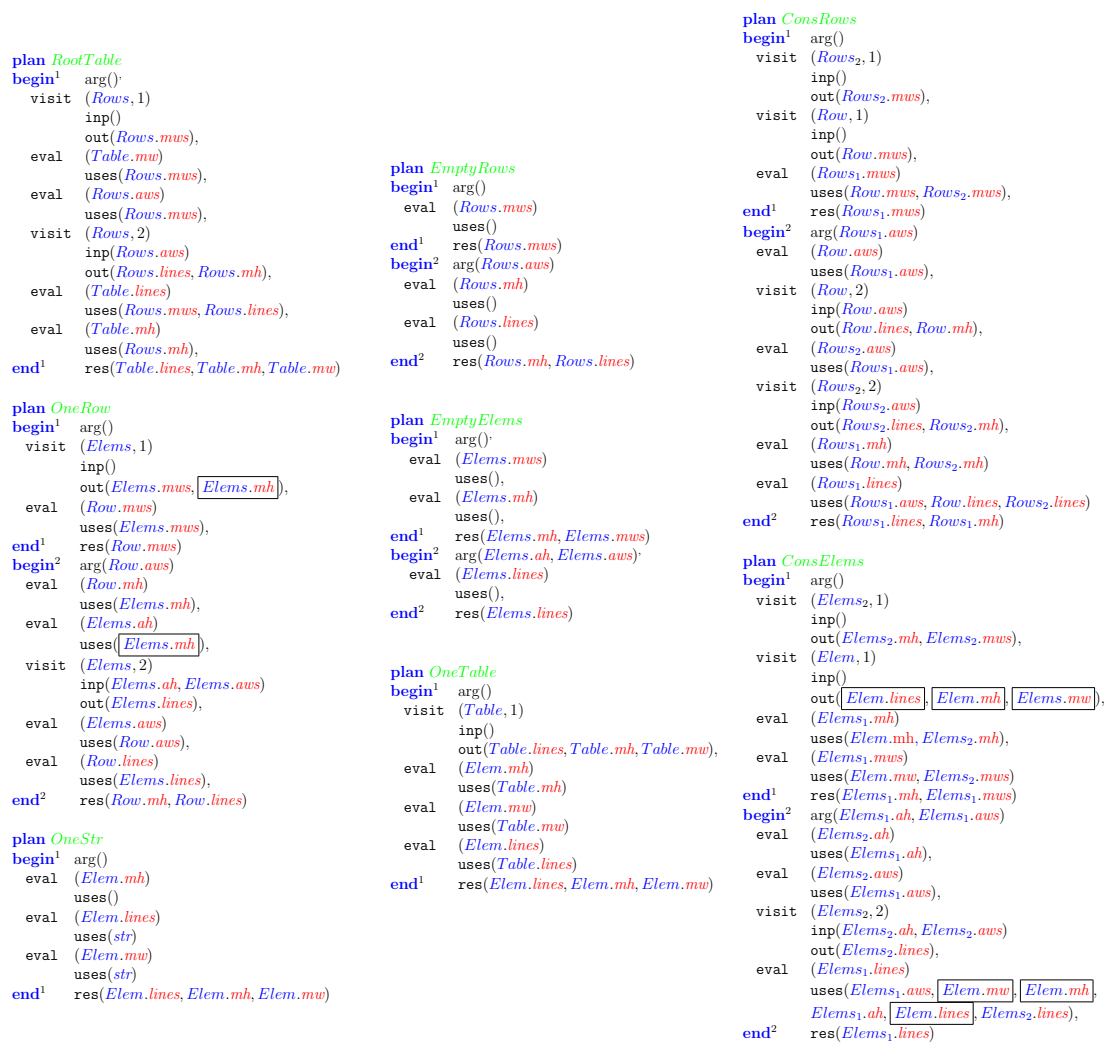

Figure 2.5: The visit-sub-sequences induced by the *Table* circular program.

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

Table 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:

$$
\begin{aligned}
\textbf{data}\ \mathit{Rows}_2 &= \mathit{ConsRows}_2\ (\mathit{Row}_2, \mathit{Rows}_2) \\
&\mid\ \mathit{EmptyRows}_2 \\
\textbf{data}\ \mathit{Row}_2 &= \mathit{OneRow}_2\ (\mathit{Int}, \mathit{Elems}_2) \\
\textbf{data}\ \mathit{Elems}_2 &= \mathit{ConsElems}_2\ ([\mathit{String}], \mathit{Int}, \mathit{Int}, \mathit{Elems}_2) \\
&\mid\ \mathit{EmptyElems}_2
\end{aligned}
$$

Note, for example, that type of $\mathit{ConsElems}_2$ constructor includes now references to the values that have to be passed from the first to its second traversal: the formatted list of strings of the element (string or nested table), its height and width. There is no reference to the *Table* or the *Elem* types because they induce a single traversal subtree. Next, we show the strict, multiple traversal program.

The sequence of abstract computations scheduled for the constructor *RootTable*, shown in Figure 2.5, is mapped to function *visitTable*.

$$
\begin{aligned}
&\mathit{visitTable} :: \mathit{Table} \rightarrow ([\mathit{String}], \mathit{Int}, \mathit{Int}) \\
&\mathit{visitTable}\ (\mathit{RootTable}\ \mathit{rows}) = (\mathit{lines}, \mathit{mw}, \mathit{mh}) \\
&\quad \textbf{where}\ (\mathit{rows}_2, \mathit{mws}) = \mathit{visitRows}_1\ \mathit{rows} \\
&\qquad\quad (\mathit{lines}_1, \mathit{mh}_1) = \mathit{visitRows}_2\ (\mathit{rows}_2, \mathit{mws}) \\
&\qquad\quad \mathit{mw} = (\mathit{sum}\ \mathit{mws}) + (\mathit{length}\ \mathit{mws}) + 1 \\
&\qquad\quad \mathit{lines} = \mathit{sepLine}\ (\mathit{mw}, \mathit{lines}_1) \\
&\qquad\quad \mathit{mh} = \mathit{mh}_1 + 2
\end{aligned}
$$

Notice that all the function calls in *visitTable* are non-circular. Remember that this was not the case of *evalRows*' function call, inside function *evalTable*, in the program presented in Section 2.2. In this sense, the calls $\mathit{visitRows}_1$ and $\mathit{visitRows}_2$ are now both strict in their arguments. They are defined as follows, according to the sequence of abstract computations scheduled for the constructors they traverse,*i.e.*, for the constructors *ConsRows* and *EmptyRows*.

$$
\begin{aligned}
&\mathit{visitRows}_1\ (\mathit{ConsRows}\ (\mathit{row}, \mathit{rows})) = (\mathit{ConsRows}_2\ (\mathit{row}_2, \mathit{rows}_2), \mathit{mws}) \\
&\quad \textbf{where}\ (\mathit{rows}_2, \mathit{mws}_2) = \mathit{visitRows}_1\ \mathit{rows} \\
&\qquad\quad (\mathit{row}_2\ , \mathit{mws}_1) = \mathit{visitRow}_1\ \ \mathit{row} \\
&\qquad\quad \mathit{mws} = \mathit{zipwith}_{\mathit{Max}}\ (\mathit{mws}_1, \mathit{mws}_2)
\end{aligned}
$$

$$
\mathit{visitRows}_1\ \mathit{EmptyRows} = (\mathit{EmptyRows}_2, [\,])
$$

$$
\begin{aligned}
&\mathit{visitRows}_2\ (\mathit{ConsRows}_2\ (\mathit{row}, \mathit{rows}), \mathit{aws}) = (\mathit{lines}, \mathit{mh}) \\
&\quad \textbf{where}\ (\mathit{lines}_1, \mathit{mh}_1) = \mathit{visitRow}_2\ (\mathit{row}, \mathit{aws}) \\
&\qquad\quad (\mathit{lines}_2, \mathit{mh}_2) = \mathit{visitRows}_2\ (\mathit{rows}, \mathit{aws}) \\
&\qquad\quad \mathit{lines} = \mathit{addSep}\ (\mathit{aws}, \mathit{lines}_1, \mathit{lines}_2) \\
&\qquad\quad \mathit{mh} = \mathit{mh}_1 + \mathit{mh}_2 + 1
\end{aligned}
$$

$$
\mathit{visitRows}_2\ (\mathit{EmptyRows}_2, \mathit{aws}) = ([\,], -1)
$$

As for constructor *OneRow*, recall Figure 2.5 to notice the two *visit-sub-sequences* scheduled over it. The first one is mapped to function $\mathit{visitRow}_1$ and the second one to function $\mathit{visitRow}_2$.

$$
\begin{aligned}
&\mathit{visitRow}_1\ (\mathit{OneRow}\ \mathit{elems}) = (\mathit{OneRow}_2\ (\mathit{mh}_1, \mathit{elems}_2), \mathit{mws}_1) \\
&\quad \textbf{where}\ (\mathit{elems}_2, \mathit{mws}_1, \mathit{mh}_1) = \mathit{visitElems}_1\ \mathit{elems}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{visitRow}_2\ (\mathit{OneRow}_2\ (\mathit{mh}_1, \mathit{elems}), \mathit{aws}) = (\mathit{lines}, \mathit{mh}_1) \\
&\quad \textbf{where}\ \mathit{lines}_1 = \mathit{visitElems}_2\ (\mathit{elems}, \mathit{mh}_1, \mathit{aws}) \\
&\qquad\quad \mathit{lines} = \mathit{addBorder}\ \mathit{lines}_1
\end{aligned}
$$

Constructors *ConsElems* and *EmptyElems* also have been scheduled two *visit-sub-sequences*, that we translate to the strict functions $\mathit{visitElems}_1$ and $\mathit{visitElems}_2$. Notice that this scheduling breaks up *evalElems*' circular invocation, inside function *evalRow*, into a two traversal strategy.

$$
\begin{aligned}
&\mathit{visitElems}_1\ (\mathit{ConsElems}\ (\mathit{elem}, \mathit{elems})) \\
&\quad = (\mathit{ConsElems}_2\ (\mathit{mh}_1, \mathit{mw}_1, \mathit{lines}_1, \mathit{elems}_2), \mathit{mh}, \mathit{mws}) \\
&\quad \textbf{where}\ (\mathit{lines}_1, \mathit{mh}_1, \mathit{mw}_1) \quad = \mathit{visitElem}\ \mathit{elem} \\
&\qquad\quad (\mathit{elems}_2, \mathit{mh}_2, \mathit{mws}_2) = \mathit{visitElems}_1\ \mathit{elems} \\
&\qquad\quad \mathit{mh} = \mathit{max}\ \mathit{mh}_1\ \mathit{mh}_2 \\
&\qquad\quad \mathit{mws} = \mathit{mw}_1 : \mathit{mws}_2
\end{aligned}
$$

$$visitElems_1\ EmptyElems = (EmptyElems_2, 0, [\,])$$

$$visitElems_2\ (ConsElems_2\ (lines_1, mh_1, mw_1, elems), ah, aws) = lines$$
$$\textbf{where}\ aws_2 = tail\ aws$$
$$lines_2 = visitElems_2\ (elems, ah, aws_2)$$
$$lines = glue\ (aws, mw_1, ah, mh_1, lines_1, lines_2)$$

$$visitElems_2\ (EmptyElems_2, ah, aws) = [\,]$$

A single traversal to constructors *OneStr* and *OneTable* is, as we have seen and as computed by the scheduling algorithm, enough to compute a single element's formatted list of strings, height and width.

$$visitElem\ (OneStr\ str) = ([str], 1, length\ str)$$

$$visitElem\ (OneTable\ table) = (lines_1, mh_1, mw_1)$$
$$\textbf{where}\ (lines_1, mh_1, mw_1) = visitTable\ table$$

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 redundant structures can, however, be eliminated by using partial evaluation techniques (Jones et al. 1993). 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 (Saraiva and Swierstra 1999). 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 construct 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 the strict, deforested *Table* program obtained by partial

evaluation of the strict one. Function *visitTable* is transformed into the following higher-order function:

$$
\begin{array}{l}
rootTable :: ([Int] \rightarrow ([String], Int), [Int]) \rightarrow ([String], Int, Int) \\
rootTable\ rows = (lines, mw, mh) \\
\quad \textbf{where}\ (rows_2, mws) = rows \\
\qquad\qquad (lines_1, mh_1) = rows_2\ mws \\
\qquad\qquad mw = (sum\ mws) + (length\ mws) + 1 \\
\qquad\qquad lines = sepLine\ (mw, lines_1) \\
\qquad\qquad mh = mh_1 + 2
\end{array}
$$

Notice that the calls $visitRows_1\ rows$ and $visitRows_2\ (rows_2, mws)$ in the strict program have been replaced, respectively, by the calls *rows* and $rows_2\ mws$ in the above definition. This means that both *rows* and $rows_2$ are now functions, instead of concrete values, as before (actually, *rows* is a special function, since it has no arguments. However, it returns a pair, whose first component, $rows_2$, is itself a function). This also means that the intermediate structure computed in the strict program, represented by variable $rows_2$, is no longer constructed: it has been deforested by partial evaluation.

Next, functions $consRows_1$, $emptyRows_1$, $consRows_2$ and $emptyRows_2$ are presented. Functions $consRows_1$ and $emptyRows_1$ specialize the definition of function $visitRows_1$ over the constructors *ConsRows* and *EmptyRows*, respectively, while functions $consRows_2$ and $emptyRows_2$ specialize the definition of $visitRows_2$ over constructors $ConsRows_2$ and $EmptyRows_2$.

$$
\begin{array}{l}
consRows_1\ (row, rows) = (consRows_2\ (row_2, rows_2), mws) \\
\quad \textbf{where}\ (rows_2, mws_2) = rows \\
\qquad\qquad (row_2, mws_1)\ = row \\
\qquad\qquad mws = zipwith\_max\ (mws_1, mws_2)
\end{array}
$$

$$
emptyRows_1 = (emptyRows_2, [\,])
$$

$$
\begin{array}{l}
consRows_2\ (row, rows, aws) = (lines, mh) \\
\quad \textbf{where}\ (lines_1, mh_1) = row\ aws
\end{array}
$$

$$
\begin{aligned}
&(lines_2, mh_2) = rows\ aws \\
&lines = addSep\ (aws, lines_1, lines_2) \\
&mh = mh_1 + mh_2 + 1
\end{aligned}
$$

$$emptyRows_2\ aws = ([\,], -1)$$

Next we present functions $oneRow_1$ and $oneRow_2$, obtained, by partial evaluation, from the definitions of $visitRow_1$ and $visitRow_2$, respectively.

$$
\begin{aligned}
&oneRow_1\ elems = (oneRow_2\ (mh_1, elems_2), mws_1) \\
&\quad \textbf{where}\ (elems_2, mws_1, mh_1) = elems
\end{aligned}
$$

$$
\begin{aligned}
&oneRow_2\ (mh_1, elems, aws) = (lines, mh_1) \\
&\quad \textbf{where}\ lines_1 = elems\ (mh_1, aws) \\
&\qquad\qquad lines\ \ = addBorder\ lines_1
\end{aligned}
$$

Functions $visitElems_1$ and $visitElems_2$ of the strict *Table* program are mapped into the following definitions.

$$
\begin{aligned}
&consElems_1\ (elem, elems) \\
&\quad = (consElems_2\ (lines_1, mh_1, mw_1, elems_2), mh, mws) \\
&\quad \textbf{where}\ (lines_1\ , mh_1, mw_1)\ \ = elem \\
&\qquad\qquad (elems_2, mh_2, mws_2) = elems \\
&\qquad\qquad mh = max\ mh_1\ mh_2 \\
&\qquad\qquad mws = mw_1 : mws_2
\end{aligned}
$$

$$emptyElems_1 = (emptyElems_2, 0, [\,])$$

$$
\begin{aligned}
&consElems_2\ ((lines_1, mh_1, mw_1, elems_2), ah, aws) = lines \\
&\quad \textbf{where}\ aws_2 = tail\ aws \\
&\qquad\qquad lines_2 = elems_2\ (ah, aws_2) \\
&\qquad\qquad lines = glue\ (aws, mw_1, ah, mh_1, lines_1, lines_2)
\end{aligned}
$$

$$emptyElems_2\ (ah, aws) = [\,]$$

Functions *oneStr* and *oneTable* consist in a simple specialization of function *visitElem*, for constructors *OneStr* and *OneTable*, respectively.

$$oneStr\ str = ([str], 1, length\ str)$$

$$
\begin{aligned}
&\mathit{oneTable}\ \mathit{table} = (\mathit{lines}_1, \mathit{mh}_1, \mathit{mw}_1) \\
&\quad \textbf{where}\ (\mathit{lines}_1, \mathit{mh}_1, \mathit{mw}_1) = \mathit{table}
\end{aligned}
$$

Although we have used a first-order circular program as the running example, the techniques introduced by the higher-order extension to attribute grammars (Swierstra and Vogt 1991) 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 (de Moor et al. 2000). 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 (Onoue et al. 1997b) to refactor the derived strict program (which uses explicit recursion) into an hylomorphism. That is 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.

2.4 Slicing Circular Programs

Although the programming language community has done a considerable amount of work on program slicing (Horwits and Reps 1992; Tip 1994), 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 (Horwits and Reps 1992).

The transformations presented in the previous sections break up the circularities that occur in a circular program. They produce a sequence of abstract computations, very suitable for further analysis and manipulation. Indeed, in the abstract computation setting, it is easy to compute forward, backward or chopping *slices* of the total program; only then the instructions selected

are mapped into a *Haskell* program. We will illustrate how we manipulate abstract computations in order to achieve slicing of circular programs.

Suppose that, from the *Table* program, we are interested in computing the table's width only. This is equivalent to saying that we want to perform bacward slicing of the *Table* program, using as criteria the variable *mw*. 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*.

We start by considering the top level constructor of that program, *RootTable*. From the total visit sequence plan scheduled for such constructor (presented in Figure 2.5), we select the following instructions:

```
plan RootTable
begin1   arg(),
  visit  (Rows, 1)
         inp()
         out(Rows.mws),
  eval   (Table.mw)
         uses(Rows.mws),
end1     res(Table.mw)
```

The `eval` instruction is filtered in since we are precisely interested in computing the result *mw*. However, that instruction states that, in order to compute *Table.mw* (the result *mw*), we must use *Rows.mws*; this value then has to be computed. The `visit` instruction is selected, since it produces exactly that value. For this constructor, slicing stops here: the `visit` instruction filtered in needs no extra arguments in order to compute *Rows.mws*.

Slicing proceeds by visiting data-type *Rows* in order to produce *mws*, as scheduled by the previous `visit` instruction. Constructors *ConsRows* and *EmptyRows* are then considered and the following instructions are selected, using the strategy described before.

```
plan ConsRows
begin[1]  arg()
  visit   (Rows_2, 1)
          inp()
          out(Rows_2.mws),
  visit   (Row, 1)
          inp()
          out(Row.mws),
  eval    (Rows_1.mws)
          uses(Row.mws, Rows_2.mws),
end[1]    res(Rows_1.mws)
```

```
plan EmptyRows
begin[1]  arg()
  eval    (Rows.mws)
          uses()
end[1]    res(Rows.mws)
```

Now, the instruction visit(*Row*, 1) tells us to traverse data-type *Row*, in order to produce the result *mws*. We obtain the following visit sequence plan for constructor *OneRow*.

```
plan OneRow
begin[1]  arg()
  visit   (Elems, 1)
          inp()
          out(Elems.mws),
  eval    (Row.mws)
          uses(Elems.mws),
end[1]    res(Row.mws)
```

Notice that, in the original program, the instruction visit(*Elems*, 1) also produced the result *Elems.mh*. However, such result has been sliced out, since we are no longer interested in producing it.

The instruction visit(*Elems*, 1) induces visits to constructors *ConsElems* and *EmptyElems*.

```
plan ConsElems
begin¹  arg()
  visit  (Elems₂, 1)
         inp()
         out(Elems₂.mws),
  visit  (Elem, 1)
         inp()
         out(Elems.mw),
  eval   (Elems₁.mws)
         uses(Elem.mw, Elems₂.mws)
end¹     res(Elems₁.mws)
```

```
plan EmptyElems
begin¹  arg()
  eval   (Elems.mws)
         uses(),
end¹     res(Elems.mws)
```

Finally, in order to compute $Elem.mw$, the instruction $\texttt{visit}(Elem, 1)$ induces the following sequence of abstract computations, for constructors $OneStr$ and $OneTable$.

```
plan OneStr
begin¹  arg()
  eval   (Elem.mw)
         uses(str)
end¹     res(Elem.mw)
```

```
plan OneTable
begin¹  arg()
  visit  (Table, 1)
         inp()
         out(Table.mw),
  eval   (Elem.mw)
         uses(Table.mw)
end¹     res(Elem.mw)
```

Next, we present the result of a backward slicing of the circular table formatter. This program is obtained by directly mapping, for every constructor of the program, the sequence of abstract computations presented into *Haskell* valid definitions.

$$
\begin{aligned}
&visitTable :: Table \rightarrow Int \\
&visitTable\ (RootTable\ rows) = mw \\
&\quad \textbf{where}\ mws_1 = visitRows\ rows \\
&\qquad\quad mw \quad = (sum\ mws_1) + (length\ mws_1) + 1
\end{aligned}
$$

$$
\begin{aligned}
&visitRows\ (ConsRows\ (row, rows)) = mws \\
&\quad \textbf{where}\ mws_2 = visitRows\ rows \\
&\qquad\quad mws_1 = visitRow\ \ row \\
&\qquad\quad mws\ \ = zipwith\_max\ mws_1\ mws_2 \\
&visitRows\ EmptyRows = [\,] \\
&visitRow\ (OneRow\ els) = mws \\
&\quad \textbf{where}\ mws = visitElems_1\ els \\
&visitElems\ (ConsElems\ (el, els)) = mw_1 : mws_2 \\
&\quad \textbf{where}\ mws_2 = visitElems\ els \\
&\qquad\quad mw_1 = visitElem\ el \\
&visitElems\ EmptyElems = [\,] \\
&visitElem\ (OneStr\ str) = length\ str \\
&visitElem\ (OneTable\ table) = mw_1 \\
&\quad \textbf{where}\ mw_1 = visitTable\ table
\end{aligned}
$$

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.

2.5 Class of Programs Considered

In the previous sections 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, Swierstra et al. (1999) 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 three 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 (Swierstra et al. 1999), however, the authors have expressed the pretty printing as an attribute grammar and 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 programs that can be evaluated without inducing non-termination. It is well-known that AG scheduling algorithm performs 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 (Okasaki 2000).

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[2].

[2]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.

2.6 Conclusion

This Chapter 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. The presented slicing techniques allow the programmer to extract different aspects of a circular program.

Chapter 3

Calculation of Circular Programs

Circular programs are a powerful technique to express multiple traversal algorithms as a single traversal function in a lazy setting. In this Chapter, we present a shortcut deforestation technique to calculate circular programs. The technique we propose takes as input the composition of two functions, such that the first builds an intermediate structure and some additional context information which are then processed by the second one, to produce the final result. Our transformation into circular programs achieves intermediate structure deforestation and multiple traversal elimination. Furthermore, the calculated programs preserve the termination properties of the original ones.

3.1 Introduction

Circular programs, as reviewed in detail in Chapter 2, provide a very appropriate formalism to model multiple traversal algorithms as elegant and concise single traversal solutions. Using this style of programming, the programmer does not have to concern him/herself with the definition and the schedulling of the different traversals and does not have to define intermediate gluing data structures. Later, circular programs can be transformed into efficient implementations using the Attribute Grammar based techniques

presented before.

However, circular programs are also known to be difficult to write and to understand. Besides, even for advanced functional programmers, it is easy to define a real circular program, that is, a program that does not terminate. Bird proposes to derive such programs from their correct and natural strict solution. Bird's approach is an elegant application of the fold-unfold transformation method coupled with tupling and circular programming. His approach, however, has a severe drawback since it preserves partial correctness only. The derived circular programs are not guaranteed to terminate. Furthermore, as an optimization technique, Bird's method focuses on eliminating multiple traversals over the same input data structure. Nevertheless, one often encounters, instead of programs that traverse the same data structure twice, programs that consist in the composition of two functions, the first of which traverses the input data and produces an intermediate structure, which is traversed by the second function, which produces the final results.

Several attempts have successfully been made to combine such compositions of two functions into a single function, eliminating the use of the intermediate structures (Wadler 1990; Onoue et al. 1997a; Gill et al. 1993; Ohori and Sasano 2007). In those situations, circular programs have also been advocated suitable for deforesting intermediate structures in compositions of two functions with accumulating parameters (Voigtländer 2004).

On the other hand, when the second traversal requires additional information, besides the intermediate structure computed in the first traversal, in order to be able to produce its outcome, no such method produces satisfactory results. In fact, as a side-effect of deforestation, they introduce multiple traversals of the input structure. This is due to the fact that deforestation methods focus on eliminating the intermediate structure, without taking into account the computation of the additional information necessary for the second traversal.

Our motivation for the work presented in this Chapter is then to transform programs of this kind into programs that construct no intermediate data-structure and that traverse the input structure only once. That is to

say, we want to perform deforestation on those programs and, subsequently, to eliminate the multiple traversals that deforestation introduces. These goals are achieved by transforming the original programs into circular ones. We allow the first traversal to produce a completely general intermediate structure together with some additional context information. The second traversal then uses such context information so that, consuming the intermediate structure produced in the first traversal, it is able to compute the desired results.

The method we propose is based on a variant of the well-known *fold/build* rule (Gill et al. 1993; Launchbury and Sheard 1995). The standard *fold/build* rule does not apply to the kind of programs we wish to calculate as they need to convey context information computed in one traversal into the following one. The new rule we introduce, called *pfold/buildp*, was designed to support contextual information to be passed between the first and the second traversals and also the use of completely general intermediate structures. Like *fold/build*, our rule is also cheap and practical to implement in a compiler.

The *pfold/buildp* rule states that the composition of such two traversals naturally induces a circular program. That is, we calculate circular programs from programs that consist of function compositions of the form $f \circ g$, where g, the producer, builds an intermediate structure t and some additional information i, and where f, the consumer, defined by structural recursion over t, traverses t and, using i, produces the desired results. The circular programs we derive compute the same results as the two original functions composed together, but they do this by performing a single traversal over the input structure. Furthermore, and since that a single traversal is performed, the intermediate structures lose their purpose. In fact, they are deforested by our rule.

In this Chapter, we not only introduce a new calculation rule, but we also present the formal proof that such rule is correct. We also present formal evidence that this rule introduces no *real* circularity, i.e., that the circular programs it derives preserve the same termination properties as the original programs. Recall that Bird's approach to circular program derivation preserves partial correctness only: the circular programs it derives are not

guaranteed to terminate, even that the original programs do.

The relevance of the rule we introduce in this Chapter may also be appreciated when observed in combination with other program transformation techniques. With our rule, we derive circular programs which most programmers would find difficult to write directly. Those programs can then be further transformed by applying manipulation techniques like, for example, the one presented in Chapter 2. This technique attempts to eliminate the performance overhead potentially introduced by circular definitions (the evaluation of such definitions requires the execution of a complex *lazy* engine) by transforming circular programs into programs that do not make essential use of lazyness. Furthermore, the obtained strict multiple traversal programs are later transformed into completely data-structure free programs. They do not traverse, nor construct, any data structure.

The work presented in this Chapter also made possible to achieve further optimizations, in calculational form (Voigtländer 2008). This particular optimization proposed a new rule that trades the circular definitions in our rule for higher-order ones. This has relevant connections with our work: the rule we present proposes circular programs as a solution to eliminate intermediate structures and multiple traversals in certain types of function compositions; Voigtländer (2008)'s rule then replaces the circular definitions obtained by higher-ordeness. This direction precisely follows the direction we took in Chapter 2: circular programs were transformed into higher-order ones, increasing its performance in a relevant factor.

This Chapter is organized as follows. In Section 3.2, we review Bird's method for deriving circular programs in the case of the *repmin* problem, and we contrast it with the (informal) derivation of the circular solution for the same problem following the method we propose. Like *fold/build*, our technique will be characterized by certain program schemes, which will be presented in Section 3.3 together with the algebraic laws necessary for the proof of the new rule. In Section 3.4 we formulate and prove the *pfold/buildp* rule; we also review the calculation of the circular program for the *repmin* problem, now in terms of the rule. Section 3.5 illustrates the application of our method to a realistic programming problem: the Algol 68 scope rules.

Section 3.6 concludes the Chapter.

3.2 Circular Programs

Circular programs were proposed by Bird (1984) as an elegant and efficient technique to eliminate multiple traversals of data structures. As the name 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.

Recall Bird's *repmin* problem of transforming a binary leaf tree into a second tree, identical in shape to the original one, but with all the leaf values replaced by the minimum leaf value. In a strict and purely functional setting, solving this problem would require a two traversal strategy: the first traversal to compute the original tree's minimum value, and the second traversal to replace all the leaf values by the minimum value, therefore producing the desired tree. This straightforward solution is as follows.

$$
\begin{aligned}
&\textbf{data}\ \mathit{LeafTree} = \mathit{Leaf}\ \ \mathit{Int} \\
&\qquad\qquad\qquad\quad \mid\ \mathit{Fork}\ (\mathit{LeafTree}, \mathit{LeafTree}) \\
&\\
&\mathit{transform} :: \mathit{LeafTree} \rightarrow \mathit{LeafTree} \\
&\mathit{transform}\ t = \mathit{replace}\ (t, \mathit{tmin}\ t) \\
&\\
&\mathit{tmin} :: \mathit{LeafTree} \rightarrow \mathit{Int} \\
&\mathit{tmin}\ (\mathit{Leaf}\ n) \qquad = n \\
&\mathit{tmin}\ (\mathit{Fork}\ (l, r)) = \mathit{min}\ (\mathit{tmin}\ l)\ (\mathit{tmin}\ r) \\
&\\
&\mathit{replace} :: (\mathit{LeafTree}, \mathit{Int}) \rightarrow \mathit{LeafTree} \\
&\mathit{replace}\ (\mathit{Leaf}\ \_, m) \qquad = \mathit{Leaf}\ m \\
&\mathit{replace}\ (\mathit{Fork}\ (l, r), m) = \mathit{Fork}\ (\mathit{replace}\ (l, m), \mathit{replace}\ (r, m))
\end{aligned}
$$

However, a two traversal strategy is not essential to solve the *repmin* problem. An alternative solution can, on a single traversal, compute the minimum value and, at the same time, replace all leaf values by that minimum value.

3.2.1 Bird's method

Bird (1984) proposed a method for deriving single traversal programs from straightforward solutions, using tupling, folding-unfolding and circular programming. For example, using Bird's method, the derivation of a single traversal solution for *repmin* proceeds as follows.

Since functions *replace* and *tmin* traverse the same data structure (a leaf tree) and given their common recursive pattern, we tuple them into one function *repmin*, which computes the same results as the previous two functions combined. Note that, in order to be able to apply such tupling step, it is essential that the two functions traverse the same data structure.

$$repmin\ (t, m) = (replace\ (t, m), tmin\ t)$$

We may now synthesize a recursive definition for *repmin* using the standard application of the fold-unfold method. Two cases have to be considered:

$$\begin{aligned}
&repmin\ (Leaf\ \ n, m)\\
&\quad = (replace\ (Leaf\ \ n, m), tmin\ (Leaf\ \ n))\\
&\quad = (Leaf\ \ m, n)
\end{aligned}$$

$$\begin{aligned}
&repmin\ (Fork\ (l, r), m)\\
&\quad = (replace\ (Fork\ (l, r), m), tmin\ (Fork\ (l, r)))\\
&\quad = (Fork\ (replace\ (l, m), replace\ (r, m)), min\ (tmin\ l)\ (tmin\ r))\\
&\quad = (Fork\ (l', r'), min\ n_1\ n_2)\\
&\quad\quad \textbf{where}\ (l',\ n_1) = repmin\ (l, m)\\
&\quad\quad\qquad\ (r', n_2) = repmin\ (r, m)
\end{aligned}$$

Finally, circular programming is used to couple the two components of the result value of *repmin* to each other. Consequently, we obtain the following circular definition of *transform*.

$$\begin{aligned}
&transform :: LeafTree \rightarrow LeafTree\\
&transform\ t = nt\\
&\quad \textbf{where}\ (nt, m) = repmin\ (t, m)
\end{aligned}$$

A single traversal is obtained because the function applied to the argument t of *transform*, the *repmin* function, traverses t only once; this single

traversal solution is possible due to the circular call of *repmin*: m is both an argument and a result of that call. This circularity ensures that the information on the minimum value is being used at the same time it is being computed.

Although the circular definitions seem to induce both cycles and non-termination of those programs, the fact is that using a *lazy* language, the *lazy* evaluation machinery is able to determine, at runtime, the right order to evaluate such circular definitions.

After the seminal paper by Bird, the style of circular programming became widely known. However, the approach followed by Bird does not guarantee termination of the resulting lazy program. In fact, Bird (1984) discusses this problem and presents an example of a non-terminating circular program obtained using his transformational technique.

3.2.2 Our method

The calculational method that we propose in this paper is, in particular, suitable for calculating a circular program that solves the *repmin* problem. In this section, we calculate such a program.

Our calculational method is used to calculate circular programs from programs that consist in the composition $f \circ g$ of a producer g and a consumer f, where $g :: a \to (b, z)$ and $f :: (b, z) \to c$.

In order to be able to apply our method to *repmin*, we then need to slightly change the straightforward solution presented earlier. In that solution, the consumer (function *replace*) fits the desired structure; however, no explicit producer occurs, since the input tree is copied as an argument to function *replace*.

We then define the following solution to *repmin*:

$$
\begin{aligned}
&\mathit{transform} :: \mathit{LeafTree} \to \mathit{LeafTree} \\
&\mathit{transform}\ t = \mathit{replace} \circ \mathit{tmint}\ \$\ t \\
\\
&\mathit{tmint} :: \mathit{LeafTree} \to (\mathit{LeafTree}, \mathit{Int}) \\
&\mathit{tmint}\ (\mathit{Leaf}\ n) \quad\ = (\mathit{Leaf}\ n, n)
\end{aligned}
$$

$$
\begin{aligned}
&tmint\ (Fork\ (l, r)) = (Fork\ (l', r'), min\ n_1\ n_2) \\
&\qquad \textbf{where}\ (l',\ n_1) = tmint\ l \\
&\qquad\qquad\ \ (r', n_2) = tmint\ r
\end{aligned}
$$

$$
\begin{aligned}
&replace :: (LeafTree, Int) \rightarrow LeafTree \\
&replace\ (Leaf\ \_, m) \quad\ = Leaf\ m \\
&replace\ (Fork\ (l, r), m) = Fork\ (replace\ (l, m), replace\ (r, m))
\end{aligned}
$$

A leaf tree (that is equal to the input one) is now the intermediate data structure that acts with the purpose of gluing the two functions.

Although the original solution needs to be slightly modified, so that it is possible to apply our method to *repmin*, we present such a modified version, and the circular program we calculate from it, since *repmin* is very intuitive, and, by far, the most well-known motivational example for circular programming. In the remaining of this Chapter we will present a realistic example (in Section 3.5) which shows that, in general, the gluing trees need to grow from traversal to traversal. This fact forces the definition of new data-structures in order to glue the different traversals together. Therefore, our rule directly applies to such examples.

Now we want to obtain a new version of *transform* that avoids the generation of the intermediate tree produced in the composition of *replace* and *tmint*. The method we propose proceeds in two steps.

First we observe that we can rewrite the original definition of *transform* as follows:

$$
\begin{aligned}
transform\ t &= replace\ (tmint\ t) \\
&= replace\ (\pi_1\ (tmint\ t), \pi_2\ (tmint\ t)) \\
&= replace' \circ \pi_1 \circ tmint\ \$\ t \\
&\quad\ \textbf{where}\ replace'\ x = replace\ (x, m) \\
&\qquad\qquad\ \ m = \pi_2\ (tmint\ t) \\
&= \pi_1 \circ (replace'\ \times\ id) \circ tmint\ \$\ t \\
&\quad\ \textbf{where}\ replace'\ x = replace\ (x, m) \\
&\qquad\qquad\ \ m = \pi_2\ (tmint\ t)
\end{aligned}
$$

where π_1 and π_2 are the projection functions and $(f\ \times\ g)\ (x, y) = (f\ x, g\ y)$.

Therefore, we can redefine *transform* as:

$$
\begin{aligned}
&\mathit{transform}\ t = nt \\
&\quad \textbf{where}\ (nt, \_) = \mathit{repmin}\ t \\
&\qquad\quad \mathit{repmin}\ t = (\mathit{replace}'\ \times\ \mathit{id}) \circ \mathit{tmint}\ \$\ t \\
&\qquad\quad \mathit{replace}'\ x = \mathit{replace}\ (x, m) \\
&\qquad\quad m = \pi_2\ (\mathit{tmint}\ t)
\end{aligned}
$$

We can now synthesize a recursive definition for *repmin* using, for example, the fold-unfold method, obtaining:

$$
\begin{aligned}
&\mathit{transform}\ t = nt \\
&\quad \textbf{where}\ (nt, \_) = \mathit{repmin}\ t \\
&\qquad\quad m = \pi_2\ (\mathit{tmint}\ t) \\
&\qquad\quad \mathit{repmin}\ (\mathit{Leaf}\ n) = (\mathit{Leaf}\ m, n) \\
&\qquad\quad \mathit{repmin}\ (\mathit{Fork}\ (l, r)) = \textbf{let}\ (l', n_1)\ = \mathit{repmin}\ l \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad (r', n_2) = \mathit{repmin}\ r \\
&\qquad\qquad\qquad\qquad\qquad\quad \textbf{in}\ (\mathit{Fork}\ (l', r'), \mathit{min}\ n_1\ n_2)
\end{aligned}
$$

In our method this synthesis will be obtained by the application of a particular short-cut fusion law. The resulting program avoids the generation of the intermediate tree, but maintains the residual computation of the minimum of the input tree, as that value is strictly necessary for computing the final tree. Therefore, this step eliminated the intermediate tree but introduced multiple traversals over t.

The second step of our method is then the elimination of the multiple traversals. Similar to Bird, we will try to obtain a single traversal function by introducing a circular definition. In order to do so, we first observe that the computation of the minimum is the same in *tmint* and *repmin*, in other words,

$$\pi_2 \circ \mathit{tmint} = \pi_2 \circ \mathit{repmin} \tag{3.1}$$

This may seem a particular observation for this specific case but it is a property that holds in general for all transformed programs of this kind. In fact, later on we will see that *tmint* and *repmin* are both instances of a same polymorphic function and actually this equality is a consequence of a

free theorem (Wadler 1989) about that function. Using this equality we may substitute *tmint* by *repmin* in the new version of *transform*, finally obtaining:

$$
\begin{aligned}
&\mathit{transform}\ t = nt \\
&\quad \textbf{where}\ (nt, m) = \mathit{repmin}\ t \\
&\qquad\quad \mathit{repmin}\ (\mathit{Leaf}\ n) = (\mathit{Leaf}\ m, n) \\
&\qquad\quad \mathit{repmin}\ (\mathit{Fork}\ (l, r)) = \textbf{let}\ (l', n_1) = \mathit{repmin}\ l \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad (r', n_2) = \mathit{repmin}\ r \\
&\qquad\qquad\qquad\qquad\qquad\quad \textbf{in}\ (\mathit{Fork}\ (l', r'), \mathit{min}\ n_1\ n_2)
\end{aligned}
$$

This new definition not only unifies the computation of the final tree and the minimum in *repmin*, but it also introduces a circularity on m. The introduction of the circularity is a direct consequence of this unification. As expected, the resulting circular program traverses the input tree only once. Furthermore, it does not construct the intermediate leaf-tree, which has been eliminated during the transformation process.

The introduction of the circularity is safe in our context. Unlike Bird, our introduction of the circularity is made in such a way that it is possible to safely schedule the computations. For instance, in our example, the essential property that makes this possible is the equality (3.1), which is a consequence of the fact that both in *tmint* and *repmin* the computation of the minimum does not depend on the computation of the corresponding tree. The fact that this property is not specific of this particular example, but it is an instance of a general one, is what makes it possible to generalize the application of our method to a wide class of programs.

In this Section, we have shown an instance of our method for obtaining a circular lazy program from an initial solution that makes no essential use of lazyness. In the next Sections we formalize our method using a calculational approach. Furthermore, we present the formal proof that guarantees its correctness.

3.3 Program schemes

Our method will be applied to a class of expressions that will be characterized in terms of program schemes. This will allow us to give a generic formulation of the transformation rule in the sense that it will be parametric in the structure of the intermediate data type involved in the function composition to be transformed.

In this section we describe two program schemes which capture structurally recursive functions and are relevant constructions in our transformation. Throughout we shall assume we are working in the context of a lazy functional language with a *cpo* semantics, in which types are interpreted as pointed cpos (complete partial orders with a least element $\perp$) and functions are interpreted as continuous functions between pointed cpos. However, our semantics differs from that of Haskell in that we do not consider lifted cpos. That is, unlike the semantics of Haskell, we do not consider lifted products and function spaces. As usual, a function f is said to be *strict* if it preserves the least element, i.e. $f\ \perp = \perp$.

3.3.1 Data types

The structure of datatypes can be captured using the concept of a *functor*. A functor consists of two components, both denoted by F: a type constructor F, and a function $F :: (a \to b) \to (F\ a \to F\ b)$, which preserves identities and compositions:

$$F\ id = id \qquad\qquad F\ (f \circ g) = F\ f \circ F\ g$$

A standard example of a functor is that formed by the list type constructor and the well-known *map* function, which applies a function to the elements of a list, building a new list with the results.

$$\begin{aligned} &map && :: (a \to b) \to [a] \to [b] \\ &map\ f\ [\,] && = [\,] \\ &map\ f\ (a : as) && = f\ a : map\ f\ as \end{aligned}$$

Another example of a functor is the product functor, which is a case of a bifunctor, a functor on two arguments. On types its action is given by the type constructor for pairs. On functions its action is defined by:

$$(\times) :: (a \to c) \to (b \to d) \to (a, b) \to (c, d)$$
$$(f \times g)\ (a, b) = (f\ a, g\ b)$$

Semantically, we assume that pairs are interpreted as the cartesian product of the corresponding cpos. Associated with the product we can define the following functions, corresponding to the projections and the split function:

$$\pi_1 :: (a, b) \to a$$
$$\pi_1\ (a, b) = a$$

$$\pi_2 :: (a, b) \to b$$
$$\pi_2\ (a, b) = b$$

$$(\triangle) :: (c \to a) \to (c \to b) \to c \to (a, b)$$
$$(f \triangle g)\ c = (f\ c, g\ c)$$

Among others properties, it holds that

$$f \circ \pi_1 = \pi_1 \circ (f \times g) \tag{3.2}$$

$$g \circ \pi_2 = \pi_2 \circ (f \times g) \tag{3.3}$$

$$f = ((\pi_1 \circ f) \triangle (\pi_2 \circ f)) \tag{3.4}$$

Another case of bifunctor is the sum functor, which corresponds to the disjoint union of types. Semantically, we assume that sums are interpreted as the separated sum of the corresponding cpos.

$$\textbf{data}\ a + b = \mathit{Left}\ a \mid \mathit{Right}\ b$$

$$(+) :: (a \to c) \to (b \to d) \to (a + b) \to (c + d)$$
$$(f + g)\ (\mathit{Left}\ a) = \mathit{Left}\ (f\ a)$$
$$(f + g)\ (\mathit{Right}\ b) = \mathit{Right}\ (g\ b)$$

Associated with the sum we can define the case analysis function, which has the property of being strict in its argument of type $a + b$:

$$(\triangledown) :: (a \to c) \to (b \to c) \to (a + b) \to c$$
$$(f \triangledown g)\ (\mathit{Left}\ a) = f\ a$$
$$(f \triangledown g)\ (\mathit{Right}\ b) = g\ b$$

Product and sum can be generalized to n components in the obvious way.

We consider declarations of datatypes of the form:[1]

$$\mathbf{data}\ \tau = C_1\ (\tau_{1,1}, \cdots, \tau_{1,k_1})\ \mid \cdots \mid\ C_n\ (\tau_{n,1}, \cdots, \tau_{n,k_n})$$

where each $\tau_{i,j}$ is restricted to be a constant type (like *Int* or *Char*), a type variable, a type constructor D applied to a type $\tau'_{i,j}$ or τ itself. Datatypes of this form are usually called regular. The derivation of a functor that captures the structure of the datatype essentially proceeds as follows: alternatives are regarded as sums ($\mid$ is replaced by $+$) and occurrences of τ are substituted by a type variable a in every $\tau_{i,j}$. In addition, the unit type () is placed in the positions corresponding to constant constructors (like e.g. the empty list constructor). As a result, we obtain the following type constructor F:

$$F\,a = (\sigma_{1,1}, \cdots, \sigma_{1,k_1}) + \cdots + (\sigma_{n,1}, \cdots, \sigma_{n,k_n})$$

where $\sigma_{i,j} = \tau_{i,j}[\tau := a]$[2]. The body of the corresponding mapping function $F :: (a \to b) \to (F\ a \to F\ b)$ is similar to that of $F\ a$, with the difference that the occurrences of the type variable a are replaced by a function $f :: a \to b$, and identities are placed in the other positions:

$$F f = g_{1,1} \times \cdots \times g_{1,k_1}\ + \cdots +\ g_{n,1} \times \cdots \times g_{n,k_n}$$

with

$$g_{i,j} = \begin{cases} f & \text{if } \sigma_{i,j} = a \\ id & \text{if } \sigma_{i,j} = t, \text{ for some type } t \\ D\ g'_{i,j} & \text{if } \sigma_{i,j} = D\ \sigma'_{i,j} \end{cases}$$

[1] For simplicity we shall assume that constructors in a datatype declaration are declared uncurried.

[2] By $s[t := a]$ we denote the replacement of every occurrence of t by a in s.

where the D in the expression $D\, g'_{i,j}$ represents the *map* function $D :: (a \to b) \to (D\ a \to D\ b)$ corresponding to the type constructor D.

For example, for the type of leaf trees

$$\begin{aligned}&\textbf{data}\ \mathit{LeafTree} = \mathit{Leaf}\ \mathit{Int}\\ &\qquad\qquad\qquad\quad \mid\ \mathit{Fork}\ (\mathit{LeafTree}, \mathit{LeafTree})\end{aligned}$$

we can derive a functor T given by

$$T\ a = \mathit{Int} + (a, a)$$

$$\begin{aligned}T\quad &:: (a \to b) \to (T\ a \to T\ b)\\ T\ f &= \mathit{id} + f\ \times\ f\end{aligned}$$

The functor that captures the structure of the list datatype needs to reflect the presence of the type parameter:

$$L_a\ b = () + (a, b)$$

$$\begin{aligned}L_a\quad &:: (b \to c) \to (L_a\ b \to L_a\ c)\\ L_a\ f &= \mathit{id} + \mathit{id}\ \times\ f\end{aligned}$$

This functor reflects the fact that lists have two constructors: one is a constant and the other is a binary operation.

Every recursive datatype is then understood as the least fixed point of the functor F that captures its structure, i.e. as the least solution to the equation $\tau \cong F\tau$. We will denote the type corresponding to the least solution as μF. The isomorphism between μF and $F\ \mu F$ is provided by the strict functions $\mathit{in}_F :: F\ \mu F \to \mu F$ and $\mathit{out}_F :: \mu F \to F\ \mu F$, each other inverse. Function in_F packs the constructors of the datatype while function out_F packs its destructors. Further details can be found in (Abramsky and Jung 1994; Gibbons 2002).

For instance, in the case of leaf trees we have that $\mu T = \mathit{LeafTree}$ and

$$\begin{aligned}\mathit{in}_T\ &::\ T\ \mathit{LeafTree} \to \mathit{LeafTree}\\ \mathit{in}_T\ &=\ \mathit{Leaf}\ \triangledown\ \mathit{Fork}\end{aligned}$$

$$\begin{aligned}&\mathit{out}_T :: \mathit{LeafTree} \to T\ \mathit{LeafTree}\\ &\mathit{out}_T\ (\mathit{Leaf}\ n) = \mathit{Left}\ n\end{aligned}$$

$$out_T\ (Fork\ (l, r)) = Right\ (l, r)$$

3.3.2 Fold

Fold (Bird and de Moor 1997; Gibbons 2002) is a pattern of recursion that captures function definitions by structural recursion. The best known example of fold is its definition for lists, which corresponds to the *foldr* operator (Bird 1998).

Given a functor F and a function $h :: F\ a \to a$, *fold* (or *catamorphism*), denoted by $fold\ h :: \mu F \to a$, is defined as the least function f that satisfies the following equation:

$$f \circ in_F = h \circ F\ f$$

Because out_F is the inverse of in_F, this is the same as:

$$\begin{aligned} fold\ \ &:: (F\ a \to a) \to \mu F \to a \\ fold\ h &= h \circ F\ (fold\ h) \circ out_F \end{aligned}$$

A function $h :: F\ a \to a$ is called an *F-algebra*.[3] The functor F plays the role of signature of the algebra, as it encodes the information about the operations of the algebra. The type a is called the carrier of the algebra. An *F-homomorphism* between two algebras $h :: F\ a \to a$ and $k :: F\ b \to b$ is a function $f :: a \to b$ between the carriers that commutes with the operations. This is specified by the condition $f \circ h = k \circ F\ f$. Notice that $fold\ h$ is a homomorphism between the algebras in_F and h.

For example, for leaf trees fold is given by:

$$\begin{aligned} &fold_T :: (Int \to a, (a, a) \to a) \to LeafTree \to a \\ &fold_T\ (h_1, h_2) = f_T \\ &\quad \textbf{where}\ f_T\ (Leaf\ n) = h_1\ n \\ &\qquad\qquad f_T\ (Fork\ (l, r)) = h_2\ (f_T\ l, f_T\ r) \end{aligned}$$

For instance,

$$tmin :: LeafTree \to Int$$

[3] When showing specific instances of fold for concrete datatypes, we will write the operations in an algebra $h_1 \triangledown \cdots \triangledown h_n$ in a tuple $(h_1, \ldots, h_n)$.

$$tmin\ (Leaf\ n) = n$$
$$tmin\ (Fork\ (l, r)) = min\ (tmin\ l)\ (tmin\ r)$$

can be defined as:

$$tmin = fold_T\ (id, uncurry\ min)$$

Fold enjoys many algebraic laws that are useful for program transformation. A well-known example is *shortcut fusion* (Gill et al. 1993; Gill 1996; Takano and Meijer 1995) (also known as the *fold/build* rule), which is an instance of a free theorem (Wadler 1989).

Law 3.3.1 (FOLD/BUILD RULE**)** *For h strict,*

$$\begin{array}{l} \qquad g :: \forall\ a\ .\ (F\ a \to a) \to c \to a \\ \Rightarrow \\ \qquad fold\ h \circ build\ g = g\ h \end{array}$$

where

$$build :: (\forall\ a\ .\ (F\ a \to a) \to c \to a) \to c \to \mu F$$
$$build\ g = g\ in_F$$

The instance of this law for leaf trees is the following:

$$fold_T\ (h_1, h_2) \circ build_T\ g = g\ (h_1, h_2) \tag{3.5}$$

where

$$build_T :: (\forall\ a\ .\ (Int \to a, (a, a) \to a) \to c \to a) \to c \to LeafTree$$
$$build_T\ g = g\ (Leaf, Fork)$$

The assumption about the strictness of the algebra disappears because every algebra $h_1 \triangledown h_2$ is strict as so is every case analysis.

As an example, we can use this law to fuse:

$$tmm = tmin \circ mirror$$

$$mirror :: LeafTree \to LeafTree$$
$$mirror\ (Leaf\ n) = Leaf\ n$$

$$mirror\ (Fork\ (l, r)) = Fork\ (mirror\ r, mirror\ l)$$

To do so, first we have to express *mirror* in terms of $build_T$:

$$\begin{aligned} &mirror = build_T\ g \\ &\quad \textbf{where}\ g\ (leaf, fork)\ (Leaf\ n) = leaf\ n \\ &\qquad\quad g\ (leaf, fork)\ (Fork\ (l, r)) = fork\ (g\ (leaf, fork)\ r, \\ &\qquad\qquad\qquad\qquad\qquad\qquad g\ (leaf, fork)\ l) \end{aligned}$$

Finally, by (3.5) we have that

$$tmm = g\ (id, uncurry\ min)$$

Inlining,

$$\begin{aligned} &tmm\ (Leaf\ n) = n \\ &tmm\ (Fork\ (l, r)) = min\ (tmm\ r)\ (tmm\ l) \end{aligned}$$

In the same line of reasoning, we can state another fusion law for a slightly different producer function:

Law 3.3.2 (FOLD/BUILDP RULE) *For h strict,*

$$\begin{aligned} & g :: \forall\ a\ .\ (F\ a \to a) \to c \to (a, z) \\ \Rightarrow \\ & (fold\ h\ \times\ id) \circ buildp\ g = g\ h \end{aligned}$$

where

$$\begin{aligned} &buildp :: (\forall\ a\ .\ (F\ a \to a) \to c \to (a, z)) \to c \to (\mu F, z) \\ &buildp\ g = g\ in_F \end{aligned}$$

Proof From the polymorphic type of g we can deduce the following free theorem: for f strict,

$$f \circ \phi = \psi \circ F\ f\ \Rightarrow\ (f\ \times\ id) \circ g\ \phi = g\ \psi$$

By taking $f = fold\ h$, $\phi = in_F$, $\psi = h$ we obtain that $(fold\ h\ \times\ id) \circ g\ in_F = g\ h$. The equation on the left-hand side of the implication becomes true by definition of fold. The requirement that f is strict is satisfied by the fact

that every fold with a strict algebra is strict, and by hypothesis h is strict. Finally, by definition of *buildp* the desired result follows. □

For example, the instance of this law for leaf trees is the following:

$$(fold_T\ (h_1, h_2)\ \times\ id) \circ buildp_T\ g = g\ (h_1, h_2) \tag{3.6}$$

where

$$\begin{aligned} &buildp_T :: (\forall\ a\ .\ (Int \to a, (a, a) \to a) \to c \to (a, z)) \to c \to (LeafTree, z) \\ &buildp_T\ g = g\ (Leaf, Fork) \end{aligned}$$

The assumption about the strictness of the algebra disappears by the same reason as for (3.5).

To see an example of the application of this law, consider the function *ssqm*:

$$\begin{aligned} &ssqm :: LeafTree \to (Int, Int) \\ &ssqm = (sumt\ \times\ id) \circ gentsqmin \end{aligned}$$

$$\begin{aligned} &sumt :: LeafTree \to Int \\ &sumt\ (Leaf\ n) = n \\ &sumt\ (Fork\ (l, r)) = sumt\ l + sumt\ r \end{aligned}$$

$$\begin{aligned} &gentsqmin :: LeafTree \to (LeafTree, Int) \\ &gentsqmin\ (Leaf\ n) = (Leaf\ (n * n), n) \\ &gentsqmin\ (Fork\ (l, r)) = \mathbf{let}\ (l', n_1) = gentsqmin\ l \\ &\qquad\qquad\qquad\qquad\qquad\qquad\quad (r', n_2) = gentsqmin\ r \\ &\qquad\qquad\qquad\qquad\qquad\quad \mathbf{in}\ (Fork\ (l', r'), min\ n_1\ n_2) \end{aligned}$$

To apply Law (3.6) we have to express *sumt* as a fold and *gentsqmin* in terms of $buildp_T$:

$$\begin{aligned} &sumt\ \ \ \ \ \ = fold_T\ (id, uncurry\ (+)) \\ &gentsqmin = buildp_T\ g \\ &\quad \mathbf{where}\ g\ (leaf, fork)\ (Leaf\ n) = (leaf\ (n * n), n) \\ &\qquad\quad g\ (leaf, fork)\ (Fork\ (l, r)) = \mathbf{let}\ (l', n_1) = g\ (leaf, fork)\ l \\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad (r', n_2) = g\ (leaf, fork)\ r \end{aligned}$$

$$\textbf{in}\ (fork\ (l', r'), min\ n_1\ n_2)$$

Hence, by (3.6),

$$ssqm = g\ (id, uncurry\ (+))$$

Inlining,

$$\begin{aligned} ssqm\ (Leaf\ n) &= (n * n, n) \\ ssqm\ (Fork\ (l, r)) &= \textbf{let}\ (s_1, n_1) = ssqm\ l \\ &\qquad\qquad (s_2, n_2) = ssqm\ r \\ &\qquad \textbf{in}\ (s_1 + s_2, min\ n_1\ n_2) \end{aligned}$$

Finally, the following property is an immediate consequence of Law 3.3.2.

Law 3.3.3 *For any strict h,*

$$\begin{aligned} & g :: \forall\ a\ .\ (F\ a \to a) \to c \to (a, z) \\ \Rightarrow \\ & \pi_2 \circ g\ in_F = \pi_2 \circ g\ h \end{aligned}$$

Proof

$$\begin{aligned} & \pi_2 \circ g\ in_F \\ = & \qquad \{\ (3.3)\ \} \\ & \pi_2 \circ (fold\ h\ \times\ id) \circ g\ in_F \\ = & \qquad \{\ \text{Law 3.3.2}\ \} \\ & \pi_2 \circ g\ h \qquad\qquad\qquad \square \end{aligned}$$

This property states that the construction of the second component of the pair returned by g is independent of the particular algebra that g carries; it only depends on the input value of type c. This is a consequence of the polymorphic type of g and the fact that the second component of its result is of a fixed type z.

3.3.3 Fold with parameters

Some recursive functions use context information in the form of constant parameters for their computation. The aim of this section is to analyze the definition of structurally recursive functions of the form $f :: (\mu F, z) \rightarrow a$, where the type z represents the context information. Our interest in these functions is because our method will assume that consumers are functions of this kind.

Functions of this form can be defined in different ways. One alternative consists of fixing the value of the parameter and performing recursion on the other. Definitions of this kind can be given in terms of a fold:

$$f :: (\mu F, z) \rightarrow a$$
$$f\ (t, z) = fold\ h\ t$$

such that the context information contained in z may eventually be used in the algebra h. This is the case of, for example, function:

$$replace :: (LeafTree, Int) \rightarrow LeafTree$$
$$replace\ (Leaf\ n, m) = Leaf\ m$$
$$replace\ (Fork\ (l, r), m) = Fork\ (replace\ (l, m), replace\ (r, m))$$

which can be defined as:

$$replace\ (t, m) = fold_T\ (\lambda n \rightarrow Leaf\ m, Fork)\ t$$

Another alternative is the use of currying, which gives a function of type $\mu F \rightarrow (z \rightarrow a)$. The curried version can then be defined as a higher-order fold. For instance, in the case of *replace* it holds that

$$curry\ replace = fold_T\ (Leaf, \lambda(f, f') \rightarrow Fork \circ ((f\ \triangle\ f')))$$

This is an alternative we won't pursue in this paper.

A third alternative is to define the function $f :: (\mu F, z) \rightarrow a$ in terms of a program scheme, called *pfold* (Pardo 2001, 2002), which, unlike fold, is able to manipulate constant and recursive arguments simultaneously. The definition of pfold relies on the concept of *strength* of a functor F, which is a polymorphic function:

$$\tau^F :: (F\ a, z) \rightarrow F\ (a, z)$$

that satisfies certain coherence axioms (see (Pardo 2002; Cockett and Spencer 1991; Cockett and Fukushima 1992) for details). The strength distributes the value of type z to the variable positions (positions of type a) of the functor. For instance, the strength corresponding to functor T is given by:

$$\tau^T :: (T\ a, z) \to T\ (a, z)$$
$$\tau^T\ (\mathit{Left}\ n, z) = \mathit{Left}\ n$$
$$\tau^T\ (\mathit{Right}\ (a, a'), z) = \mathit{Right}\ ((a, z), (a', z))$$

In the definition of pfold the strength of the underlying functor plays an important role as it represents the distribution of the context information contained in the constant parameters to the recursive calls.

Given a functor F and a function $h :: (F\ a, z) \to a$, *pfold*, denoted by $\mathit{pfold}\ h :: (\mu F, z) \to a$, is defined as the least function f that satisfies the following equation:

$$f \circ (in_F\ \times\ id) = h \circ (((F\ f \circ \tau^F) \vartriangle \pi_2))$$

Observe that now function h also accepts the value of the parameters. It is a function of the form $(h_1\ \triangledown\ \ldots\ \triangledown\ h_n) \circ d$ where each $h_i :: (F_i\ a, z) \to a$ if $F\ a = F_1\ a + \cdots + F_n\ a$, and $d :: (x_1 + \cdots + x_n, z) \to (x_1, z) + \cdots + (x_n, z)$ is the distribution of product over sum. When showing specific instances of pfold we will simply write the tuple of functions $(h_1, \ldots, h_n)$ instead of h.

For example, in the case of leaf trees the definition of pfold is as follows:

$$\mathit{pfold}_T :: ((\mathit{Int}, z) \to a, ((a, a), z) \to a) \to (\mathit{LeafTree}, z) \to a$$
$$\mathit{pfold}_T\ (h_1, h_2) = p_T$$
$$\textbf{where}\ p_T\ (\mathit{Leaf}\ n, z) = h_1\ (n, z)$$
$$p_T\ (\mathit{Fork}\ (l, r), z) = h_2\ ((p_T\ (l, z), p_T\ (r, z)), z)$$

We can then write *replace* in terms of a pfold:

$$\mathit{replace} = \mathit{pfold}_T\ (\mathit{Leaf} \circ \pi_2, \mathit{Fork} \circ \pi_1)$$

The following equation shows one of the possible relationships between pfold and fold.

$$\mathit{pfold}\ h\ (t, z) = \mathit{fold}\ k\ t\ \ \textbf{where}\ \ k_i\ x = h_i\ (x, z) \tag{3.7}$$

Like fold, pfold satisfies a set of algebraic laws. We don't show any of them

here as they are not necessary for this paper. The interested reader may consult (Pardo 2001, 2002).

3.4 The pfold/buildp rule

In this section we present a generic formulation and proof of correctness of the transformation rule we propose. The rule takes a composition of the form $cons \circ prod$, composed by a producer $prod :: a \to (t, z)$ followed by a consumer $cons :: (t, z) \to b$, and returns an equivalent deforested circular program that performs a single traversal over the input value. The reduction of this expression into an equivalent one without intermediate data structures is performed in two steps. Firstly, we apply standard deforestation techniques in order to eliminate the intermediate data structure of type t. The program obtained is deforested, but in general contains multiple traversals over the input as a consequence of residual computations of the other intermediate values (e.g. the computation of the minimum in the case of *repmin*). Therefore, as a second step, we perform the elimination of the multiple traversals by the introduction of a circular definition.

The rule makes some natural assumptions about *cons* and *prod*: t is a recursive data type μF, the consumer *cons* is defined by structural recursion on t, and the intermediate value of type z is taken as a constant parameter by *cons*. In addition, it is required that *prod* is a "good producer", in the sense that it is possible to express it as the instance of a polymorphic function by abstracting out the constructors of the type t from the body of *prod*. In other words, *prod* should be expressed in terms of the *buildp* function corresponding to the type t. The fact that the consumer *cons* is assumed to be structurally recursive leads us to consider that it is given by a pfold. In summary, the rule is applied to compositions of the form: $pfold\ h \circ buildp\ g$.

Law 3.4.1 (PFOLD/BUILDP RULE) *For any* $h = (h_1 \triangledown \ \ldots \ \triangledown\ h_n) \circ d$,

$$
\begin{array}{l}
\quad g :: \forall\ a\ .\ (F\ a \to a) \to c \to (a, z) \\
\Rightarrow \\
\quad \mathit{pfold}\ h \circ \mathit{buildp}\ g\ \$\ c \\
\qquad = v \\
\qquad \textbf{where}\ (v, z) = g\ k\ c \\
\qquad\qquad\quad k = k_1\ \triangledown\ \ldots\ \triangledown\ k_n \\
\qquad\qquad\quad k_i\ x = h_i\ (x, z)
\end{array}
$$

Proof The proof will show in detail the two steps of our method. The first step corresponds to the application of deforestation, which is represented by Law 3.3.2. For that reason we need first to express the pfold as a fold.

$$
\begin{array}{ll}
 & \mathit{pfold}\ h \circ \mathit{buildp}\ g\ \$\ c \\
= & \quad \{\ \text{definition of } \mathit{buildp}\ \} \\
 & \mathit{pfold}\ h \circ g\ \mathit{in}_F\ \$\ c \\
= & \quad \{\ (3.4)\ \} \\
 & \mathit{pfold}\ h \circ (((\pi_1 \circ g\ \mathit{in}_F)\ \vartriangle\ (\pi_2 \circ g\ \mathit{in}_F)))\ \$\ c \\
= & \quad \{\ (3.7)\ \} \\
 & \mathit{fold}\ k \circ \pi_1 \circ g\ \mathit{in}_F\ \$\ c \\
 & \quad \textbf{where}\ z = \pi_2 \circ g\ \mathit{in}_F\ \$\ c \\
 & \qquad\quad k_i\ x = h_i\ (x, z) \\
= & \quad \{\ (3.2)\ \} \\
 & \pi_1 \circ (\mathit{fold}\ k\ \times\ \mathit{id}) \circ g\ \mathit{in}_F\ \$\ c \\
 & \quad \textbf{where}\ z = \pi_2 \circ g\ \mathit{in}_F\ \$\ c \\
 & \qquad\quad k_i\ x = h_i\ (x, z) \\
= & \quad \{\ \text{Law 3.3.2}\ \} \\
 & \pi_1 \circ g\ k\ \$\ c \\
 & \quad \textbf{where}\ z = \pi_2 \circ g\ \mathit{in}_F\ \$\ c \\
 & \qquad\quad k_i\ x = h_i\ (x, z)
\end{array}
$$

Law 3.3.2 was applicable because by construction the algebra k is strict.

Once we have reached this point we observe that the resulting program is deforested, but it contains two traversals on c. The elimination of the multiple traversals is then performed by introducing a circular definition. The essential property that makes it possible the safe introduction of a circularity is Law 3.3.3, which states that the computation of the second component of type z is independent of the particular algebra that is passed to g. This is a consequence of the polymorphic type of g. Therefore, we can replace in_F by another algebra and we will continue producing the same value z. In particular, we can take k as this other algebra, and in that way we are introducing the circularity. It is this property that ensures that no terminating program is turned into a nonterminating one.

$$
\begin{array}{ll}
 & \pi_1 \circ g\ k\ \$\ c \\
 & \quad \textbf{where}\ z = \pi_2 \circ g\ in_F\ \$\ c \\
 & \qquad\qquad k_i\ x = h_i\ (x, z) \\
= & \quad \{\ \text{Law 3.3.3}\ \} \\
 & \pi_1 \circ g\ k\ \$\ c \\
 & \quad \textbf{where}\ z = \pi_2 \circ g\ k\ \$\ c \\
 & \qquad\qquad k_i\ x = h_i\ (x, z) \\
= & \quad \{\ (3.4)\ \} \\
 & v \\
 & \quad \textbf{where}\ (v, z) = g\ k\ c \\
 & \qquad\qquad k_i\ x = h_i\ (x, z) \qquad \square
\end{array}
$$

Now, let us see the application of the pfold/buildp rule in the case of the *repmin* problem. Recall the definition we want to transform:

$$
\begin{array}{l}
transform :: LeafTree \rightarrow LeafTree \\
transform\ t = replace \circ tmint\ \$\ t
\end{array}
$$

To apply the rule, first we have to express *replace* and *tmint* in terms of pfold and buildp for leaf trees, respectively:

$$replace = pfold_T\ (Leaf \circ \pi_2, Fork \circ \pi_1)$$

$$
\begin{aligned}
&tmint = buildp_T\ g \\
&\quad \textbf{where}\ g\ (leaf, fork)\ (Leaf\ n) = (leaf\ n, n) \\
&\qquad\quad g\ (leaf, fork)\ (Fork\ (l, r)) = \textbf{let}\ (l', n_1) = g\ (leaf, fork)\ l \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad (r', n_2) = g\ (leaf, fork)\ r \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{in}\ (fork\ (l', r'), min\ n_1\ n_2)
\end{aligned}
$$

Therefore, by applying Law 3.4.1 we get:

$$
\begin{aligned}
&transform\ t = nt \\
&\quad \textbf{where}\ (nt, m) = g\ (k_1, k_2)\ t \\
&\qquad\quad k_1\ \_ = Leaf\ m \\
&\qquad\quad k_2\ (l, r) = Fork\ (l, r)
\end{aligned}
$$

Inlining, we obtain the definition we showed previously in Section 3.2.2:

$$
\begin{aligned}
&transform\ t = nt \\
&\quad \textbf{where} \\
&\qquad (nt, m) = repmin\ t \\
&\qquad repmin\ (Leaf\ n) = (Leaf\ m, n) \\
&\qquad repmin\ (Fork\ (l, r)) = \textbf{let}\ (l', n_1) = repmin\ l \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad (r', n_2) = repmin\ r \\
&\qquad\qquad\qquad\qquad\qquad\quad \textbf{in}\ (Fork\ (l', r'), min\ n_1\ n_2)
\end{aligned}
$$

3.5 Algol 68 scope rules

In this section, we consider the application of our rule to a real example: the Algol 68 scope rules (Saraiva 1999; de Moor et al. 2000). These rules are used, for example, in the Eli system [4] (Waite et al. 2007) to define a generic component for the name analysis task of a compiler.

We wish to construct a program to deal with the scope rules of a block structured language, the Algol 68. In this language a definition of an identifier x is visible in the smallest enclosing block, with the exception of local blocks that also contain a definition of x. In the latter case, the definition of x in

[4] A well known compiler generator toolbox.

the local scope hides the definition in the global one. In a block an identifier may be declared at most once. We shall analyze these scope rules via our favorite (toy) language: the BLOCK language, which consists of programs of the following form:

$$
\begin{array}{l}
[\,\mathbf{use}\ y; \mathbf{decl}\ x; \\
\quad [\,\mathbf{decl}\ y; \mathbf{use}\ y; \mathbf{use}\ w;\,] \\
\mathbf{decl}\ x; \mathbf{decl}\ y;\,]
\end{array}
$$

We define the following *Haskell* data-type to represent BLOCK programs.

$$
\begin{array}{ll}
\mathbf{type}\ \mathit{Prog} = [\mathit{It}] & \mathbf{data}\ \mathit{It} = \mathit{Use}\ \mathit{Var} \\
 & \quad \mid \mathit{Decl}\ \mathit{Var} \\
\mathbf{type}\ \mathit{Var} = \mathit{String} & \quad \mid \mathit{Block}\ \mathit{Prog}
\end{array}
$$

Such programs describe the basic block-structure found in many languages, with the peculiarity however that declarations of identifiers may also occur after their first use (but in the same level or in an outer one). According to these rules the above program contains two errors: at the outer level, the variable x has been declared twice and the use of the variable w, at the inner level, has no binding occurrence at all.

We aim to develop a program that analyses BLOCK programs and computes a list containing the identifiers which do not obey to the rules of the language. In order to make the problem more interesting, and also to make it easier to detect which identifiers are being incorrectly used in a BLOCK program, we require that the list of invalid identifiers follows the sequential structure of the input program. Thus, the semantic meaning of processing the example sentence is $[w, x]$.

Because we allow an *use-before-declare* discipline, a conventional implementation of the required analysis naturally leads to a program which traverses the abstract syntax tree twice: once for accumulating the declarations of identifiers and constructing the environment, and once for checking the uses of identifiers, according to the computed environment. The uniqueness of names is detected in the first traversal: for each newly encountered declaration it is checked whether that identifier has already been declared at the

current level. In this case an error message is computed. Of course the identifier might have been declared at a global level. Thus we need to distinguish between identifiers declared at different levels. We use the level of a block to achieve this. The environment is a partial function mapping an identifier to its level of declaration. In *Haskell* we represent the environment as follows.

$$\textbf{type } Env = [(Var, Int)]$$

Semantic errors resulting from duplicate definitions are then computed during the first traversal of a block and errors resulting from missing declarations in the second one. In a straightforward implementation of this program, this strategy has two important effects: the first is that a *"gluing"* data structure has to be defined and constructed to pass explicitly the detected errors from the first to the second traversal, in order to compute the final list of errors in the desired order; the second is that, in order to be able to compute the missing declarations of a block, the implementation has to explicitly pass (using, again, an intermediate structure), from the first traversal of a block to its second traversal, the names of the variables that are used in it.

Observe also that the environment computed for a block and used for processing the use-occurrences is the global environment for its nested blocks. Thus, only during the second traversal of a block (*i.e.*, after collecting all its declarations) the program actually begins the traversals of its nested blocks; as a consequence the computations related to first and second traversals are intermingled. Furthermore, the information on its nested blocks (the instructions they define and the blocks' level) has to be explicitly passed from the first to the second traversal of a block. This is also achieved by defining and constructing an intermediate data structure. In order to pass the necessary information from the first to the second traversal of a block, we define the following intermediate data structure:

$$\textbf{type } Prog_2 = [It_2] \qquad \textbf{data } It_2 = Block_2\ (Int, Prog) \mid Dupl_2\ \ Var \mid Use_2\ \ Var$$

Errors resulting from duplicate declarations, computed in the first traver-

sal, are passed to the second, using constructor $Dupl_2$. The level of a nested block, as well as the instructions it defines, are passed to the second traversal using constructor $Block_2$'s pair containing an integer and a sequence of instructions.

According to the strategy defined earlier, computing the semantic errors that occur in a block sentence would resume to:

$$
\begin{aligned}
&semantics :: Prog \rightarrow [\,Var\,] \\
&semantics = missing \circ (duplicate\ 0\ [\,])
\end{aligned}
$$

The function *duplicate* detects duplicate variable declarations by collecting all the declarations occurring in a block. It is defined as follows:

$$
\begin{aligned}
&duplicate :: Int \rightarrow Env \rightarrow Prog \rightarrow (Prog_2, Env) \\
&duplicate\ lev\ ds\ [\,] = ([\,], ds) \\
&\\
&duplicate\ lev\ ds\ ((Use\ var) : its) \\
&\quad = \mathbf{let}\ (its_2, ds') = duplicate\ lev\ ds\ its \\
&\qquad \mathbf{in}\ \ ((Use_2\ var) : its_2, ds') \\
&\\
&duplicate\ lev\ ds\ ((Decl\ var) : its) \\
&\quad = \mathbf{let}\ (its_2, ds') = duplicate\ lev\ ((var, lev) : ds)\ its \\
&\qquad \mathbf{in\ if}\ ((var, lev) \in ds)\ \mathbf{then}\ ((Dupl_2\ var) : its_2, ds') \\
&\qquad\qquad\qquad\qquad\qquad \mathbf{else}\ (its_2, ds') \\
&\\
&duplicate\ lev\ ds\ ((Block\ nested) : its) \\
&\quad = \mathbf{let}\ (its_2, ds') = duplicate\ lev\ ds\ its \\
&\qquad \mathbf{in}\ \ ((Block_2\ (lev + 1, nested)) : its_2, ds')
\end{aligned}
$$

Besides detecting the invalid declarations, the *duplicate* function also computes a data structure, of type $Prog_2$, that is later traversed in order to detect variables that are used without being declared. This detection is performed by function *missing*, that is defined such as:

$$
\begin{aligned}
&missing :: (Prog_2, Env) \rightarrow [\,Var\,] \\
&missing\ ([\,], \_) = [\,] \\
&\\
&missing\ ((Use_2\ var) : its_2, env)
\end{aligned}
$$

$$
\begin{aligned}
&= \textbf{let } errs = missing\ (its_2, env) \\
&\quad \textbf{in if } (var \in (map\ \pi_1\ env)) \textbf{ then } errs \\
&\qquad\qquad\qquad\qquad\qquad \textbf{else } var : errs
\end{aligned}
$$

$$
\begin{aligned}
&missing\ ((Dupl_2\ var) : its_2, env) \\
&\quad = var : missing\ (its_2, env)
\end{aligned}
$$

$$
\begin{aligned}
&missing\ ((Block_2\ (lev, its)) : its_2, env) \\
&\quad = \textbf{let } errs_1 = missing \circ (duplicate\ lev\ env)\ \$\ its \\
&\qquad\quad errs_2 = missing\ (its_2, env) \\
&\quad \textbf{in } errs_1 +\!\!+ errs_2
\end{aligned}
$$

The *semantics* program constructs an intermediate structure, of type $Prog_2$, that we would like to eliminate with fusion. In order to apply our rule, we first have to express the functions *missing* and *duplicate* in terms of pfold and buildp for $Prog_2$, respectively. The functor that captures the structure of $Prog_2$ lists is:

$$
\begin{aligned}
&L\ b = () + (It_2, b) \\
&L \quad :: (b \to c) \to (L\ b \to L\ c) \\
&L\ f = id + id\ \times\ f
\end{aligned}
$$

Pfold and buildp for $Prog_2$ lists are defined as follows.

$$
\begin{aligned}
&pfold_L :: (z \to b, It_2 \to b \to z \to b) \to (Prog_2, z) \to b \\
&pfold_L\ (hnil, hcons) = p_L \\
&\quad \textbf{where } p_L\ ([\,], z) = hnil\ z \\
&\qquad\quad p_L\ (h : t, z) = hcons\ h\ (p_L\ (t, z))\ z
\end{aligned}
$$

$$
\begin{aligned}
&buildp_L :: (\forall\ a\ .\ ([a], a \to [a] \to [a]) \to c \to ([a], z)) \to c \to (Prog_2, z) \\
&buildp_L\ g = g\ ([\,], (:))
\end{aligned}
$$

We may now write *missing* and *duplicate* in terms of them; function *missing* is defined, in terms of $pfold_L$, as:

$$
\begin{aligned}
&missing = pfold_L\ (hnil, hcons) \\
&\quad \textbf{where } hnil\ z = [\,]
\end{aligned}
$$

$$
\begin{aligned}
&hcons\ (Use_2\ var)\ errs\ env \\
&\quad = \mathbf{if}\ (var \in (map\ \pi_1\ env)) \\
&\qquad \mathbf{then}\ errs \\
&\qquad \mathbf{else}\ var : errs
\end{aligned}
$$

$$
hcons\ (Dupl_2\ var)\ errs\ env = var : errs
$$

$$
\begin{aligned}
&hcons\ (Block_2\ (lev, its))\ errs\ env \\
&\quad = \mathbf{let}\ errs' = missing \circ (duplicate\ lev\ env)\ \$\ its \\
&\qquad \mathbf{in}\ \ errs' +\!\!+ errs
\end{aligned}
$$

and function *duplicate* is defined, in terms of $buildp_L$, as:

$$
\begin{aligned}
&duplicate\ lev\ ds = buildp_L\ (g\ lev\ ds) \\
&\quad \mathbf{where}\ g\ lev\ ds\ (nil, cons)\ [\,] = (nil, ds)
\end{aligned}
$$

$$
\begin{aligned}
&g\ lev\ ds\ (nil, cons)\ ((Use\ var) : its) \\
&\quad = \mathbf{let}\ (its_2, ds') = g\ lev\ ds\ (nil, cons)\ its \\
&\qquad \mathbf{in}\ \ (cons\ (Use_2\ var)\ its_2, ds')
\end{aligned}
$$

$$
\begin{aligned}
&g\ lev\ ds\ (nil, cons)\ ((Decl\ var) : its) \\
&\quad = \mathbf{let}\ (its_2, ds') = g\ lev\ ((var, lev) : ds)\ (nil, cons)\ its \\
&\qquad \mathbf{in}\ \ \mathbf{if}\ ((var, lev) \in ds) \\
&\qquad\quad \mathbf{then}\ (cons\ (Dupl_2\ var)\ its_2, ds') \\
&\qquad\quad \mathbf{else}\ \ (its_2, ds')
\end{aligned}
$$

$$
\begin{aligned}
&g\ lev\ ds\ (nil, cons)\ ((Block\ nested) : its) \\
&\quad = \mathbf{let}\ (its_2, ds') = g\ lev\ ds\ (nil, cons)\ its \\
&\qquad \mathbf{in}\ (cons\ (Block_2\ (lev + 1, nested))\ its_2, ds')
\end{aligned}
$$

Recall the definition we want to transform:

$$
\begin{aligned}
&semantics :: Prog \rightarrow [\,Var\,] \\
&semantics = missing \circ (duplicate\ (0, [\,]))
\end{aligned}
$$

and notice that we have just given this composition an explicit $pfold \circ buildp$ form. By application of Law 3.4.1 to the above definition, we obtain the program:

$$
\begin{array}{l}
\mathit{semantics}\ \mathit{its} = \mathit{errs} \\
\quad \textbf{where}\ (\mathit{errs}, \mathit{env}) = g\ 0\ [\,]\ (\mathit{knil}, \mathit{kcons})\ \mathit{its} \\
\qquad\quad \mathit{knil} = \mathit{hnil}\ \mathit{env} \\
\qquad\quad \mathit{kcons}\ x\ y = \mathit{hcons}\ x\ y\ \mathit{env}
\end{array}
$$

Inlining the above definition, we obtain:

$$
\begin{array}{l}
\mathit{semantics}\ \mathit{its} = \mathit{errs} \\
\quad \textbf{where}\ (\mathit{errs}, \mathit{env}) = \mathit{gk}\ 0\ [\,]\ \mathit{its} \\
\qquad\quad \mathit{gk}\ \mathit{lev}\ \mathit{ds}\ [\,] = ([\,], \mathit{ds}) \\
\qquad\quad \mathit{gk}\ \mathit{lev}\ \mathit{ds}\ ((\mathit{Use}\ \mathit{var}) : \mathit{its}) \\
\qquad\qquad = \textbf{let}\ (\mathit{errs}, \mathit{ds}') = \mathit{gk}\ \mathit{lev}\ \mathit{ds}\ \mathit{its} \\
\qquad\qquad\quad \textbf{in}\ \ (\textbf{if}\ (\mathit{var} \in (\mathit{map}\ \pi_1\ \mathit{env}))\ \textbf{then}\ \mathit{errs} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{else}\ \mathit{var} : \mathit{errs}, \mathit{ds}') \\
\qquad\quad \mathit{gk}\ \mathit{lev}\ \mathit{ds}\ ((\mathit{Decl}\ \mathit{var}) : \mathit{its}) \\
\qquad\qquad = \textbf{let}\ (\mathit{errs}, \mathit{ds}') = \mathit{gk}\ \mathit{lev}\ ((\mathit{var}, \mathit{lev}) : \mathit{ds})\ \mathit{its} \\
\qquad\qquad\quad \textbf{in}\ \ \textbf{if}\ ((\mathit{var}, \mathit{lev}) \in \mathit{ds})\ \textbf{then}\ (\mathit{var} : \mathit{errs}, \mathit{ds}') \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{else}\ (\mathit{errs}, \mathit{ds}') \\
\qquad\quad \mathit{gk}\ \mathit{lev}\ \mathit{ds}\ ((\mathit{Block}\ \mathit{nested}) : \mathit{its}) \\
\qquad\qquad = \textbf{let}\ (\mathit{errs}, \mathit{ds}') = \mathit{gk}\ \mathit{lev}\ \mathit{ds}\ \mathit{its} \\
\qquad\qquad\quad \textbf{in}\ \ (\textbf{let}\ \mathit{errs}' = \mathit{missing} \circ (\mathit{duplicate}\ (\mathit{lev} + 1)\ \mathit{env})\ \$\ \mathit{nested} \\
\qquad\qquad\qquad\ \ \textbf{in}\ \mathit{errs}' +\!\!+ \mathit{errs}, \mathit{ds}')
\end{array}
$$

We may notice that the above program is a circular one: the environment of a BLOCK program (variable *env*) is being computed at the same time it is being used. The introduction of such circularity made it possible to eliminate some intermediate structures that occurred in the program we started with: the intermediate list of instructions that was computed in order to glue the two traversals of the outermost level of a BLOCK sentence has been eliminated by application of Law 3.4.1. We may also notice, however, that, for nested blocks, in the definition

$$\mathit{gk}\ \mathit{lev}\ \mathit{ds}\ ((\mathit{Block}\ \mathit{nested}) : \mathit{its})$$

$$
\begin{aligned}
&= \textbf{let } (errs, ds') = gk\ lev\ ds\ its \\
&\quad \textbf{in } (\textbf{let } errs' = missing \circ (duplicate\ (lev+1)\ env)\ \$\ nested \\
&\qquad \textbf{in } errs' +\!\!+ errs, ds')
\end{aligned}
$$

an intermediate structure is still being used in order to glue functions *missing* and *duplicate* together. This intermediate structure can easily be eliminated: we have already expressed function *missing* in terms of pfold, and function *duplicate* in terms of buildp. Therefore, by direct application of Law 3.4.1 to the above function composition, we obtain:

$$
\begin{aligned}
&gk\ lev\ ds\ ((Block\ nested) : its) \\
&\quad = \textbf{let } (errs, ds') = gk\ lev\ ds\ its \\
&\quad\quad \textbf{in } (\textbf{let } (errs', env') = g\ (lev+1)\ env\ (knil, kcons)\ nested \\
&\qquad\qquad\quad \textbf{where } knil = hnil\ env' \\
&\qquad\qquad\qquad\qquad\quad kcons\ x\ y = hcons\ x\ y\ env' \\
&\qquad\qquad \textbf{in } errs' +\!\!+ errs, ds')
\end{aligned}
$$

Again, we could inline the definition of function g into a new function, for example, into function gk'. However, the definition of gk' would exactly match the definition of gk, except for the fact that where gk searched for variable declarations in the environment env, gk' needs to search for them in the environment env'.

In order to use the same function for both gk and gk', we choose to add an extra argument to function gk. Such argument will make possible to use circular definitions to pass the appropriate environment variable to the appropriate block of instructions (the top level block or the nested ones).

It should be clear that, in general, this extra effort is not required: it was necessary, in this particular example, due to the facts that it is possible to calculate two circular definitions from the straightforward solution and that both circular functions share almost the exact same definition. In all other cases, inlining the calculated circular program is enough to derive an elegant and efficient *lazy* program from a function composition between a pfold and a buildp.

We finally obtain the program:

$$
\begin{array}{l}
semantics\ its = errs \\
\quad \textbf{where}\ (errs, env) = gk\ 0\ [\,]\ env\ its \\
\qquad\quad gk\ lev\ ds\ env\ [\,] = ([\,], ds) \\
\\
\qquad\quad gk\ lev\ ds\ env\ ((Use\ var) : its) \\
\qquad\qquad = \textbf{let}\ (errs, ds') = gk\ lev\ ds\ env\ its \\
\qquad\qquad\quad \textbf{in}\ \ (\textbf{if}\ (var \in (map\ \pi_1\ env)) \\
\qquad\qquad\qquad\quad \textbf{then}\ errs \\
\qquad\qquad\qquad\quad \textbf{else}\ var : errs, ds') \\
\\
\qquad\quad gk\ lev\ ds\ env\ ((Decl\ var) : its) \\
\qquad\qquad = \textbf{let}\ (errs, ds') = gk\ lev\ ((var, lev) : ds)\ env\ its \\
\qquad\qquad\quad \textbf{in}\ \ \textbf{if}\ ((var, lev) \in ds) \\
\qquad\qquad\qquad\quad \textbf{then}\ (var : errs, ds') \\
\qquad\qquad\qquad\quad \textbf{else}\ (errs, ds') \\
\\
\qquad\quad gk\ lev\ ds\ env\ ((Block\ nested) : its) \\
\qquad\qquad = \textbf{let}\ (errs, ds') = gk\ lev\ ds\ env\ its \\
\qquad\qquad\quad \textbf{in}\ \ (\textbf{let}\ (errs', env') = gk\ (lev + 1)\ env\ env'\ nested \\
\qquad\qquad\qquad\quad \textbf{in}\ errs' +\!\!+ errs, ds')
\end{array}
$$

Regarding the above program, we may notice that it is a circular one. Indeed, two circularities occur in its definition:

$$
\begin{array}{l}
\ldots \\
\qquad\quad (errs, env) = gk\ 0\ [\,]\ env\ its \\
\ldots \\
\qquad\quad (errs', env') = gk\ (lev + 1)\ env\ env'\ nested \\
\ldots
\end{array}
$$

The introduction of these circularities, by application of our fusion Law, completely eliminated the intermediate lists of It_2 instructions that were used in the straightforward solution we started with. Furthermore, such circularities made it possible to compute the list of semantic errors that occur in a BLOCK program by traversing it only once.

3.6 Conclusions

In this Chapter we have presented a new program transformation technique for intermediate structure elimination. The programs we are able of dealing with consist in the composition of a producer and a consumer functions. The producer constructs an intermediate structure that is later traversed by the consumer. Furthermore, we allow the producer to compute additional values that may be needed by the consumer. This kind of compositions is general enough to deal with a wide number of practical examples. Our approach is calculational, and proceeds in two steps: we apply standard deforestation methods to obtain intermediate structure-free programs and we introduce circular definitions to avoid multiple traversals that are introduced by deforestation. Since that, in the first step, we apply standard fusion techniques, the expressive power of our rule is then bound by deforestation.

We introduce a new calculational rule conceived using a similar approach to the one used in the *fold/build* rule: our rule is also based on parametricity properties of the functions involved. Therefore, it has the same benefits and drawbacks of *fold/build* since it assumes that the functions involved are instances of specific program schemes. Therefore, it could be used, like *fold/build*, in the context of a compiler. In fact, we have used the rewrite rules (RULES pragma) of the Glasgow Haskell Compiler (GHC) in order to obtain a prototype implementation of our fusion rule.

According to Danielsson et al. (2006), the calculation rule we present in this Chapter is morally correct *only*, in Haskell. In fact, in the formal proof of our rule, surjective pairing (Law (3.4)) is applied twice to the result of function g. However, (3.4) is not valid in Haskell: though it holds for defined values, it fails when the result of function g is undefined, because $\bot$ is different from $(\bot, \bot)$ as a consequence of lifted products. Therefore, (3.4) is morally correct *only* and, in the same sense, so is our rule. We may, however, argue that, for all cases with practical interest (the ones for which function g produces defined results), our rule directly applies in Haskell. Furthermore, due to the presence of *seq* in Haskell, further strictness pre-conditions may need to be defined in our rule in order to guarantee its correctness in Haskell

(Johann and Voigtländer 2004).

The rule that we propose is easy to apply: in this paper, we have presented a real example that shows that our rule is effective in its aim. Other examples may be found in (Fernandes et al. 2007). The calculation of circular programs may be understood as an intermediate stage: the circular programs we calculate may be further transformed into very efficient, completely data structure free programs.

Chapter 4

Calculation of Monadic Circular Programs

Functional programs often combine separate parts using intermediate data structures for communicating results. Such programs are easier to understand and maintain, but suffer from inefficiencies due to the generation of those data structures. Indeed, in order to eliminate them, several program transformation techniques have been proposed. One such technique is shortcut fusion, and has been studied in the context of both pure and monadic functional programs.

In the previous Chapter, we have extended shortcut fusion: in addition to intermediate structures, the program parts may now communicate context information, and still it is possible to eliminate the intermediate structures. This is achieved by transforming the original function composition into a circular program. This new technique, however, has been studied in the context of purely functional programs only. In this Chapter, we propose an extension to this new form of fusion, but in the context of monadic programming. Our extension is provided in terms of generic calculation rules, that can be uniformly defined for a wide class of data types and monads.

4.1 Introduction

Functional programs often combine separate parts of the program using intermediate structures for communicating results. In general, we have programs such as $prog = cons \circ prod$, where *prod* is called the producer function and *cons* is called the consumer function. Programs so defined are modular and have many benefits, such as clarity and maintainability, but suffer from innefficiencies caused by the generation of the intermediate data structures that glue functions *cons* and *prod* together.

In response to this problematic, some program transformation techniques have been studied aiming at the elimination of intermediate data structures. One of these techniques, that we have reviewed in detail in Section 3.3, is known as shortcut fusion, or shortcut deforestation (Gill et al. 1993). This technique eliminates the generation of the intermediate structure, of type b, when $prod :: a \to b$, $cons :: b \to c$ and $prog = cons \circ prod$. Shortcut deforestation has recently been studied and applied also in the context of monadic functional programs (Ghani and Johann 2008; Manzino and Pardo 2008).

In Chapter 3, we have proposed circular programs as an extension to *standard* shortcut fusion: in order to achieve intermediate structure deforestation in programs such as $prog = cons \circ prod$, where $prod :: a \to (b, z)$ and $cons :: (b, z) \to c$, we transform *prog* into a circular program. This means that the producer function may generate, besides the intermediate structure b, an additional value, of type z, that the consumer function may need to compute its result. Later, a calculation rule is applied to *prog*, which is transformed into an equivalent circular program that does not construct any intermediate structure and that traverses the input data (of type a) only once. The rule applied to *prog* is generic in the sense that it can be applied to a wide range of programs and datatypes. However, it does not handle monadic functional programs, that is, programs that, for example, rely on a global state or perform I/O operations. Thus, the rule has a limited applicability scope since several functions, like compilers, pretty-printers or parsers do rely on global effects.

Our motivation for the work presented in this Chapter is to extend shortcut fusion to this kind of programs in the context of monadic programming. The goal is to achieve fusion of monadic programs, maintaining the global effects. We study two cases: the case where the producer function is monadic and the consumer is given by a pure function, and the case where both functions are monadic.

Our extension is provided in terms of calculational rules. An important feature of our rules is that they are generic, in the sense that they can be given by a uniform, single definition that can be instantiated to a wide class of algebraic data types and monads. Throughout we will use Haskell notation, assuming a cpo semantics (in terms of pointed cpos), but without the presence of the *seq* function (Johann and Voigtländer 2004).

This Chapter is organized as follows. Sections 4.2 and 4.3 present two motivating examples that serve to illustrate the applicability of our technique. The generic constructions that give rise to the specific program schemes and laws presented in those examples are developed in Section 4.4. Finally, in Section 4.5 we draw some conclusions and describes directions for future work.

4.2 Bit String Transformation

To illustrate our technique we first consider an example based on a simple bit string conversion that has applications in criptography (Harald Baier and Margraf 2007). Suppose we want to transform a sequence of bits into a new one, of the same length, by applying the exclusive or between each bit and the binary sum (sum modulo 2) of the sequence. We will consider that the input sequence is given as a string of bits, which will be parsed into a list and then transformed. It is in the parsing phase that computational effects will come into play, as we will use a monadic parser.

Suppose we are given the string `"101110110001"`. To transform this string of bits, we start by parsing it, computing as result a list of bits $[1, 0, 1, 1, 1, 0, 1, 1, 0, 0, 0, 1]$, and its binary sum (1 in this case). Having the list and the binary sum, the original sequence is transformed into this one

$[0,1,0,0,0,1,0,0,1,1,1,0]$ after applying the exclusive or of each bit with 1 (the binary sum).

To construct the parser, we adopt a usual definition of parser monad (see (Hutton and Meijer 1998) for more details):

$$\begin{aligned}
&\textbf{newtype } \mathit{Parser}\ a = P\ (\mathit{String} \to [(a, \mathit{String})])\\
&\\
&\textbf{instance } \mathit{Monad}\ \mathit{Parser}\ \textbf{where}\\
&\quad \mathit{return}\ a = P\ (\lambda cs \to [(a, cs)])\\
&\quad p \gg\!= f \quad = P\ (\lambda cs \to \mathit{concat}\ [\mathit{parse}\ (f\ a)\ cs' \mid (a, cs') \leftarrow \mathit{parse}\ p\ cs])\\
&\\
&\mathit{parse} :: \mathit{Parser}\ a \to \mathit{String} \to [(a, \mathit{String})]\\
&\mathit{parse}\ (P\ p) = p\\
&\\
&(< \mid >) :: \mathit{Parser}\ a \to \mathit{Parser}\ a \to \mathit{Parser}\ a\\
&(P\ p)\ \ < \mid >\ \ (P\ q) = P\ (\lambda cs \to \textbf{case}\ p\ cs +\!\!+ q\ cs\ \textbf{of}\ [\,] \qquad \to [\,]\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (x : xs) \to [x])\\
&\\
&\mathit{pzero} :: \mathit{Parser}\ a\\
&\mathit{pzero} = P\ (\lambda cs \to [\,])\\
&\\
&\mathit{item} :: \mathit{Parser}\ \mathit{Char}\\
&\mathit{item} = P\ (\lambda cs \to \textbf{case}\ cs\ \textbf{of}\ [\,] \qquad \to [\,]\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (c : cs) \to [(c, cs)])
\end{aligned}$$

Alternatives are represented by a deterministic choice operator $(< \mid >)$, which returns at most one result. The parser *pzero* is a parser that always fails. The *item* parser returns the first character in the input string.

We can use these simple parser combinators to define parsers for bits and bit strings. The binary sum is calculated as the exclusive or of the bits of the parsed sequence. We write $\oplus$ to denote exclusive or over the type *Bit*.

$$\begin{aligned}
&\textbf{data } \mathit{Bit} = Z \mid O\\
&\\
&\mathit{bit} :: \mathit{Parser}\ \mathit{Bit}\\
&\mathit{bit} = \textbf{do}\ \{\, c \leftarrow \mathit{item};\\
&\qquad\qquad \textbf{case}\ c\ \textbf{of}\ \texttt{'0'} \to \mathit{return}\ Z\\
&\qquad\qquad\qquad\qquad\ \ \texttt{'1'} \to \mathit{return}\ O
\end{aligned}$$

$$\_ \quad \rightarrow pzero\,\}$$

$$\begin{aligned}
&bitstring :: Parser\ ([Bit], Bit)\\
&bitstring = \mathbf{do} \quad b \quad \leftarrow bit\\
&\qquad\qquad\quad (bs, s) \leftarrow bitstring\\
&\qquad\qquad\quad return\ (b : bs, exor\ b\ s)\,\}\\
&\qquad\quad < | > return\ ([\,], Z)
\end{aligned}$$

Now, we implement the transformation function:

$$\begin{aligned}
&transform :: ([Bit], Bit) \rightarrow [Bit]\\
&transform\ ([\,], _) \quad = [\,]\\
&transform\ (b : bs, s) = (exor\ b\ s) : transform\ (bs, s)
\end{aligned}$$

In summary, the transformation consists of:

$$\begin{aligned}
&shift :: Parser\ [Bit]\\
&shift = \mathbf{do}\ (bs, s) \leftarrow bitstring\\
&\qquad\quad return\ (transform\ (bs, s))
\end{aligned}$$

We may notice that the above solution constructs an intermediate list of bits that we would like to eliminate with fusion. The fusion law to be used is a law in the style of shortcut fusion, similar to that conceived, in Chapter 3, for the derivation of purely functional circular programs, but with the difference that now it deals with monadic functions. Below, we present the specific instance for lists (which is the type of the intermediate structure), and in Section 4.4 we show that both the law and the programs schemes respond to generic definitions that can be formulated for several datatypes.

Like in standard shortcut fusion (Gill et al. 1993), our law assumes that the producer and the consumer (*bitstring* and *transform* in this case) are expressed in terms of certain program schemes. In standard shortcut fusion the consumer is required to be given by a structural recursive definition in terms of a recursion scheme called fold (usually called *foldr* in the case of lists (Bird 1998)). In our law, we also require the consumer to be given by a structural recursive definition, but in terms of a variation of fold, called *pfold*, which admits as input an additional constant parameter to be used

along the recursive calls:

$$
\begin{aligned}
&pfold_L :: (z \to b, a \to b \to z \to b) \to ([a], z) \to b \\
&pfold_L\ (hnil, hcons) = p_L \\
&\quad \textbf{where}\ p_L\ ([\,], z) \quad\ = hnil\ z \\
&\qquad\qquad p_L\ (a : as, z) = hcons\ a\ (p_L\ (as, z))\ z
\end{aligned}
$$

Like in standard shortcut fusion, we require the producer to be able to show that the list constructors can be abstracted from the process that generates the intermediate list. The difference with the standard case is that we consider producers that generate the intermediate list as part of a pair which in turn is the result of monadic computation. This is expressed by a function called $mbuildp_L$:

$$
\begin{aligned}
&mbuildp_L :: Monad\ m \Rightarrow (\forall\ b\ .\ (b, a \to b \to b) \to m\ (b, z)) \to m\ ([a], z) \\
&mbuildp_L\ g = g\ ([\,], (:))
\end{aligned}
$$

Having stated the forms required to the producer and the consumer it is now possible to formulate the law.

Law 4.2.1 (pfold/mbuildp for lists)

$$
\begin{aligned}
&\quad \textbf{do}\ (xs, z) \leftarrow mbuildp_L\ g \\
&\qquad return\ (pfold_L\ (hnil, hcons)\ (xs, z)) \\
&= \\
&\quad \textbf{mdo}\ (v, z) \leftarrow \textbf{let}\ knil \quad\ = hnil\ z \\
&\qquad\qquad\qquad\qquad\ kcons\ x\ y = hcons\ x\ y\ z \\
&\qquad\qquad\qquad\ \textbf{in}\ g\ (knil, kcons) \\
&\qquad\ return\ v
\end{aligned}
$$

This law transforms a monadic composition, where the producer is an effectful function but may not necessarily the consumer be, into a single monadic function with a circular argument z. Indeed, z is a value computed by $g\ (knil, kcons)$ but in turn used by $knil$ and $kcons$. An interesting feature of this law is the fact that the introduction of the circularity on z requires the use of a recursive binding within a monadic computation, which can be

expressed in terms of the so-called **mdo**-notation (a recursive **do**) supported by Haskell (Erkök and Launchbury 2002).

To see the law in action, we write *transform* and *bitstring* in terms of $pfold_L$ and $mbuildp_L$, respectively:

$$
\begin{array}{l}
transform = pfold_L\ (hnil, hcons) \\
\quad \textbf{where}\ hnil\ \_ \qquad\ = [\,] \\
\qquad\qquad\ \ hcons\ b\ r\ s = (exor\ b\ s) : r
\end{array}
$$

$$
\begin{array}{l}
bitstring = mbuildp_L\ g \\
\quad \textbf{where}\ g\ (nil, cons) = \textbf{do} \qquad b \qquad\ \leftarrow bit \\
\qquad\qquad\qquad\qquad\qquad\qquad\ \ (bs, s) \leftarrow g\ (nil, cons) \\
\qquad\qquad\qquad\qquad\qquad\qquad\ \ return\ (cons\ b\ bs, exor\ b\ s) \\
\qquad\qquad\qquad\qquad\qquad\ <|>\ return\ (nil, Z)
\end{array}
$$

Then, by applying Law 4.2.1 we obtain:

$$
\begin{array}{l}
shift = \textbf{mdo}\ (bs, s) \leftarrow g\ ([\,], \lambda b\ r \rightarrow (exor\ b\ s) : r) \\
\qquad\qquad return\ bs
\end{array}
$$

Inlining, we get the following circular monadic program, that avoids the construction of the intermediate list of bits:

$$
\begin{array}{l}
shift = \textbf{mdo}\ (bs, s) \leftarrow \textbf{let}\ gk = \textbf{do} \qquad b \qquad\ \leftarrow bit \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\ \ (bs', s') \leftarrow gk \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\ \ return\ ((exor\ b\ s) : bs', exor\ b\ s') \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\ <|>\ return\ ([\,], Z) \\
\qquad\qquad\qquad\qquad\qquad\qquad\ \textbf{in}\ gk \\
\qquad\qquad\qquad\qquad return\ bs
\end{array}
$$

4.3 The Algol 68 scope rules Revisited

Let us now consider an improvement on the example we presented in Section 3.5: the Algol 68 scope rules. Our goal was to construct a program to deal with the scope rules of a block structured language, such that, for the input Block program

$$
\begin{array}{l}
[\,\mathbf{use}\ y; \mathbf{decl}\ x; \\
\quad [\,\mathbf{decl}\ y; \mathbf{use}\ y; \mathbf{use}\ w;\,] \\
\mathbf{decl}\ x; \mathbf{decl}\ y;\,]
\end{array}
$$

it produces the list of errors $[w, x]$: at the outer level, the variable x has been declared twice and the use of the variable w, at the inner level, has no binding occurrence at all.

Now, we still aim to develop a semantic function that analyses a sequence of instructions and computes a list containing the variable identifiers of the instructions which do not obey to the rules of the language. However, when such an instruction is found, we want to output an error message explaining the programming error encountered.

So, for the example program considered, we want to output the messages:

```
"Duplicate: decl x"

"Missing:   decl w"
```

A straightforward implementation of the semantic function, that closely follows the solution presented in Section 3.5, may be defined:

$$
\begin{array}{l}
semantics :: Prog \rightarrow IO\ [\,Var\,] \\
semantics\ p = \mathbf{do}\ (p', env) \leftarrow duplicate\ 0\ [\,]\ p \\
\qquad\qquad\qquad\quad missing\ (p', env)
\end{array}
$$

Notice that function *semantics* now needs to be monadic (it returns a value within the *IO* monad), as functions *duplicate* and *missing* need to output the error messages. The function *duplicate* detects duplicate variable declarations by collecting all the declarations occurring in a program. It is a monadic function since it needs to output error messages resulting from the errors it detects. The definition of such function is as follows:

$$
\begin{array}{l}
duplicate :: Int \rightarrow Env \rightarrow Prog \rightarrow IO\ (Prog_2, Env) \\
\\
duplicate\ lev\ ds\ [\,] = return\ ([\,], ds) \\
\\
duplicate\ lev\ ds\ ((Use\ var) : its) \\
\quad = \mathbf{do}\ (its_2, ds') \leftarrow duplicate\ lev\ ds\ its
\end{array}
$$

```
                return ((Use2 var) : its2, ds')

duplicate lev ds ((Decl var) : its)
  = do (its2, ds') <- duplicate lev ((var, lev) : ds) its
       if ((var, lev) ∈ ds)
         then do putStrLn ("Duplicate: decl " ++ var)
                 return ((Dupl2 var) : its2, ds')
         else return (its2, ds')

duplicate lev ds ((Block nested) : its)
  = do (its2, ds') <- duplicate lev ds its
       return ((Block2 (lev + 1, nested)) : its2, ds')
```

Besides detecting the invalid declarations, function *duplicate* also computes a data structure, of type $Prog_2$, that is later traversed in order to detect variables that are used without being declared. This detection is performed by function *missing*, which is monadic as it also outputs error messages:

```
missing :: (Prog2, Env) → IO [Var]

missing ([], _) = return []

missing ((Use2 var) : its2, env)
  = do errs <- missing (its2, env)
       if (var ∈ (map π1 env))
         then return errs
         else do putStrLn ("Missing: decl " ++ var)
                 return (var : errs)

missing ((Dupl2 var) : its2, env)
  = do errs <- missing (its2, env)
       return (var : errs)

missing ((Block2 (lev, its)) : its2, env)
  = do errs1 <- do (p', env) <- duplicate lev env its
                   missing (p', env)
       errs2 <- missing (its2, env)
```

$$return\ (errs_1 +\!\!+ errs_2)$$

We would like to eliminate the intermediate structure of type $Prog_2$ generated by *duplicate*. If we attempted to directly apply Law 4.2.1 for that aim, then we would see that in this case the result of the law is a function that returns a monadic computation which in turn yields a monadic computation (and not a value) as result, that is, something of type $m\ (m\ a)$, for some a. This is because the consumer is also monadic. To obtain a value and not a computation as final result, it is simply necessary to run the computation. This gives the following shortcut fusion law, which requires the same schemes for consumer and producer as Law 4.2.1 but is able to fuse two effectful functions.

Law 4.3.1 (Effectfull pfold/mbuildp for lists)

$$\begin{array}{l}
\quad \textbf{do}\ (xs, z) \leftarrow mbuildp_L\ g\ c \\
\qquad pfold_L\ (hnil, hcons)\ (xs, z) \\
= \\
\quad \textbf{mdo}\ (m, z) \leftarrow \textbf{let}\ knil = hnil\ z \\
\qquad\qquad\qquad\qquad\quad kcons\ x\ y = hcons\ x\ y\ z \\
\qquad\qquad\qquad\quad \textbf{in}\ \ g\ (knil, kcons)\ c \\
\qquad\quad v \leftarrow m \\
\qquad\quad return\ v
\end{array}$$

Observe that, in this case, $hnil :: z \to m\ b$ and $hcons :: a \to m\ b \to z \to m\ b$, for some monad m, and therefore $pfold_L\ (hnil, hcons) :: ([a], z) \to m\ b$. Also, notice that,

$$\begin{array}{l}
mbuildp_L :: Monad\ m \Rightarrow (\forall\ b\ .\ (b, a \to b \to b) \to c \to m\ (b, z)) \\
\qquad\qquad\qquad\qquad\qquad\quad \to c \to m\ ([a], z)
\end{array}$$

that is, $mbuildp_L\ g$ is a function of type $c \to m\ ([a], z)$. It is in this way that it will be considered in Section 4.4 when we will define the generic formulation of the laws. However, in Section 4.2 it was defined as a value of type $m\ ([a], z)$ because that form is more appropriate for writing monadic parsers.

Now, if we write *missing* in terms of $pfold_L$

$$
\begin{aligned}
&missing = pfold_L\ (hnil, hcons) \\
&\quad \textbf{where}\ hnil\ z = return\ [\,] \\
&\qquad\qquad hcons\ (Use_2\ var)\ mr\ env \\
&\qquad\qquad\quad = \textbf{do}\ r \leftarrow mr \\
&\qquad\qquad\qquad\quad \textbf{if}\ (var \in (map\ \pi_1\ env)) \\
&\qquad\qquad\qquad\qquad \textbf{then}\ return\ r \\
&\qquad\qquad\qquad\qquad \textbf{else do}\ putStrLn\ (\texttt{"Missing: decl "} +\!\!+ var) \\
&\qquad\qquad\qquad\qquad\qquad\quad return\ (var : r) \\
&\qquad\qquad hcons\ (Dupl_2\ var)\ mr\ env \\
&\qquad\qquad\quad = \textbf{do}\ r \leftarrow mr \\
&\qquad\qquad\qquad\quad return\ (var : r) \\
&\qquad\qquad hcons\ (Block_2\ (lev, its))\ mr\ env \\
&\qquad\qquad\quad = \textbf{do}\ (p', env') \leftarrow duplicate\ lev\ env\ \$\ its \\
&\qquad\qquad\qquad\quad errs' \leftarrow missing\ (p', env') \\
&\qquad\qquad\qquad\quad r \leftarrow mr \\
&\qquad\qquad\qquad\quad return\ (errs' +\!\!+ r)
\end{aligned}
$$

and *duplicate* in terms of $mbuildp_L$

$$
\begin{aligned}
&duplicate\ lev\ ds = buildp_L\ (g\ lev\ ds) \\
&\quad \textbf{where}\ g\ lev\ ds\ (nil, cons)\ [\,] = return\ (nil, ds) \\
&\qquad\qquad g\ lev\ ds\ (nil, cons)\ ((Use\ var) : its) \\
&\qquad\qquad\quad = \textbf{do}\ (its_2, ds') \leftarrow g\ lev\ ds\ (nil, cons)\ its \\
&\qquad\qquad\qquad\quad return\ (cons\ (Use_2\ var)\ its_2, ds') \\
&\qquad\qquad g\ lev\ ds\ (nil, cons)\ ((Decl\ var) : its) \\
&\qquad\qquad\quad = \textbf{do}\ (its_2, ds') \leftarrow g\ lev\ ((var, lev) : ds)\ (nil, cons)\ its \\
&\qquad\qquad\qquad\quad \textbf{if}\ ((var, lev) \in ds) \\
&\qquad\qquad\qquad\qquad \textbf{then do}\ putStrLn\ (\texttt{"Duplicate: decl "} +\!\!+ var) \\
&\qquad\qquad\qquad\qquad\qquad\quad return\ (cons\ (Dupl_2\ var)\ its_2, ds') \\
&\qquad\qquad\qquad\qquad \textbf{else}\ \ return\ (its_2, ds')
\end{aligned}
$$

$$
\begin{aligned}
&g\ lev\ ds\ (nil, cons)\ ((Block\ nested) : its) \\
&\quad = \mathbf{do}\ (its_2, ds') \leftarrow g\ lev\ ds\ (nil, cons)\ its \\
&\qquad\quad return\ (cons\ (Block_2\ (lev + 1, nested))\ its_2, ds')
\end{aligned}
$$

we can apply Law 4.3.1 to *semantics* obtaining a deforested circular definition, which, when inlined, gives the following:

$$
\begin{aligned}
&semantics\ its = \\
&\quad \mathbf{mdo}\ (mr, env) \leftarrow \mathbf{let} \\
&\qquad gk\ lev\ ds\ env\ [\,] = return\ (return\ [\,], ds) \\
&\qquad gk\ lev\ ds\ env\ ((Use\ var) : its) \\
&\qquad\quad = \mathbf{do}\ (mr, ds') \leftarrow gk\ lev\ ds\ env\ its \\
&\qquad\qquad return\ (\mathbf{do}\ r \leftarrow mr \\
&\qquad\qquad\qquad\qquad \mathbf{if}\ (var \in (map\ \pi_1\ env)) \\
&\qquad\qquad\qquad\qquad\quad \mathbf{then}\ return\ r \\
&\qquad\qquad\qquad\qquad\quad \mathbf{else\ do}\ putStrLn\ (\texttt{"Missing: decl"} +\!\!+ var) \\
&\qquad\qquad\qquad\qquad\qquad\quad return\ (var : r), ds') \\
&\qquad gk\ lev\ ds\ env\ ((Decl\ var) : its) \\
&\qquad\quad = \mathbf{do}\ (mr, ds') \leftarrow gk\ lev\ ((var, lev) : ds)\ env\ its \\
&\qquad\qquad \mathbf{if}\ ((var, lev) \in ds) \\
&\qquad\qquad\quad \mathbf{then\ do}\ putStrLn\ (\texttt{"Duplicate: decl"} +\!\!+ var) \\
&\qquad\qquad\qquad\quad return\ (\mathbf{do}\ r \leftarrow mr \\
&\qquad\qquad\qquad\qquad\qquad\quad return\ (var : r), ds') \\
&\qquad\qquad\quad \mathbf{else}\ return\ (mr, ds') \\
&\qquad gk\ lev\ ds\ env\ ((Block\ nested) : its) \\
&\qquad\quad = \mathbf{do}\ (mr, ds') \leftarrow gk\ lev\ ds\ env\ its \\
&\qquad\qquad return\ (\mathbf{mdo}\ (mr', env') \leftarrow gk\ (lev + 1)\ env\ env'\ nested \\
&\qquad\qquad\qquad\qquad r' \leftarrow mr' \\
&\qquad\qquad\qquad\qquad r \leftarrow mr \\
&\qquad\qquad\qquad\qquad return\ (r' +\!\!+ r), ds') \\
&\quad \mathbf{in}\ gk\ 0\ [\,]\ env\ its
\end{aligned}
$$

$$
\begin{aligned}
&r \leftarrow mr \\
&return\ r
\end{aligned}
$$

The above program is obtained by applying Law 4.3.1 twice to the *semantics* program: we apply the Law to the composition of functions *missing* and *duplicate* defined over a BLOCK sentence

$$
\begin{aligned}
semantics\ p = \mathbf{do}\ &(p', env) \leftarrow duplicate\ 0\ [\,]\ p \\
&missing\ (p', env)
\end{aligned}
$$

and to the same composition defined over the nested blocks of a sentence:

$$
\begin{aligned}
&missing\ ((Block_2\ (lev, its)) : its_2, env) \\
&\quad = \mathbf{do}\ errs_1 \leftarrow \mathbf{do}\ (p', env) \leftarrow duplicate\ lev\ env\ its \\
&\qquad\qquad\qquad missing\ (p', env) \\
&\qquad\ errs_2 \leftarrow missing\ (its_2, env) \\
&\qquad\ return\ (errs_1 +\!\!+ errs_2)
\end{aligned}
$$

Again, we have manually added an argument to function *gk*, in order to reuse the definition of *gk* to traverse all the blocks (top level and nested ones) that occur in a BLOCK program.

4.4 Calculating monadic circular programs, generically

In this section, we show that the definition of the program schemes *pfold* and *mbuildp*, and the *pfold/mbuildp* laws, presented for lists in the previous sections, are instances of generic definitions valid for a wide class of datatypes.

4.4.1 Extended shortcut fusion

Shortcut fusion laws for monadic programs can be obtained as a special case of an extended form of shortcut fusion that captures the case when the intermediate data structure is generated as part of another structure given by a functor. Such extension is based on an extended form of build: Given a functor F (signature of a datatype) and another functor N, we define

$$buildFN :: (\forall\ a\ .\ (F\ a \to a) \to c \to N\ a) \to c \to N\ \mu F$$
$$buildFN\ g = g\ in_F$$

This is a natural extension of the standard *build*. In fact, *build* can be obtained from *buildFN* by considering the identity functor as N. Moreover, *buildp* is also a particular case obtained by considering the functor $N\ a = (a, z)$ (Ghani and Johann 2008).

We can now state an extended form of shortcut fusion (see (Manzino and Pardo 2008; Ghani and Johann 2008) for more details and a proof):

Law 4.4.1 (EXTENDED FOLD/BUILD) *For strict h and strictness preserving N,*

$$mapN\ (foldF\ h) \circ buildFN\ g = g\ h$$

Similarly, we can also consider an extension for *buildp*:

$$\begin{aligned} buildpFN\ \ &:: (\forall\ a\ .\ (F\ a \to a) \to c \to N\ (a, z)) \\ &\to c \to N\ (\mu F, z) \\ buildpFN\ g &= g\ in_F \end{aligned}$$

and an associated shortcut fusion law.

Law 4.4.2 (EXTENDED FOLD/BUILDP) *For strict h and strictness-preserving N,*

$$mapN\ (prod\ (fold\ h)\ id) \circ buildpFN\ g = g\ h$$

Proof 4.1 *By considering $N'\ a = N\ (a, z)$, we have that $buildpFN\ g = build_{N'}\ g$ and $map_{N'}\ f = mapN\ (prod\ f\ id)$. Then, the left-hand side of the equation can be rewritten as: $map_{N'}\ (fold\ h) \circ build_{N'}\ g$. Finally, we apply Law 4.4.1.*

The following law is an immediate consequence of the previous one.

Law 4.4.3 *For strictness-preserving N and $g :: \forall\ a\ .\ (F\ a \to a) \to c \to N\ (a, z)$,*

$$mapN\ \pi_2 \circ g\ in_F = mapN\ \pi_2 \circ g\ h$$

Proof 4.2

$$
\begin{aligned}
& mapN\ \pi_2 \circ g\ in_F \\
= \quad & \{\ \text{(3.3) and functor } N\ \} \\
& mapN\ \pi_2 \circ mapN\ (prod\ (fold\ h)\ id) \circ g\ in_F \\
= \quad & \{\ \text{Law 4.4.2}\ \} \\
& mapN\ \pi_2 \circ g\ h
\end{aligned}
$$

4.4.2 Monadic shortcut fusion

The case we are interested in is when the functor N is the composition of a monad m with a product: $N\ a = m\ (a, z)$, for some type z, and $mapN\ f = mmap\ (prod\ f\ id)$, where

$$
\begin{aligned}
& mmap \qquad :: Monad\ m \Rightarrow (a \to b) \to (m\ a \to m\ b) \\
& mmap\ f\ m = \mathbf{do}\ \{a \leftarrow m; return\ (f\ a)\}
\end{aligned}
$$

In such a case the producer corresponds to a monadic version of buildp:

$$
\begin{aligned}
& mbuildpF :: Monad\ m \\
& \quad \Rightarrow (\forall\ a\ .\ (F\ a \to a) \to c \to m\ (a, z)) \to c \to m\ (\mu F, z) \\
& mbuildpF\ g = g\ in_F
\end{aligned}
$$

A first monadic shortcut fusion law can be directly obtained as an instance of Law 4.4.2. We unfold the definition of *mmap* to get a formulation in terms of do-notation:

Law 4.4.4 (FOLD/MBUILDP**)** *For strict k and strictness preserving mmap,*

$$\mathbf{do}\ \{(t, z) \leftarrow mbuildpF\ g\ c; return\ (foldF\ k\ t, z)\} = g\ k\ c$$

This is a version for *mbuildp* of the shortcut fusion law introduced by Manzino and Pardo (Manzino and Pardo 2008), which is associated to a monadic build.

Using this law we can state a first monadic extension of the pfold/buildp rule. Observe that, in the last step of the proof, the introduction of the circularity on z requires the use of a recursive binding within a monadic

computation, expressed in terms of **mdo**-notation (Erkök and Launchbury 2002).

Law 4.4.5 (PFOLD/MBUILDP) *For strict h with components $(h_1, \ldots, h_n)$ and strictness-preserving mmap,*

$$\begin{aligned}
&\textbf{do}\ \{(t, z) \leftarrow \mathit{mbuildpF}\ g\ c; \mathit{return}\ (\mathit{pfold}\ h\ (t, z))\} \\
&= \\
&\textbf{mdo}\{(v, z) \leftarrow \textbf{let}\ k_i\ \bar{x} = h_i\ \bar{x}\ z\ \textbf{in}\ g\ k\ c; \mathit{return}\ v\}
\end{aligned}$$

Proof 4.3

$$\begin{aligned}
&\textbf{do}\ \{(t, z) \leftarrow \mathit{mbuildpF}\ g\ c; \mathit{return}\ (\mathit{pfold}\ h\ (t, z))\} \\
=\quad &\{\ (3.7)\ \} \\
&\textbf{do}\ (t, z) \leftarrow \mathit{mbuildpF}\ g\ c \\
&\quad \textbf{let}\ k_i\ \bar{x} = h_i\ \bar{x}\ z\ \textbf{in}\ \mathit{return}\ (\mathit{foldF}\ k\ t) \\
=\quad &\{\ \text{definition of } \pi_1\ \} \\
&\textbf{do}\ (t, z) \leftarrow \mathit{mbuildpF}\ g\ c \\
&\quad \textbf{let}\ k_i\ \bar{x} = h_i\ \bar{x}\ z\ \textbf{in}\ \mathit{return}\ (\mathit{foldF}\ k\ \$\ \pi_1\ (t, z)) \\
=\quad &\{\ (3.2)\ \} \\
&\textbf{do}\ (t, z) \leftarrow \mathit{mbuildpF}\ g\ c \\
&\quad \textbf{let}\ k_i\ \bar{x} = h_i\ \bar{x}\ z\ \textbf{in}\ \mathit{return}\ (\pi_1\ (\mathit{foldF}\ k\ t, z)) \\
=\quad &\{\ \text{Law 4.4.4 and Law 4.4.3}\ \} \\
&\textbf{mdo}\ (v, z) \leftarrow \textbf{let}\ k_i\ \bar{x} = h_i\ \bar{x}\ z\ \textbf{in}\ g\ k\ c \\
&\qquad \mathit{return}\ v
\end{aligned}$$

When the consumer is also an effectful function, it is possible to state two other laws, similar to Laws 4.4.4 and 4.4.5, respectively, but that deal with fusion of effectful functions. The formulation of these laws follow the approach presented by Chitil (Chitil 2000) and Ghani and Johann (Ghani and Johann 2008).

Law 4.4.6 (EFFECTFUL FOLD/MBUILDP) *For strict* $k :: F\ (m\ a) \to m\ a$ *and strictness preserving mmap,*

$$\begin{aligned}
&\mathbf{do}\ \{(t, z) \leftarrow mbuildpF\ g\ c; v \leftarrow foldF\ k\ t; return\ (v, z)\} \\
=\ &\mathbf{do}\ \{(m, z) \leftarrow g\ k\ c; v \leftarrow m; return\ (v, z)\}
\end{aligned}$$

Proof 4.4

$$\begin{aligned}
&\mathbf{do}\ \{(t, z) \leftarrow mbuildpF\ g\ c; v \leftarrow foldF\ k\ t; return\ (v, z)\} \\
&\quad = \mathbf{do}\ (t, z) \leftarrow mbuildpF\ g\ c \\
&\qquad\quad (m, _) \leftarrow return\ (foldF\ k\ t, z) \\
&\qquad\quad v \leftarrow m \\
&\qquad\quad return\ (v, z) \\
&\quad = \mathbf{do}\ (m, z) \leftarrow \mathbf{do}\ (t, z) \leftarrow mbuildpF\ g\ c \\
&\qquad\qquad\qquad\quad return\ (foldF\ k\ t, z) \\
&\qquad\quad v \leftarrow m \\
&\qquad\quad return\ (v, z) \\
&\quad = \mathbf{do}\ \{(m, z) \leftarrow g\ k\ c; v \leftarrow m; return\ (v, z)\}
\end{aligned}$$

Using this law we can now state a shortcut fusion law for the derivation of monadic circular programs in those cases when both the producer and consumer are effectful functions. Again, like in Law 4.4.5, it is necessary the use of a recursive binding in terms of **mdo**-notation because of the introduction of a circular value within the monadic computation.

Law 4.4.7 (EFFECTFUL PFOLD/MBUILDP) *For strict* $h :: F\ (m\ a, z) \to m\ a$ *with components* $(h_1, \ldots, h_n)$ *and strictness-preserving mmap,*

$$\begin{aligned}
&\mathbf{do}\ \{(t, z) \leftarrow mbuildpF\ g\ c; pfold\ h\ (t, z)\} \\
=\ &\mathbf{mdo}\ (m, z) \leftarrow \mathbf{let}\ k_i\ \bar{x} = h_i\ \bar{x}\ z\ \mathbf{in}\ g\ k\ c \\
&\qquad\quad v \leftarrow m \\
&\qquad\quad return\ v
\end{aligned}$$

Proof 4.5

$$
\begin{aligned}
&\mathbf{do}\ \{(t,z) \leftarrow mbuildpF\ g\ c; v \leftarrow pfold\ k\ (t,z)\}\\
&\quad = \mathbf{do}\ (t,z) \leftarrow mbuildpF\ g\ c\\
&\qquad\quad m \leftarrow return\ (pfold\ k\ (t,z))\\
&\qquad\quad v \leftarrow m\\
&\qquad\quad return\ v\\
&\quad = \mathbf{do}\ (m,z) \leftarrow \mathbf{do}\ (t,z) \leftarrow mbuildpF\ g\ c\\
&\qquad\qquad\qquad\quad return\ (pfold\ k\ (t,z))\\
&\qquad\quad v \leftarrow m\\
&\qquad\quad return\ v\\
&\quad = \mathbf{do}\ m \leftarrow \mathbf{mdo}\ (m,z) \leftarrow \mathbf{let}\ k_i\ \bar{x} = h_i\ \bar{x}\ z\ \mathbf{in}\ g\ k\ c\\
&\qquad\qquad\qquad\quad return\ m\\
&\qquad\quad v \leftarrow m\\
&\qquad\quad return\ v\\
&\quad = \mathbf{mdo}\ (m,z) \leftarrow \mathbf{let}\ k_i\ \bar{x} = h_i\ \bar{x}\ z\ \mathbf{in}\ g\ k\ c\\
&\qquad\quad v \leftarrow m\\
&\qquad\quad return\ v
\end{aligned}
$$

4.5 Conclusions

In this Chapter we have presented rules to calculate monadic circular programs from the composition of monadic functions. The rules presented are generic, as they can be instantiated for several algebraic data types and monads. Our rules are also generally applicable. We have shown two examples that demonstrate their pratical interest: our rules were used to calculate single traversal, deforested programs in the context of monadic parsing and in the context of a programming environment.

These examples, however, consist of a single producer and consumer function composition. In the next Chapter, we show how to generalize our work in order to optimize programs that are defined by an arbitrary number of function compositions of the form $f_n \circ ... \circ f_1$ such that in each composition

a data structure t_i and a value z_i are produced.

Circular programs, monads and attribute grammars are closely related (Swierstra 1993). Indeed, in Chapter 2, attribute grammar techniques are used to model and manipulate circular programs in order to derive efficient non-lazy equivalent programs. We would like to express such transformation in a calculational form.

Chapter 5

Tools and Libraries to Model and Manipulate Circular Programs

5.1 Introduction

In this Chapter, we present the implementation of the techniques formally introduced in Chapter 2 as a *Haskell* library: the *CircLib* library. Using this library, we have constructed two tools 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. In this Chapter, we also 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 Chapter is organized as follows: in Section 5.2 we present the *CircLib* library and the *HaCirc* and *OCirc* tools and in Section 5.3 we present the results of the benchmarks we have conducted. Section reftool:conclusions concludes the Chapter.

5.2 Tools and Libraries for Circular Programming

5.2.1 The *CircLib* Library

The *CircLib* library is a library written in *Haskell* and that manipulate circular programs (its API is given in appendix .1). 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.

5.2.2 The *HaCirc* Tool

The *HaCirc* tool is an Haskell refactor. It refactors circular programs into its strict counterparts. The tool accepts, as input, *Haskell* circular programs and produces, as output, strict *Haskell* programs. Furthermore, it is also possible to obtain strict programs that use no explicit intermediate data structure.

HaCirc is also slicer of circular programs. Indeed, the tool is able to compute circular programs' slices, which can be obtained in two different programming styles: as multiple traversal strict programs that use intermediate data structures and as deforested programs (i.e., programs with no intermediate, traversal gluing, structures).

5.2.3 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;

- a web-based interactive tool(s) that allows the tool(s) to be used on-line[1]. The execution of such interactive version of the tool(s) requires no further instalation.

  We will describe a possible interaction with this interface by running an example and presenting the output produced.

  When loading the interface, the user is shown following web page presented in Figure 5.1.

  The user may then introduce a circular program in the white text box constructed or just select, by clicking the corresponding button, one of the built-in circular programs. In the later case, the interactive interface will then automatically fill the text box with the selected program. Furthermore, the interface will allow the user to select the type program the circular program is to be transformed into.

  If we select the *repmin* circular program and transform it into it's deforested equivalent, we obtain the program presented in Figure 5.2.

The reader may also use one of the two versions of *HaCirc* and *OCirc* to produce, for example, the *Ocaml* or *Haskell* strict programs of the *repmin* program and the *Table* processor from the circular definitions presented in this paper (and available in both tool versions, as we have seen for *repmin* and for the Web version of *HaCirc*).

The slicing of circular programs can also be performed using either one of the tool versions.

[1]The tools are available online at `http://www.di.uminho.pt/~jpaulo`

Figure 5.1: Web interactive interface of the *HaCirc* tool.

5.3 Benchmarks

In order to benchmark the different implementations of circular programs, we conducted several experiments. In this Section, we show the results of two of them: we compared the running performance of the circular *Table* formatter with the running performances of the programs we derived from it, *i.e.*, the

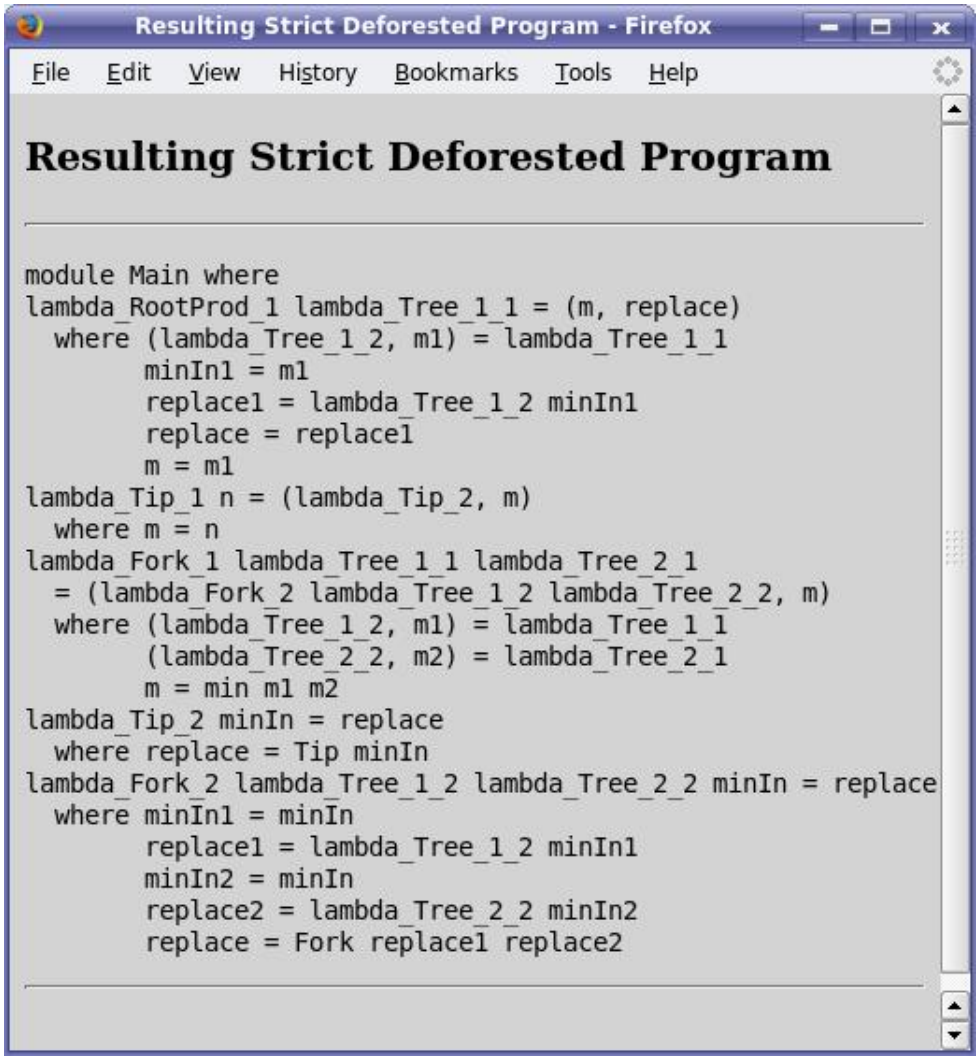

Figure 5.2: The deforested version of the *repmin* program.

strict multiple traversal and the higher-order deforested equivalents; furthermore, we have also compared the performance of a circular program that processes a tiny subset of the *C* language, called *MicroC* with its equivalent derived strict and deforested programs. The *Table* circular program induces a simple two traversal strict program, while the *MicroC* circular program induces a *six* traversal program.

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.

5.3.1 The Table Formatter:

The three *Table* formatters presented earlier 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 5.1.

		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 5.1: 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.

5.3.2 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 (Swierstra et al. 1999). 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.

	Input size	Circular Mem (Kb)	Circular Time (sec)	Strict Mem (Kb)	Strict Time (sec)	Deforested Mem (Kb)	Deforested 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 5.2: 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 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.

5.4 Conclusions

The techniques presented in Chapter 2 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 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. Finally, the first experimental results show that the strict deforested

Haskell programs are more efficient than the *Haskell* lazy circular programs.

Chapter 6

Conclusions

This thesis discussed the design, implementation and calculation of circular programs. In Chapter 2, we have 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.

In Chapter 3 we have presented a new program transformation technique for intermediate structure elimination. The programs we are able of dealing with consist in the composition of a producer and a consumer functions. The producer constructs an intermediate structure that is later traversed by the consumer. Furthermore, we allow the producer to compute additional values that may be needed by the consumer. This kind of compositions is general enough to deal with a wide number of practical examples. Our approach is calculational, and proceeds in two steps: we apply standard deforestation methods to obtain intermediate structure-free programs and we introduce circular definitions to avoid multiple traversals that are introduced by deforestation. Since that, in the first step, we apply standard fusion techniques, the expressive power of our rule is then bound by deforestation.

We introduce a new calculational rule conceived using a similar approach to the one used in the *fold/build* rule: our rule is also based on parametricity properties of the functions involved. Therefore, it has the same benefits

and drawbacks of *fold/build* since it assumes that the functions involved are instances of specific program schemes. Therefore, it could be used, like *fold/build*, in the context of a compiler. In fact, we have used the rewrite rules (RULES pragma) of the Glasgow Haskell Compiler (GHC) in order to obtain a prototype implementation of our fusion rule.

The rule that we propose is easy to apply: in this thesis, we have presented a real example that shows that our rule is effective in its aim. Other examples may be found in (Fernandes et al. 2007). The calculation of circular programs may be understood as an intermediate stage: the circular programs we calculate may be further transformed into very efficient, completely data structure free programs.

In Chapter 4 we have presented rules to calculate monadic circular programs from the composition of monadic functions. The rules presented are generic, as they can be instantiated for several algebraic data types and monads. Our rules are also generally applicable. We have shown two examples that demonstrate their pratical interest: our rules were used to calculate single traversal, deforested programs in the context of monadic parsing and in the context of a programming environment.

These examples, however, consist of a single producer and consumer function composition. In the next Chapter, we show how to generalize our work in order to optimize programs that are defined by an arbitrary number of function compositions of the form $f_n \circ ... \circ f_1$ such that in each composition a data structure t_i and a value z_i are produced.

Circular programs, monads and attribute grammars are closely related (Swierstra 1993). Indeed, in Chapter 2, attribute grammar techniques are used to model and manipulate circular programs in order to derive efficient non-lazy equivalent programs. We would like to express such transformation in a calculational form.

The techniques presented in Chapter 2 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 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. Finally, the first experimental results show that the strict deforested *Haskell* programs are more efficient than the *Haskell* lazy circular programs. The *CircLib* library, the *HaCirc* and *OCirc* tools and the benchmark results were presented in Chapter 5.

Bibliography

S. Abramsky and A. Jung. Domain theory. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 3, pages 1–168. Clarendon Press, 1994.

R. Bird. *Introduction to Functional Programming using Haskell,* 2nd edition. Prentice-Hall, UK, 1998.

Richard Bird and Oege de Moor. *Algebra of Programming*, volume 100 of *Prentice-Hall Inernational Series in Computer Science*. Prentice-Hall, 1997.

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

O. Chitil. *Type-inference based deforestation of functional programs.* PhD thesis, RWTH Aachen, October 2000.

R. Cockett and T. Fukushima. About Charity. Technical Report 92/480/18, University of Calgary, June 1992.

R. Cockett and D. Spencer. Strong Categorical Datatypes I. In R.A.C. Seely, editor, *International Meeting on Category Theory 1991,* volume 13 of *Canadian Mathematical Society Conference Proceedings*, pages 141–169, 1991.

Nils Anders Danielsson, John Hughes, Patrik Jansson, and Jeremy Gibbons. Fast and loose reasoning is morally correct. In *POPL '06: Conference*

record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pages 206–217, New York, NY, USA, 2006. ACM.

Olivier Danvy and Mayer Goldberg. There and back again. In *ICFP '02: Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, pages 230–234, New York, NY, USA, 2002. ACM Press. ISBN 1-58113-487-8. doi: http://doi.acm.org/10.1145/581478.581500.

Oege de Moor, Kevin Backhouse, and S. Doaitse Swierstra. First-class attribute grammars. *Informatica (Slovenia)*, 24(3), 2000. URL `citeseer.ist.psu.edu/demoor00firstclass.html`.

Oege de Moor, Simon Peyton-Jones, and Eric Van Wyk. Aspect-oriented compilers. *Lecture Notes in Computer Science*, 1799, 2000. URL `citeseer.ist.psu.edu/demoor99aspectoriented.html`.

Atze Dijkstra. *Stepping through Haskell*. PhD thesis, Department of Computer Science, Utrecht University, The Netherlands, November 2005.

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

Joost Engelfriet and Gilberto Filé. Simple multi-visit Attribute Grammars. *Journal of Computer and System Sciences*, 24(3):283–314, 1982.

L. Erkök and J. Launchbury. A Recursive do for Haskell. In *Haskell '02: Proceedings of the ACM SIGPLAN Haskell Workshop*, pages 29–37. ACM, 2002.

João Paulo Fernandes, Alberto Pardo, and João Saraiva. A shortcut fusion rule for circular program calculation. In *Haskell '07: Proceedings of the ACM SIGPLAN Haskell workshop*, pages 95–106, New York, NY, USA, 2007. ACM.

N. Ghani and P. Johann. Short Cut Fusion of Recursive Programs with Computational Effects. In *Symposium on Trends in Functional Programming (TFP 2008)*, 2008.

J. Gibbons. Calculating Functional Programs. In *Algebraic and Coalgebraic Methods in the Mathematics of Program Construction,* LNCS 2297, pages 148–203. Springer-Verlag, January 2002.

A. Gill. *Cheap Deforestation for Non-strict Functional Languages*. PhD thesis, Department of Computing Science, University of Glasgow, UK, 1996.

Andrew Gill, John Launchbury, and Simon L. Peyton Jones. A short cut to deforestation. In *Conference on Functional Programming Languages and Computer Architecture*, pages 223–232, June 1993.

Dennis Kugler Harald Baier and Marian Margraf. Elliptic Curve Cryptography Based on ISO 15946. Technical Report TR-03111, Federal Office for Information Security, 2007.

Ralf Hinze and Johan Jeuring. Generic Haskell: Practice and theory. In *Summer School on Generic Programming*, 2002. URL `http://www.cs.uu.nl/~johanj/publications/GH.pdf`.

Susan Horwits and Thomas 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.

G. Hutton and E. Meijer. Monadic Parsing in Haskell. *Journal of Functional Programming*, 8(4):437–444, July 1998.

Patricia Johann and Janis Voigtländer. Free theorems in the presence of seq. In *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 99–110, New York, NY, USA, 2004. ACM.

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

Neil Jones, Carsten Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall Inernational Series in Computer Science. Prentice-Hall, 1993.

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

Donald 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).

Matthijs Kuiper and Doaitse Swierstra. Using attribute grammars to derive efficient functional programs. In *Computing Science in the Netherlands CSN'87*, November 1987.

John Launchbury and Tim Sheard. Warm fusion: Deriving build-catas from recursive definitions. In *Conf. Record 7th ACM SIGPLAN/SIGARCH Int. Conf. on Functional Programming Languages and Computer Architecture, FPCA'95, La Jolla, San Diego, CA, USA, 25–28 June 1995*, pages 314–323. ACM Press, New York, 1995.

Julia L. Lawall. Implementing Circularity Using Partial Evaluation. In *Proceedings of the Second Symposium on Programs as Data Objects PADO II*, volume 2053 of *LNCS*. Springer-Verlag, May 2001.

C. Manzino and A. Pardo. Short Cut Fusion of Monadic Programs. In *Brazilian Symposium on Programming Languages (SBLP 2008)*, 2008.

Simon Marlow and Simon Peyton Jones. The new GHC/Hugs Runtime System. URL `http://research.microsoft.com/Users/simonpj/Papers/new-rts.ps.gz`. 1999.

Atsushi Ohori and Isao Sasano. Lightweight fusion by fixed point promotion. In *POPL '07: Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 143–154, New York, NY, USA, 2007. ACM Press.

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

Y. Onoue, Z. Hu, H. Iwasaki, and M. Takeichi. A Calculational Fusion System HYLO. In *IFIP TC 2 Working Conference on Algorithmic Languages and Calculi, Le Bischenberg, France*, pages 76–106. Chapman & Hall, February 1997a.

Yoshiyuki Onoue, Zhenjiang Hu, Hideya Iwasaki, and Masato Takeichi. A calculational fusion system HYLO. In *Algorithmic Languages and Calculi*, pages 76–106, 1997b. URL `citeseer.ist.psu.edu/onoue97calculational.html`.

A. Pardo. Generic Accumulations. In *IFIP WG2.1 Working Conference on Generic Programming*, Dagstuhl, Germany, July 2002.

A. Pardo. *A Calculational Approach to Recursive Programs with Effects*. PhD thesis, Technische Universität Darmstadt, October 2001.

Maarten Pennings. *Generating Incremental Evaluators*. PhD thesis, Department of Computer Science, Utrecht University, The Netherlands, November 1994.

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

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

João Saraiva and Doaitse 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.

Doaitse Swierstra. Tutorial on attribute grammars. In *Generative Programming and Component Engineering*, 1993.

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.

Doaitse Swierstra, Pablo Azero, and João Saraiva. Designing and Implementing Combinator Languages. In Doaitse Swierstra, Pedro Henriques, and José Oliveira, editors, *Third Summer School on Advanced Functional Programming*, volume 1608 of *LNCS Tutorial*, pages 150–206. Springer-Verlag, September 1999.

S. Doaitse Swierstra and Pablo Azero. Attribute grammars in a functional style. In *Systems Implementation 2000*, Berlin, 1998. Chapman & Hall.

A. Takano and E. Meijer. Shortcut to Deforestation in Calculational Form. In *Functional Programming Languages and Computer Architecture'95*, 1995.

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

Janis Voigtländer. Semantics and pragmatics of new shortcut fusion rules. In *FLOPS '08: Proceedings of the 2008 International Symposium on Functional and Logic Programming*, pages 163–179. Springer-Verlag, 2008.

Janis Voigtländer. Using circular programs to deforest in accumulating parameters. *Higher-Order and Symbolic Computation*, 17:129–163, 2004. Previous version appeared in *ASIA-PEPM 2002*, Proceedings, pages 126–137, ACM Press, 2002.

P. Wadler. Theorems for free! In *4th International Conference on Functional Programming and Computer Architecture*, London, 1989.

P. Wadler. Deforestation: transforming programs to eliminate trees. *Theoretical Computer Science*, 73:231–248, 1990.

William Waite, Uwe Kastens, and Anthony M. Sloane. *Generating Software from Specifications*. Jones and Bartlett Publishers, Inc., USA, 2007. ISBN 0763741248.

.1 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[1]:

$$\begin{aligned}
&\textbf{data } CP = CP\{\, constrs &&:: [Constr],\\
&\qquad\qquad\quad types &&:: [DT],\\
&\qquad\qquad\quad prods &&:: Map\ Constr\ [DT],\\
&\qquad\qquad\quad args &&:: Map\ DT\ [VarName],\\
&\qquad\qquad\quad results &&:: Map\ DT\ [VarName],\\
&\qquad\qquad\quad deps &&:: Map\ Constr\ [Dep]\\
&\qquad\qquad\quad semantics &&:: Map\ Constr\ (Map\ VarName\ Function)\}
\end{aligned}$$

$$\textbf{type } Var = (Constr, Int, String)$$

$$\textbf{type } Dep = ((Int, Name), (Int, Name))$$

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

$$\begin{aligned}
dp &:: CP \rightarrow Rel\ Var\ Var\\
idp &:: CP \rightarrow Rel\ Var\ Var\\
ids &:: CP \rightarrow Rel\ (DT, Name)\ (DT, Name)\\
a &:: CP \rightarrow DT \rightarrow Int \rightarrow Set\ (DT, Name)\\
ds &:: CP \rightarrow DT \rightarrow Rel\ (DT, Name)\ (DT, Name)\\
edp &:: CP \rightarrow Rel\ Var\ Var\\
isOrdered &:: CP \rightarrow Bool\\
interface &:: CP \rightarrow DT \rightarrow Interface
\end{aligned}$$

[1] These functions correspond to the *Haskell* versions of the formal definitions presented in Section 2.3.3.

$$\textbf{type}\ Interface = [(Set\ (DT, Name), Set\ (DT, Name))]$$

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

$$\textbf{data}\ VisitSequences = VS\ (Map\ Constr\ [\,VisitSubSequence\,])$$

$$\begin{aligned}\textbf{data}\ VisitSubSequence = VSS\{\,& n \quad :: Int,\\ & prod :: [\,DT\,],\\ & arg \ :: [\,VarName\,],\\ & res \ :: [\,VarName\,],\\ & instructions :: [\,Instruction\,]\}\end{aligned}$$

$$\begin{aligned}\textbf{data}\ Instruction = Eval\,\{\,& variable :: Var,\\ & uses \quad :: [\,Var\,]\}\\ \mid\ Visit\{\,& visit \quad :: (Int, Int),\\ & inp \quad :: [\,Name\,],\\ & out :: [\,Name\,]\}\end{aligned}$$

$$visit\_sequences :: CP \ \rightarrow\ VisitSequences$$

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

$$backward\_slice :: CP \ \rightarrow\ Criteria \ \rightarrow\ VisitSequences$$

$$forward\_slice \ :: CP \ \rightarrow\ Criteria \ \rightarrow\ VisitSequences$$

$$\textbf{type}\ Criteria = [\,VarName\,]$$