Design, Implementation and Calculation of Circular Programs

João Paulo Fernandes

July 9, 2009

Acknowledgements

Several people contributed to the work presented in this thesis in different ways.

First of all, I would like to express my infinite gratitude to João Saraiva. For more than 4 years now, João Saraiva has been much more than just an advisor. Ever since the beginning of this project, he has always believed in me and in the project itself. His confidence was very important to me, especially in the difficult moments. João Saraiva understands research (and life itself) in ways that I truly admire. Also when not researching, João Saraiva has always been a true friend. You always shared your vision of things with me, while still letting me make my own decisions. I owe you a lot!

Oege de Moor kindly accepted to *keep an eye* on my work. He also provided me the unique experience of living and working at Oxford. I have enjoyed my stays there very much and I learned a lot from working in such an excellency environment. My sincere thanks for that. I would also like to thank Oege's students in the basement. A special thanks goes also to Bruno Oliveira, who made my Oxford stays even more memorable.

Alberto Pardo played a crucial role in this project, which he entered half-way through. Alberto is not only an exceptional researcher, but also a very special person, that I truly admire and that I am very happy to have as a friend. I will never forget the warm way he received me in Montevideo. I had a wonderful time there and I look forward to going back. My stay in Montevideo was made even more memorable by Alberto's colleagues and students. I would like to thank Marcos Viera and Daniel Calegari in a special way.

I would also like to thank Janis Voigtländer and Alberto Pardo for incredibly accurate and very useful comments on the work presented in this thesis.

During this project, I had the opportunity to work in fantastic environments. I will always treasure the memories of working in the best office ever: *lab 1.08*, that later moved to *lab 1.06*. Alexandra Silva, Jácome Cunha, José Proença, Miguel Vilaça, Paulo Silva and Tiago Alves: you were the best office mates I could ever want. A special word goes to Miguel Vilaça, long time friend and office mate. Our professional paths have been similar for a while now. I sincerely wish you all the best in life. I would also like to thank all the unnamed members of the glorious *OsSemEstatuto* group. Beach volleyball in the afternoon is, I see it now, priceless.

Nuno Rodrigues, João Fernando and, again, Alexandra Silva and Miguel Vilaça joined me in unforgettable summer schools and *side-effect* travels. Thank you enormously for that. Alexandra Silva deserves a special note for being not only the most intelligent person I know, but also for being a true friend that deserves all the best.

I would also like to thank Alberto Mendes, Daniel Mendes, Filipe Pereira, Miguel Eiriz and José Álvaro for a memorable undergraduate period. My gratitude goes also to Marta Santos, whose friendship lasts ever since high-school days.

These acknowledgments would no be complete without thanking my family, the most important pillar of my life.

Pedro Jorge, my godchild, is an endless source of joy. Whenever we are together, we always share great moments. You are a very special kid, and I thank you for being so genuine.

Alice and Manuel Pinto, and Ana and Pedro Pinto have always been my *borrowed* parents, sister and brother. I know that they will always be there, come what may. Our bond is remarkable, and I am in great debt to you all for that.

My grandparents Emília and António Sousa always took care of their grandchildren in an impressive way. They have always made me feel like I was special and I am very grateful for that. You will always be the best. I also want to thank my grandfather José Fernandes, that I wish I could have known better.

I would especially like to thank my parents, Angelina and Joaquim Fernandes. They have always provided me the best education I could ever dream of. I am truly inspired by the way you live your life. Thank you for your love, for your dedication and for your hard-work in making everything possible. You deserve the biggest credit for the work presented in this thesis.

My brother Rui believes the most in my abilities. He proudly speaks of me in terms that I clearly do not deserve. Thank you for always being there, for your courage and example. I am sure that life will grant you everything you deserve.

This project had just began when I met my girlfriend Adélia. I have several times tried to describe Adélia, and what she means to me, in words. Even though I fail time after time, I shall try again. Adélia amazes me. She fills my horizon with joy and happiness. We are so close to each other that I sometimes feel myself living through her. You know best that this project had its difficult moments. Thank you for always believing in me, even when I did not. I hope one day you will accept becoming my wife and being the mother of my children.

Ao avô António, cuja sabedoria seria suficiente para escrever várias teses de doutoramento, dedico esta tese.

Several institutions also contributed to this thesis in important 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 2007 and 2009 ACM SIGPLAN Symposiums 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 in the ACM SIGPLAN 2009 Symposium on Partial Evaluation and Program Manipulation. The NATO Science for Peace and Security Programme provided financial support for me to participate in the Summer School Marktoberdorf 2007.

Contents

Chapter 1

Introduction

Summary

In this chapter, we present a small introduction to the notation and concepts that will be studied throughout this thesis. We introduce the technique of circular programming by reviewing the repmin problem. We also consider different approaches to solve repmin: we define a naive functional program and transform it into a circular equivalent that is later transformed into a higher-order program. We also present the definition of repmin in terms of an attribute grammar.

In a purely functional setting, programs are often constructed as a set of simple and clear components, which are *glued* together by using intermediate data structures. Compilers are a typical example of programs designed in this way: a function, the *parser*, constructs a syntax tree that is later decorated by another function.

There are significant advantages in structuring our programs in this way. Considered in isolation, each function, or traversal, may be relatively simple. Consequently, programs so defined are easier to write, to understand and are potentially more reusable. By separating many distinct phases, it becomes possible to focus on a single task, rather than attempting to do many things at the same time. Furthermore, there are algorithms that rely on a multiple traversal strategy because *context information* must first be collected before it can be used. That is, information flows from one traversal to a following one. Functional programmers usually define intermediate data structures to convey the information computed in one traversal and used in the following one. Such intermediate data structures glue the different traversals. The construction, traversal and destruction of a (potentially) large number of intermediate data structures, however, may degrade the efficiency of the total program, both in terms of runtime and memory consumption.

An alternative way to write multiple traversal programs in a lazy functional setting, avoiding the definition of all redundant intermediate data structures, is to use circular programming. Circular programs were first 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. 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 that we want to transform 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. An example of this transformation is given in Figure 1.1.

In order to implement *repmin*, we must start by defining a representation for binary leaf trees. For this purpose, we may use the following data-type definition, that is written in the *Haskell* programming language (Peyton Jones et al. 1999; Peyton Jones 2003):

```
data LeafTree = Leaf Int
              | Fork (LeafTree, LeafTree)
```

A binary leaf tree will then be represented by an element of the *LeafTree* data-type. Such an element is either a leaf that holds an integer value (built using the data-type constructor *Leaf* :: *Int* → *LeafTree*) or the fork of two

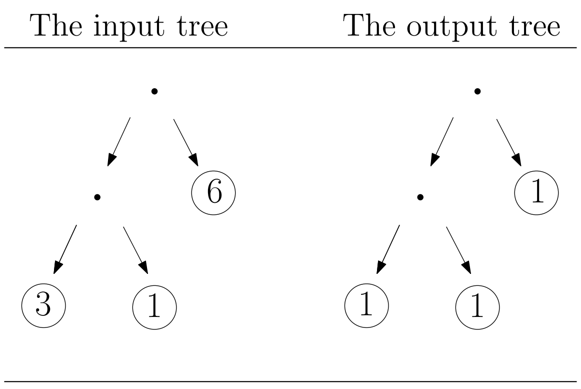


Figure 1.1: An example of the use of *repmin*

leaf trees (such an element is built using the constructor function $Fork :: (LeafTree, LeafTree) \rightarrow LeafTree$). The input tree presented in Figure 1.1, for example, is represented by the following *LeafTree* element:

```
tree :: LeafTree
tree = Fork (Fork (Leaf 3,
                   Leaf 1),
             Leaf 6)
```

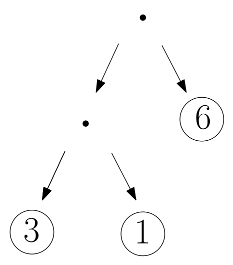
Having defined a representation for leaf trees, we may now consider different implementations for solving *repmin*. In a strict and purely functional setting, for example, solving this problem would require a two traversal strategy. First, we need to traverse the input tree in order to compute its minimum value. This computation is performed by the function *tmin*, that we present next[1].

```
tmin :: LeafTree → Int
tmin (Leaf n)       = n
tmin (Fork (l, r)) = min (tmin l) (tmin r)
```

This function has been given the type signature $tmin :: LeafTree \rightarrow Int$. This means that function *tmin* receives as input a leaf tree, and produces as result an integer value (which will be the minimum value of the input tree).

[1]The function $min :: Int \rightarrow Int \rightarrow Int$ computes the minimum of two numbers.

For example, for the input tree considered in Figure 1.1,



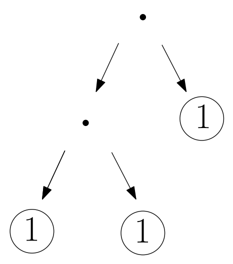
function *tmin* would produce the integer value 1 as result, since 1 is indeed the minimum value of this tree.

Having traversed the input tree to compute its minimum value, we need to traverse that tree again. We need to replace all its leaf values by the minimum value, therefore producing the desired tree result. This is implemented in function *replace*, as follows:

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

The symbol _, that is used in the definition of function *replace*, stands for *any value*. In this function, it is used to express that, no matter what value a leaf has in the original tree, it is going to be replaced by the minimum value computed for that tree.

If we now call function *replace* with the input tree considered before and with the integer value 1 as its arguments, function *replace* produces, as expected, the tree result:



Having implemented functions *replace* and *tmin*, we now only need, in order to solve *repmin*, to combine them appropriately:

$$\begin{aligned}
&\mathit{transform} :: \mathit{LeafTree} \rightarrow \mathit{LeafTree}\\
&\mathit{transform}\ t = \mathit{replace}\ (t, \mathit{tmin}\ t)
\end{aligned}$$

As we have noticed, this solution to *repmin* traverses any input tree twice. The original tree serves as the intermediate data structure that glues the two traversals together. 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. Bird used the following well known techniques: tupling, fold-unfold and local recursion[2].

$$\begin{aligned}
&\mathit{repmin}\ (\mathit{Leaf}\ n, m) = (\mathit{Leaf}\ m, n)\\
&\mathit{repmin}\ (\mathit{Fork}\ (l, r), m) = (\mathit{Fork}\ (t_1, t_2), \mathit{min}\ m_1\ m_2)\\
&\quad \mathbf{where}\ (t_1, m_1) = \mathit{repmin}\ (l, m)\\
&\qquad\qquad\ (t_2, m_2) = \mathit{repmin}\ (r, m)
\end{aligned}$$

$$\begin{aligned}
&\mathit{transform}\ t = \mathit{nt}\\
&\quad \mathbf{where}\ (\mathit{nt}, \boxed{\mathrm{m}}) = \mathit{repmin}\ (t, \boxed{\mathrm{m}})
\end{aligned}$$

Notice the circularity in the above program: m is both an argument and a result of the *repmin* call, in the *transform* function[3]. 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 this circular definition.

Bird's work showed the power of circular programming, not only as an optimization technique to eliminate multiple traversals 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 scheduling of the different traversal functions, since only a single (traversal) function has to be defined. Neither does the programmer have to define intermediate

[2]We will review Bird's transformation in detail in chapter 2.

[3]In order to make it easier for the reader to identify circular definitions, we frame the occurrences of variables that induce them (m in this case).

gluing data structures to convey values computed in one traversal and needed in following ones because there is a single traversal function only.

Due to their nice properties, circular programs are a widely used technique. In fact, circular programs have been studied in varied contexts:

- in the construction of *Haskell* compilers (Marlow and Peyton Jones 1999; Hinze and Jeuring 2002);
- in the implementation of aspect-oriented compilers (Moor et al. 2000);
- to express pretty printing algorithms (Swierstra et al. 1999; Swierstra and Chitil 2009);
- to implement breadth-first traversal strategies (Okasaki 2000);
- to express type systems (Dijkstra and Swierstra 2004);
- as an optimization technique in the deforestation of accumulating parameters (Voigtländer 2004);
- in the implementation of programming environments (Michiel 2004);
- they have been studied in the context of partial evaluation (Lawall 2001) and continuations (Danvy and Goldberg 2002);
- they are used in bidirectional transformations (Mu et al. 2005).

Circular programs are also strongly related to attribute grammars (Knuth 1968; Paakki 1995). Indeed, as Johnsson (1987) and 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). This means that we can obtain a single traversal *repmin* solution not only by applying Bird's techniques to the straightforward two traversal solution of *repmin*, that we presented on page 5. Indeed, if we express *repmin* in terms of an attribute grammar for its input trees, it is possible to transform it into a single traversal solution, using the techniques presented by Kuiper and Swierstra (1987). In that paper, the authors also present the attribute grammar for solving the *repmin* problem, that we review next.

The construction of an attribute grammar for *repmin* attaches the arguments and results of functions *tmin* and *replace* to the nodes of the input *LeafTree*, as presented next.

```
Leaf : LeafTree → n :: Int
            { LeafTree . tmin := n;
              LeafTree . replace := Leaf (LeafTree . m)}

Fork : LeafTree → (l :: LeafTree, r :: LeafTree)
            { LeafTree . tmin := min (l . tmin) (r . tmin);
              LeafTree . replace := Fork (l . replace, r . replace);
              l . m := LeafTree . m;
              r . m := LeafTree . m}

Repmin : R → t :: LeafTree
            { R . replace := t . replace;
              t . m := t . tmin}
```

An attribute grammar is a context-free grammar augmented with semantic rules. The context-free grammar corresponds to the algebraic structure defined in the functional solution, where every non-terminal in the grammar corresponds to a data-type in the functional program. Thus, the data-type *LeafTree* of the *repmin* circular program is a non-terminal of the *repmin* attribute grammar. Furthermore, every constructor in the functional program corresponds to a production in the attribute grammar: this is the case of constructors *Leaf* and *Fork* of the *repmin* program, which consist of productions of the *repmin* attribute grammar. We also add to the attribute grammar a root non-terminal symbol and the corresponding production: notice that non-terminal *Repmin* has been added to the *repmin* attribute grammar, and a production has been defined for that non-terminal symbol. This corresponds to adding the following data-type definition to the *repmin* functional program.

data $R = \mathit{Repmin}\ \mathit{LeafTree}$

The attribute grammar then associates a fixed set of attributes to each non-terminal symbol in the grammar (we annotate the attribute a of non-terminal T as $T.a$). Attributes may be inherited or synthesized: function arguments, such as m, correspond to inherited attributes and function results, such as *tmin* and *replace* correspond to synthesized attributes[4]. For each production of the grammar, its semantic rules specify the values of the synthesized attributes of the left hand side nonterminal and the inherited ones for the nonterminals of the right hand side of the production rules. Inherited attributes of the left hand side non-terminal, and inherited and synthesized attributes of the right hand side non-terminal can be used as arguments of the semantic rules. In the *repmin* attribute grammar presented before, semantic rules are written between curly braces and immediately follow the corresponding context free production.

In (Kuiper and Swierstra 1987), the authors define a straightforward mapping between attribute grammars and single traversal circular programs. The program we obtain by applying such techniques to the previously presented attribute grammar is exactly the same circular program that we obtain by using Bird's technique on the straightforward *repmin* solution. That is to say that the *repmin* attribute grammar is implemented by the *repmin* circular program of page 5. In fact, circular programs are a simple and natural implementation of attribute grammars in a lazy language. Modern attribute grammar systems, like LRC (Kuiper and Saraiva 1998), Silver (Wyk et al. 2006) or UUAG (Swierstra et al. 2004) use this approach to implement attribute grammars in *Haskell*.

Deriving circular programs, however, is not the only way to eliminate multiple traversals of data structures. In particular, the straightforward *repmin* solution of page 5 may be transformed, by the application of a well-known technique called lambda-abstraction (Pettorossi and Skowron 1987), into a

[4]By definition, the root non-terminal symbol of an attribute grammar does not have inherited attributes. This is the reason why we added the non-terminal *Repmin* and its production to the *repmin* attribute grammar.

higher-order program. As a result, we obtain[5]:

$$
\begin{aligned}
&\mathit{repmin}\ (\mathit{Leaf}\ n) = (\lambda m \rightarrow \mathit{Leaf}\ m, n)\\
&\mathit{repmin}\ (\mathit{Fork}\ (l, r)) = (\lambda m \rightarrow \mathit{Fork}\ (t_1\ m, t_2\ m), \mathit{min}\ m_1\ m_2)\\
&\quad \mathbf{where}\ (t_1, m_1) = \mathit{repmin}\ l\\
&\quad \phantom{\mathbf{where}}\ (t_2, m_2) = \mathit{repmin}\ r
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{transform}\ t = f\ m\\
&\quad \mathbf{where}\ (f, m) = \mathit{repmin}\ t
\end{aligned}
$$

Regarding this new version of *repmin*, we may notice that it is a higher-order program, since f, the first component of the result produced by the call *repmin* t, is a function. Later, f is applied to m, the second component of the result produced by that same call, therefore producing the desired tree result. Thus, this version does not perform multiple traversals. Furthermore, it does not use any intermediate data structure: instead it constructs an intermediate tree of function calls. This higher-order *repmin* solution is equivalent to both the *repmin* solutions presented so far: the straightforward solution, and the circular solution derived from it.

This thesis studies methods to model, analyze, manipulate and calculate circular programs. In particular,

(i) we introduce new program calculation techniques to transform strict multiple traversal programs into circular ones. Our methods are presented in the style of shortcut fusion, under generic calculation rules, that can be instantiated for a wide class of programs. Furthermore, the rules that we present are studied both in the context of pure and monadic/effectful programming. Post-calculation optimizations, trading circularities for higher-order definitions, are also exploited and presented.

(ii) we use a different approach to perform the transformation in the opposite direction. Indeed, we introduce a new program manipulation tech-

[5]In the program, we use two anonymous functions that are defined using the symbol λ. Defining $\lambda m \rightarrow \mathit{Leaf}\ m$, for example, is equivalent to defining $g\ m = \mathit{Leaf}\ m$.

nique based on attribute grammars that transforms circular programs into strict multiple traversal ones. The execution of the derived programs is not restricted to a lazy evaluation setting as they do not rely on circular definitions, but on intermediate data structures to glue different traversals together. A post-transformation optimization, based on program specialisation, is also implemented: the intermediate data structures are deforested and we obtain programs consisting of a set of higher-order functions. Furthermore, we exploit the breaking up of circularities in the original programs to perform slicing on circular programs.

(iii) we present tools that implement our circular program manipulation techniques. The developed tools can be used in the construction of an attribute grammar system and they can also be used to transform circular programs into strict and strict deforested programs. Furthermore, we also present a systematic benchmark, comparing the execution of the different types of programs we manipulate and calculate, in this thesis.

The following diagram summarizes the results that we present in this thesis.

$$\begin{tikzcd}
& \textit{circular programs} \arrow[dl, "a)"'] \arrow[dr, "b)"] & \\
\textit{strict programs} \arrow[ur, dashed, shift right=2, "c)"'] \arrow[rr, dashed, "d)"] & & \textit{deforested programs}
\end{tikzcd}$$

The arrows *a*) and *b*) correspond to the attribute grammar based program manipulation techniques described in (ii) and that we have presented in (Fernandes and Saraiva 2007). The arrows *c*) and *d*) correspond to the calculational methods described in (i) and that we have presented, for pure programs, in (Fernandes et al. 2007) and, for monadic programs, in (Pardo et al. 2008).

1.1 Contributions

The main contributions of this thesis are:

- a generic program calculation rule, in the style of shortcut deforestation, to obtain circular programs from strict ones;
- a generic calculation rule, similar to the above one, but that introduces higher-order definitions instead of circular ones;
- the formal proof that such rules are correct;
- the study of program calculation rules, developed in the same setting as the above ones, but in the monadic context;
- the monadic rules were also proved correct;
- a strictification technique, based on well-known attribute grammar techniques, that we have developed and applied to transform circular programs into strict and strict deforested programs;
- a systematic benchmark comparing the performances of circular, strict and strict deforested programs;
- a set of tools and a reusable library that can be used to model and manipulate circular programs.

1.2 Structure of the thesis

This thesis is organized as follows. In chapter 2 we present a shortcut deforestation technique to calculate circular programs. The technique we propose takes as input the composition of two functions: the first builds an intermediate data structure and some additional context information which are then processed by the second one, to produce the final result. Our transformation, proposed under a generic calculation rule, achieves intermediate structure deforestation and multiple traversal elimination. Moreover, the calculated programs preserve the termination properties of the original ones.

We also study the implementation of the rule we propose in the context of an optimizing compiler.

In chapter 3, we propose an extension to the new form of fusion presented in chapter 2, but in the context of monadic programming. The 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 4, we study an alternative transformation such that, instead of circular programs, higher-order programs are derived. The alternative transformation is studied in the context of both pure and monadic programs, and it is also developed under a calculational setting.

Chapter 5 presents techniques to model circular lazy programs in a strict, purely functional setting. Circular programs are first modeled in a functional language, and they are then transformed, by applying attribute grammar based techniques, into strict and strict and deforested equivalent programs. We also propose techniques to identify sub-programs of circular programs, using standard slicing techniques.

All the techniques proposed in chapters 2, 3, 4 and 5 are applied to motivational examples that we introduce in each chapter.

In chapter 6, we present the implementation of the techniques formally introduced in chapter 5 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. Based on *CircLib*, we have also developed a simple attribute grammar system. Furthermore, we also present the results we obtained in the first systematic benchmarking comparing the performances of circular, strict and higher-order programs.

Finally, in chapter 7, we draw the conclusions concerning the work presented in the thesis and suggest directions for future work.

Chapter 2

Calculation of Circular Programs

Summary

In this chapter, we present shortcut deforestation techniques to calculate circular programs. The techniques we propose take 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 transformations into circular programs achieve intermediate data structure deforestation and multiple traversal elimination. Furthermore, the calculated programs preserve the termination properties of the original ones.

2.1 Introduction

Circular programs 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 scheduling of the different traversals and does not have to define intermediate gluing data structures.

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 construct an intermediate structure different from the input one. Indeed, programs are often defined as the composition of two functions: the first traverses the input data and produces an intermediate data structure, whose type (possibly) differs from the type of the input data, which is traversed by the second function to produce 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. 1997; 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 some additional information in order to be able to produce its outcome, no such method produces satisfactory results. In fact, as a side-effect of eliminating the intermediate structure, these methods 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 such as

$$prog = cons \circ prod$$

where

$$prod :: a \rightarrow (b, z)$$
$$cons :: (b, z) \rightarrow c$$

into programs that construct no intermediate data-structure b and that traverse the input structure a 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 *prog* into a circular program. We allow the first traversal, *prod*, to produce completely general intermediate structures b and context informations z. The second traversal, *cons* then uses z so that, consuming the intermediate structure b, 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 pratical to implement in a compiler.

The *pfold/buildp* rule states that program compositions such as the one defined in *prog* naturally induce circular programs. That is, we calculate circular programs from programs such as *prog*. The circular programs we derive compute the same results as *prog*, but they do this by performing a single traversal over a, 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 this rule is correct. We also present formal evidence that the *pfold/buildp* 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 when the original programs do.

This chapter is organized as follows. In section 2.2, 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*, the *pfold/buildp* rule will be characterized by certain program schemes, which will be presented in section 2.3 together with the algebraic laws necessary for the proof of the new rule. In section 2.4 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. In section 2.4.1, we review a systematic study performed by Voigtländer (2008) on the semantics of the *pfold/buildp* rule. Section 2.5 illustrates the application of our method to a realistic programming problem: the Algol 68 scope rules. Section 2.6 concludes the chapter.

2.2 Circular programs

Circular programs were proposed as a program transformation 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.

```
data LeafTree = Leaf Int
              | Fork (LeafTree, LeafTree)

transform :: LeafTree → LeafTree
transform t = replace (t, tmin t)
```

$$\begin{aligned}
&\mathit{tmin} :: \mathit{LeafTree} \to \mathit{Int}\\
&\mathit{tmin}\ (\mathit{Leaf}\ n) \qquad\quad = n\\
&\mathit{tmin}\ (\mathit{Fork}\ (l,r)) = \mathit{min}\ (\mathit{tmin}\ l)\ (\mathit{tmin}\ r)
\end{aligned}$$

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

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.

2.2.1 Bird's method

Bird (1984) proposed a method for deriving single traversal programs from straightforward solutions, using tupling, folding-unfolding and local recursion. 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 a tupling step, it is essential that the two functions traverse the same data structure.

$$\mathit{repmin}\ (t, m) = (\mathit{replace}\ (t, m), \mathit{tmin}\ t)$$

We may now synthesize a recursive definition for *repmin* using the standard application of the fold-unfold method. Two cases have to be considered, corresponding to the two constructors of the traversed data structure:

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

$$\begin{aligned}
&\mathit{repmin}\ (\mathit{Fork}\ (l,r), m)\\
&\quad = (\mathit{replace}\ (\mathit{Fork}\ (l,r), m), \mathit{tmin}\ (\mathit{Fork}\ (l,r)))
\end{aligned}$$

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

Finally, local recursion 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*.

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

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 circular definitions always induce cycles and non-termination under a *strict* evaluation mechanism, they can sometimes be evaluated using a *lazy* evaluation strategy. The *lazy* evaluation machinery is able to determine, at runtime, the right order to evaluate such circular definitions, if that order exists.

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.

2.2.2 Our method

The calculational method that we propose in this chapter 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 of 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*. Thus, we define the following solution to *repmin*[1]:

```
$transform :: LeafTree \to LeafTree$
$transform\ t = replace \circ tmint\ \$\ t$

$tmint :: LeafTree \to (LeafTree, Int)$
$tmint\ (Leaf\ n)$        $= (Leaf\ n, n)$
$tmint\ (Fork\ (l, r)) = (Fork\ (l', r'), min\ n_1\ n_2)$
        $\mathbf{where}\ (l', n_1) = tmint\ l$
               $(r', n_2) = tmint\ r$

$replace :: (LeafTree, Int) \to LeafTree$
$replace\ (Leaf\ \_, m)$        $= Leaf\ m$
$replace\ (Fork\ (l, r), m) = Fork\ (replace\ (l, m), replace\ (r, m))$
```

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[2].

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. Later in this chapter we will present a realistic example (in section 2.5) which shows that, in general, the gluing trees need to grow from traversal to

[1]We have used $(\$) :: (a \to b) \to a \to b$ in the expression $replace \circ tmint\ \$\ t$ to avoid the use of parenthesis. The same expression could be defined as $(replace \circ tmint)\ t$.

[2]Notice that we could have used a *smaller* gluing tree. Indeed, since function *replace* discards the values stored in the leaves of the intermediate tree, it suffices if function *tmint* generates a tree of the same shape as the input one. We will come back to this in section 6.2.2.

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:

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

where π_1 and π_2 are the projection functions $\pi_1\ (x, y) = x$ and $\pi_2\ (x, y) = y$ and $(f\ \times\ g)\ (x, y) = (f\ x, g\ y)$. Therefore, we can redefine *transform* as:

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

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

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

$$\begin{aligned}&(r', n_2) = \mathit{repmin}\ r\\ \textbf{in}\ &(\mathit{Fork}\ (l', r'), \mathit{min}\ n_1\ n_2)\end{aligned}$$

In our method this synthesis will be obtained by the application of a particular shortcut 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 did eliminate 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,

$$\pi_2 \circ \mathit{tmint} = \pi_2 \circ \mathit{repmin} \tag{2.1}$$

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 single 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:

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

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 always 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 (2.1), 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 to this particular example, but 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.

2.3 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 the section 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. $f \perp = \perp$.

2.3.1 Data types

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

$$map_F\ id = id \tag{2.2}$$

$$map_F\ (f \circ g) = map_F\ f \circ map_F\ g \tag{2.3}$$

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.

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

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:

$$\begin{array}{l} (\times) :: (a \to c) \to (b \to d) \to (a, b) \to (c, d) \\ (f\ \times\ g)\ (a, b) = (f\ a, g\ b) \end{array}$$

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:

$$\begin{array}{l} \pi_1 :: (a, b) \to a \\ \pi_1\ (a, b) = a \end{array}$$

$$\begin{array}{l} \pi_2 :: (a, b) \to b \\ \pi_2\ (a, b) = b \end{array}$$

$$\begin{array}{l} (\vartriangle) :: (c \to a) \to (c \to b) \to c \to (a, b) \\ (f\ \vartriangle\ g)\ c = (f\ c, g\ c) \end{array}$$

Among other properties, it holds that

$$f \circ \pi_1 = \pi_1 \circ (f \times g) \tag{2.4}$$

$$g \circ \pi_2 = \pi_2 \circ (f \times g) \tag{2.5}$$

$$f = ((\pi_1 \circ f) \vartriangle (\pi_2 \circ f)) \tag{2.6}$$

Another case of a 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[3]:

$$\textbf{data}\ \tau\ (\alpha_1, \cdots, \alpha_m) = C_1\ (\tau_{1,1}, \cdots, \tau_{1,k_1})\ \mid \cdots \mid\ C_n\ (\tau_{n,1}, \cdots, \tau_{n,k_n})$$

where each $\tau_{i,j}$ is restricted to be a constant type (like *Int* or *Char*), a type variable α_t, a type constructor D applied to a type $\tau'_{i,j}$ or $\tau\ (\alpha_1, \cdots, \alpha_m)$ 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 constructors C_i are omitted. Every $\tau_{i,j}$ that consists of a type variable α_t or of a constant

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

type remain unchanged and occurrences of $\tau\ (\alpha_1, \cdots, \alpha_m)$ 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 $\sigma_{i,j} = \tau_{i,j}[\tau\ (\alpha_1, \cdots, \alpha_m) := a]$[4]. The body of the corresponding mapping function $map_F :: (a \to b) \to (F\ a \to F\ b)$ is similar to that of $F\ a$, with the difference that the occurrences of the type variable a are replaced by a function $f :: a \to b$:

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

with

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

where map_D represents the *map* function $map_D :: (a \to b) \to (D\ a \to D\ b)$ corresponding to the type constructor D.

For example, for the type of leaf trees

data $LeafTree = Leaf\ Int$
$\qquad\qquad\qquad\quad \mid Fork\ (LeafTree, LeafTree)$

we can derive a functor T given by

$$T\ a = Int + (a, a)$$

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

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

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

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

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

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 $in_F :: F\ \mu F \to \mu F$ and $out_F :: \mu F \to F\ \mu F$, 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 $\mu T = LeafTree$ and

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

$$\begin{aligned} &out_T :: LeafTree \to\ T\ LeafTree \\ &out_T\ (Leaf\ n) = Left\ n \\ &out_T\ (Fork\ (l, r)) = Right\ (l, r) \end{aligned}$$

2.3.2 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* (also called *catamorphism*), denoted by $fold\ h :: \mu F \to a$, is defined as the least function f that satisfies the following equation:

$$f \circ in_F = h \circ map_F\ f$$

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

$$\begin{aligned}&\mathit{fold} &&:: (F\ a \to a) \to \mu F \to a\\ &\mathit{fold}\ h &&= h \circ \mathit{map}_F\ (\mathit{fold}\ h) \circ \mathit{out}_F\end{aligned}$$

A function $h :: F\ a \to a$ is called an *F-algebra*[5]. The functor F plays the role of the 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 $f \circ h = k \circ \mathit{map}_F\ f$. Notice that $\mathit{fold}\ h$ is a homomorphism between the algebras in_F and h.

For example, for leaf trees fold is given by:

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

And we can express the recursive function *tmin*,

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

in terms of a fold for leaf trees[6]:

$$\mathit{tmin} = \mathit{fold}_T\ (\mathit{id}, \mathit{uncurry}\ \mathit{min})$$

Fold enjoys many algebraic laws that are useful for program transformation (Augusteijn 1998). 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).

[5]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 $(h_1, \ldots, h_n)$.

[6]*uncurry* takes a function $f :: a \to b \to c$ and produces a function $f' :: (a, b) \to c$.

Law 2.3.1 (FOLD/BUILD RULE) *For h strict,*

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

where

$$\begin{array}{l} \mathit{build} :: (\forall\ a\ .\ (F\ a \to a) \to c \to a) \to c \to \mu F \\ \mathit{build}\ g = g\ \mathit{in}_F \end{array}$$

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

$$\mathit{fold}_T\ (h_1, h_2) \circ \mathit{build}_T\ g = g\ (h_1, h_2) \tag{2.7}$$

where

$$\begin{array}{l} \mathit{build}_T :: (\forall\ a\ .\ (\mathit{Int} \to a, (a, a) \to a) \to c \to a) \to c \to \mathit{LeafTree} \\ \mathit{build}_T\ g = g\ (\mathit{Leaf}, \mathit{Fork}) \end{array}$$

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 the following program, that computes the minimum value of a *mirrored* leaf tree.

$$\mathit{tmm} = \mathit{tmin} \circ \mathit{mirror}$$

$$\begin{array}{l} \mathit{mirror} :: \mathit{LeafTree} \to \mathit{LeafTree} \\ \mathit{mirror}\ (\mathit{Leaf}\ n) = \mathit{Leaf}\ n \\ \mathit{mirror}\ (\mathit{Fork}\ (l, r)) = \mathit{Fork}\ (\mathit{mirror}\ r, \mathit{mirror}\ l) \end{array}$$

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

$$\begin{array}{l} \mathit{mirror} = \mathit{build}_T\ g \\ \quad \textbf{where}\ g\ (\mathit{leaf}, \mathit{fork})\ (\mathit{Leaf}\ n) = \mathit{leaf}\ n \\ \qquad\qquad g\ (\mathit{leaf}, \mathit{fork})\ (\mathit{Fork}\ (l, r)) = \mathit{fork}\ (g\ (\mathit{leaf}, \mathit{fork})\ r, \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad g\ (\mathit{leaf}, \mathit{fork})\ l) \end{array}$$

Finally, by (2.7) we have that

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

Inlining, we have

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

As expected, this function does not construct the intermediate mirror tree.

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

Law 2.3.2 (FOLD/BUILDP RULE) *For h strict,*

$$g :: \forall\ a\ .\ (F\ a \to a) \to c \to (a,z)$$
$$\Rightarrow$$
$$(fold\ h\ \times\ id) \circ buildp\ g = g\ h$$

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,

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

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:

$$(fold_T\ (h_1,h_2)\ \times\ id) \circ buildp_T\ g = g\ (h_1,h_2) \tag{2.8}$$

where

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

The assumption about the strictness of the algebra disappears by the same reason as for (2.7).

To see an example of the application of this law, consider the program *ssqm*: it replaces every leaf in a tree by its square while computing the minimum value of the tree; later, it sums all the (squared) elements of an input tree.

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

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

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

To apply Law (2.8) we have to express *sumt* as a fold and *gentsqmin* in terms of buildp_T:

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

Hence, by (2.8), we have

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

Inlining, we obtain

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

Finally, the following property is an immediate consequence of Law 2.3.2.

Law 2.3.3 *For any strict h,*

$$g :: \forall\ a\ .\ (F\ a \to a) \to c \to (a, z)$$

$$\Rightarrow$$

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

Proof

$$\begin{aligned} &\quad \pi_2 \circ g\ in_F \\ = &\qquad \{\ (2.5)\ \} \\ &\quad \pi_2 \circ (fold\ h\ \times\ id) \circ g\ in_F \\ = &\qquad \{\ \text{Law 2.3.2}\ \} \\ &\quad \pi_2 \circ g\ h \qquad\qquad \Box \end{aligned}$$

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.

2.3.3 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 $f :: (\mu F, z) \to a$, 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:

$$\begin{aligned}&f :: (\mu F, z) \to a\\ &f\ (t, z) = \mathit{fold}\ h\ t\end{aligned}$$

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

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

which can be defined as:

$$\mathit{replace}\ (t, m) = \mathit{fold}_T\ (\lambda n \to \mathit{Leaf}\ m, \mathit{Fork})\ t$$

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

$$\mathit{curry}\ \mathit{replace} = \mathit{fold}_T\ (\lambda n \to \mathit{Leaf}, \lambda(f, f') \to \mathit{Fork} \circ (f \vartriangle f'))$$

This is an alternative we will study in detail in section 4.2.

A third alternative is to define the function $f :: (\mu F, z) \to a$ 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:

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

that satisfies the coherence axioms:

$$\mathit{map}_F\ \pi_1 \circ \tau^F = \pi_1$$

$$map_F\ \alpha \circ \tau^F = \tau^F \circ (\tau^F \ \times \ id) \circ \alpha$$

where $\alpha :: (a,(b,c)) \ \rightarrow \ ((a,b),c)$ is the product associativity (see (Pardo 2002; Cockett and Spencer 1991; Cockett and Fukushima 1992) for further 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:

$$\begin{array}{l}\tau^T :: (T\ \ a, z) \rightarrow T\ (a, z)\\ \tau^T\ (\mathit{Left}\ n, z) = \mathit{Left}\ n\\ \tau^T\ (\mathit{Right}\ (a, a'), z) = \mathit{Right}\ ((a, z), (a', z))\end{array}$$

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

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

Observe that now function h also accepts the value of the parameter. It is a function of the form $(h_1 \triangledown \ \ldots \ \triangledown h_n) \circ d$ where each $h_i :: (F_i\ \ a, z) \rightarrow a$ if $F\ \ a = F_1\ \ a + \cdots + F_n\ \ a$, and $d :: (x_1 + \cdots + x_n, z) \rightarrow (x_1, z) + \cdots + (x_n, z)$ 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:

$$\begin{array}{l}\mathit{pfold}_T :: ((\mathit{Int}, z) \rightarrow a, ((a, a), z) \rightarrow a) \rightarrow (\mathit{LeafTree}, z) \rightarrow a\\ \mathit{pfold}_T\ (h_1, h_2) = p_T\\ \quad \textbf{where}\ p_T\ (\mathit{Leaf}\ \ n, z) = h_1\ (n, z)\\ \qquad\qquad p_T\ (\mathit{Fork}\ (l, r), z) = h_2\ ((p_T\ (l, z), p_T\ (r, z)), z)\end{array}$$

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.

$$\mathit{pfold}\ h\ (t, z) = \mathit{fold}\ k\ t\ \ \textbf{where}\ \ k_i\ x = h_i\ (x, z) \tag{2.9}$$

Like fold, pfold satisfies a set of algebraic laws. We do not show any of them here as they are not necessary for the calculational work presented in this thesis. The interested reader may consult (Pardo 2001, 2002).

2.4 The pfold/buildp rule

In this section, we present a generic formulation and proof of correctness of a transformation rule for calculating circular programs. The rule takes a composition of the form $\mathit{cons} \circ \mathit{prod}$, composed by a producer $\mathit{prod} :: a \to (t, z)$ followed by a consumer $\mathit{cons} :: (t, z) \to b$, 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: $\mathit{pfold}\ h \circ \mathit{buildp}\ g$.

Law 2.4.1 (PFOLD/BUILDP RULE) *For any* $h = (h_1 \triangledown \; \ldots \; \triangledown \; h_n) \circ d$, [7]

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

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 2.3.2. For that reason we need first to express the pfold as a fold.

$$\begin{aligned}
& \mathit{pfold} \; h \circ \mathit{buildp} \; g \; \$ \; c \\
= & \quad \{ \; \text{definition of } \mathit{buildp} \; \} \\
& \mathit{pfold} \; h \circ g \; \mathit{in}_F \; \$ \; c \\
= & \quad \{ \; (2.6) \; \} \\
& \mathit{pfold} \; h \circ (((\pi_1 \circ g \; \mathit{in}_F) \vartriangle (\pi_2 \circ g \; \mathit{in}_F))) \; \$ \; c \\
= & \quad \{ \; (2.9) \; \} \\
& \mathit{fold} \; k \circ \pi_1 \circ g \; \mathit{in}_F \; \$ \; c \\
& \quad \textbf{where} \; z = \pi_2 \circ g \; \mathit{in}_F \; \$ \; c \\
& \qquad\qquad k_i \; \bar{x} = h_i \; (\bar{x}, z) \\
= & \quad \{ \; (2.4) \; \} \\
& \pi_1 \circ (\mathit{fold} \; k \; \times \; \mathit{id}) \circ g \; \mathit{in}_F \; \$ \; c \\
& \quad \textbf{where} \; z = \pi_2 \circ g \; \mathit{in}_F \; \$ \; c \\
& \qquad\qquad k_i \; \bar{x} = h_i \; (\bar{x}, z) \\
= & \quad \{ \; \text{Law 2.3.2} \; \} \\
& \pi_1 \circ g \; k \; \$ \; c
\end{aligned}$$

[7] We denote by $\bar{x}$ a tuple of values $(x_1, \cdots, x_{r_i})$.

$$\begin{aligned}&\textbf{where}\ z = \pi_2 \circ g\ in_F \ \$\ c\\ &\qquad k_i\ \bar{x} = h_i\ (\bar{x}, z)\end{aligned}$$

Law 2.3.2 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 possible the safe introduction of a circularity is Law 2.3.3, 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.

$$\begin{aligned}&\pi_1 \circ g\ k\ \$\ c\\ &\quad \textbf{where}\ z = \pi_2 \circ g\ in_F\ \$\ c\\ &\qquad\quad k_i\ \bar{x} = h_i\ (\bar{x}, z)\\ =&\quad \{\ \text{Law 2.3.3}\ \}\\ &\pi_1 \circ g\ k\ \$\ c\\ &\quad \textbf{where}\ \boxed{z} = \pi_2 \circ g\ k\ \$\ c\\ &\qquad\quad k_i\ \bar{x} = h_i\ (\bar{x}, \boxed{z})\\ =&\quad \{\ (2.6)\ \}\\ &v\\ &\quad \textbf{where}\ (v, \boxed{z}) = g\ k\ c\\ &\qquad\quad k_i\ \bar{x} = h_i\ (\bar{x}, \boxed{z}) \qquad \Box\end{aligned}$$

Now, let us see the application of the pfold/buildp rule in the case of the *repmin* problem. Recall the definition, presented on page 19, that we want to transform:

$$transform :: LeafTree \rightarrow LeafTree$$

$$\mathit{transform}\ t = \mathit{replace} \circ \mathit{tmint}\ \$\ t$$

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

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

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

Therefore, by applying Law 2.4.1 we get:

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

Inlining, we obtain the definition we showed previously in section 2.2.2:

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

The pfold/buildp rule (Law 2.4.1) can also be used in an automated way. Indeed, it is possible to implement each instance of pfold/buildp using the rewrite rules (RULES pragma) of the Glasgow Haskell Compiler (GHC).

Again, let us consider the *repmin* solution:

$$\begin{aligned}
&\mathit{transform} :: \mathit{LeafTree} \to \mathit{LeafTree}\\
&\mathit{transform}\ t = \mathit{replace} \circ \mathit{tmint}\ \$\ t
\end{aligned}$$

This solution constructs an intermediate leaf tree and functions *replace* and *tmint* have already been expressed in terms of pfold and buildp for leaf trees, respectively. Then, in order to apply Law 2.4.1 to *transform*, we define the rewrite rule:

```
{-# RULES
  "pfoldT/buildpT"
    $\forall\ c\ h_1\ h_2\ (g :: \forall\ a\ .\ (Int \rightarrow a, (a, a) \rightarrow a) \rightarrow b \rightarrow (a, z))\ .$
      $pfold_T\ (h_1, h_2)\ (buildp_T\ g\ c)$
        $= \textbf{let}\ (v, \boxed{z}) = g\ (k_1, k_2)\ c$
              $k_1\ x = h_1\ (x, \boxed{z})$
              $k_2\ (l, r) = h_2\ ((l, r), \boxed{z})$
          $\textbf{in}\ v$
    #-}
```

Once this rule is defined and activated, the compiler will automatically replace all the expressions occurring in a program that match the left hand side of the `"pfoldT/builpT"` rule by its right hand side. In particular, the rule will apply to *transform*, which is converted into a circular program.

2.4.1 Semantics of the pfold/buildp rule

According to Danielsson et al. (2006), Law 2.4.1 is morally correct *only*, in *Haskell*. In fact, in the formal proof of our rule, surjective pairing (Law (2.6)) is applied twice to the result of function g. However, (2.6) is not valid in *Haskell*: though it holds for defined values, it fails when the result of function g is undefined, because $\perp$ is different from $(\perp, \perp)$ as a consequence of lifted products. Therefore, (2.6) is morally correct *only* and, in the same sense, so is our rule. In (Fernandes et al. 2007), we also pointed out that, due to the presence of *seq* in *Haskell*, additional pre-conditions may need to be defined in our rule in order to guarantee its correctness in *Haskell* (Johann and Voigtländer 2004).

Following our work, Voigtländer (2008) performed a rigorous study on various shortcut fusion rules, for languages like *Haskell*. In particular, the author presents semantic and pragmatic considerations on Law 2.4.1. As a first result, pre-conditions are added to our rule, so that its total correctness can be established.

The definition of Law 2.4.1 becomes:

Law 2.4.2 (Haskell valid pfold/buildp rule) *For any*

$h = (h_1 \triangledown \ \ldots \ \triangledown \ h_n) \circ d,$

$\forall\ i \ \in \{1, .., n\}.\ h_i\ ((\bot, ..., \bot), \bot) \ \neq \ \bot$

$$g :: \forall\ a\ .\ (F\ a \to a) \to c \to (a, z)$$

$$\Rightarrow$$

$$\begin{aligned}
&\mathit{pfold}\ h \circ \mathit{buildp}\ g\ \$\ c\\
&\quad = v\\
&\quad \textbf{where}\ (v, \boxed{z}) = g\ k\ c\\
&\qquad\qquad k = k_1 \triangledown \ \ldots \ \triangledown \ k_n\\
&\qquad\qquad k_i\ \bar{x} = h_i\ (\bar{x}, \boxed{z})
\end{aligned}$$

It is now possible to prove total correctness of Law 2.4.2 (Voigtländer 2008). However, although Law 2.4.2 is the one that guarantees totally correct transformations, in the semantics of *Haskell*, it is somewhat *pessimistic*.

By this we mean that even if the newly added pre-condition is violated, it does not necessarily imply that the Law gets broken. In fact, Voigtländer (2008) presents an example where such pre-condition is violated, causing no harm in the calculated equivalent program. We review here such an example.

Consider the following programming problem: from the initial part of an input list before a certain predicate holds for the first time, return those elements that are repeated afterwards. The specification of a natural solution to this problem is as follows:

$$\begin{aligned}
&\mathit{repeatedAfter} :: \mathit{Eq}\ b \Rightarrow (b \to \mathit{Bool}) \to [\,b\,] \to [\,b\,]\\
&\mathit{repeatedAfter}\ p\ bs = (\mathit{pfilter}\ \mathit{elem}) \circ (\mathit{splitWhen}\ p)\ \$\ bs
\end{aligned}$$

$$\begin{aligned}
&\mathit{pfilter} :: (b \to z \to \mathit{Bool}) \to ([\,b\,], z) \to [\,b\,]\\
&\mathit{pfilter}\ \_\ ([\,], \_) = [\,]\\
&\mathit{pfilter}\ p\ (b : bs, z) = \textbf{let}\ bs' = \mathit{pfilter}\ p\ (bs, z)
\end{aligned}$$

```
                    in if $p\ b\ z$
                       then $b : bs'$
                       else $bs'$

$splitWhen :: (b \to Bool) \to [\,b\,] \to ([\,b\,], [\,b\,])$
$splitWhen\ p\ bs$
  $=$ case $bs$ of $[\,] \to ([\,], bs)$
             $b : bs' \to$ if $p\ b$
                        then $([\,], bs)$
                        else let $(xs, ys) = splitWhen\ p\ bs'$
                             in $(b : xs, ys)$
```

This definition uses a list as the intermediate structure that serves the purpose of gluing the two composed functions. This intermediate list can be eliminated using Law 2.4.1. However, in order to apply that law to the *repeatedAfter* program, *pfilter* and *splitWhen p* must first be given in terms of pfold and buildp for lists (the type of the intermediate structure), respectively. The definition of pfold and buildp for lists is as follows.

```
$buildp_L :: (\forall\ b\ .\ (b, (a, b) \to b) \to c \to (b, z)) \to c \to ([\,a\,], z)$
$buildp_L\ g = g\ ([\,], uncurry\ (:))$

$pfold_L :: (z \to b, ((a, b), z) \to b) \to ([\,a\,], z) \to b$
$pfold_L\ (hnil, hcons) = p_L$
  where $p_L\ ([\,], z)\qquad = hnil\ z$
        $p_L\ (a : as, z) = hcons\ ((a, p_L\ (as, z)), z)$
```

Now, we write *pfilter* and *splitWhen p* in terms of them:

```
$splitWhen\ p = buildp_L\ go$
  where $go\ (nil, cons)\ bs$
          $=$ case $bs$ of $[\,] \to (nil, bs)$
                    $b : bs' \to$ if $p\ b$
                               then $(nil, bs)$
                               else let $(xs, ys)$
```

```
                                      $= go\ (nil, cons)\ bs'$
                                  $\textbf{in}\ (cons\ (b, xs), ys)$

   $pfilter\ p = pfold_L\ (hnil, hcons)$
     $\textbf{where}\ hnil\ \_ = [\,]$
           $hcons\ ((b, bs), z) = \textbf{if}\ (p\ b\ z)\ \textbf{then}\ (b : bs)\ \textbf{else}\ bs$
```

Regarding this example, we may notice that $hcons\ ((\bot, \bot), \bot) = \bot$, given that ($\textbf{if}\ elem\ \bot\ \bot\ \textbf{then}\ \bot : \bot\ \textbf{else}\ \bot$) equals $\bot$. This means that the pre-condition $\forall\ i\ .\ h_i\ ((\bot, ..., \bot), \bot) \neq \bot$, newly added to Law 2.4.1, fails. However, it is still possible to use Law 2.4.1 to calculate a correct circular program equivalent to the *repeatedAfter* program presented earlier:

```
   $repeatedAfter\ p\ bs = a$
     $\textbf{where}\ (a, \boxed{z}) = go'\ bs$
           $go'\ bs = \textbf{case}\ bs\ \textbf{of}\ [\,] \rightarrow ([\,], bs)$
                               $b : bs' \rightarrow \textbf{if}\ p\ b$
                                     $\textbf{then}\ ([\,], bs)$
                                     $\textbf{else let}\ (xs, ys) = go'\ bs'$
                                        $\textbf{in}\ (\textbf{if}\ elem\ b\ \boxed{z}$
                                           $\textbf{then}\ b : xs$
                                           $\textbf{else}\ xs, ys)$
```

It is in this sense that we say Law 2.4.2 is *pessimistic*. However, this Law is the most general one can present, so far, in terms of total correctness.

In chapter 4, we will present an alternative way to transform compositions between pfold and buildp such that, instead of circular programs, higher-order programs are obtained as result. A good thing about the new transformation is that its total correctness can be established defining fewer pre-conditions than the ones defined in Law 2.4.2.

2.5 Algol 68 scope rules

In the previous section we have introduced new calculation rules and applied them to small, but illustrative examples. In this section, we consider the application of the calculational methods we have introduced in this chapter to a real example: the Algol 68 scope rules. These rules are used, for example, in the Eli system[8] (Kastens et al. 1998; Waite et al. 2007) to define a generic component for the name analysis task of a compiler.

The problem we consider is as follows: 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 this case, the definition of x in 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:

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

In *Haskell* we may define the following data-types to represent *Block* programs.

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

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.

[8]A well known compiler generator toolbox.

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 a *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 can be 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.

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:

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

Errors resulting from duplicate declarations, computed in the first traversal, 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 consists of:

$$\begin{array}{l} semantics :: Prog \to [\,Var\,] \\ semantics = missing \circ (duplicate\ 0\ [\,]) \end{array}$$

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

$$\begin{array}{l} duplicate :: Int \to Env \to Prog \to (Prog_2, Env) \\ duplicate\ lev\ ds\ [\,] = ([\,], ds) \\ \\ duplicate\ lev\ ds\ (Use\ var : its) \\ \quad = \textbf{let}\ (its_2, ds') = duplicate\ lev\ ds\ its \\ \quad\quad\ \textbf{in}\ \ (Use_2\ var : its_2, ds') \end{array}$$

```
$duplicate\ lev\ ds\ (Decl\ var : its)$
    $= \mathbf{let}\ (its_2, ds') = duplicate\ lev\ ((var, lev) : ds)\ its$
      $\mathbf{in}\ \ \mathbf{if}\ ((var, lev) \in ds)\ \mathbf{then}\ (Dupl_2\ var : its_2, ds')$
                                   $\mathbf{else}\ (its_2, ds')$

$duplicate\ lev\ ds\ (Block\ nested : its)$
    $= \mathbf{let}\ (its_2, ds') = duplicate\ lev\ ds\ its$
      $\mathbf{in}\ \ (Block_2\ (lev + 1, nested) : its_2, ds')$
```

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:

```
$missing :: (Prog_2, Env) \to [\,Var\,]$
$missing\ ([\,], \_) = [\,]$

$missing\ (Use_2\ var : its_2, env)$
    $= \mathbf{let}\ errs = missing\ (its_2, env)$
      $\mathbf{in}\ \mathbf{if}\ (var \in map\ \pi_1\ env)\ \mathbf{then}\ errs$
                                     $\mathbf{else}\ var : errs$

$missing\ (Dupl_2\ var : its_2, env)$
    $= var : missing\ (its_2, env)$

$missing\ (Block_2\ (lev, its) : its_2, env)$
    $= \mathbf{let}\ errs_1 = missing \circ (duplicate\ lev\ env)\ \$\ its$
            $errs_2 = missing\ (its_2, env)$
      $\mathbf{in}\ errs_1 +\!\!+\ errs_2$
```

The construction and traversal of an intermediate data structure, however, is not essential to implement the semantic analysis described. Indeed, in the next section we will transform *semantics* into an equivalent program that does not construct any intermediate structure.

2.5.1 Calculating a circular program

In this section, we calculate a circular program equivalent to the *semantics* program presented in the previous section. In our calculation, we will use the specific instance of Law 2.4.1 for the case when the intermediate structure gluing the consumer and producer functions is a list:

Law 2.5.1 *(*PFOLD/BUILDP RULE FOR LISTS)

$$
\begin{aligned}
& \mathit{pfold}_L\ (\mathit{hnil}, \mathit{hcons}) \circ \mathit{buildp}_L\ g\ \$\ c \\
& \quad = v \\
& \quad \textbf{where}\ (v, \boxed{z}) = g\ (\mathit{knil}, \mathit{kcons})\ c \\
& \qquad\qquad \mathit{knil} = \mathit{hnil}\ \boxed{z} \\
& \qquad\qquad \mathit{kcons}\ (x, y) = \mathit{hcons}\ ((x, y), \boxed{z})
\end{aligned}
$$

where the schemes pfold_L and buildp_L have already been defined as:

$$
\begin{aligned}
& \mathit{buildp}_L :: (\forall\ b\ .\ (b, (a, b) \to b) \to c \to (b, z)) \to c \to ([\,a\,], z) \\
& \mathit{buildp}_L\ g = g\ ([\,], \mathit{uncurry}\ (:))
\end{aligned}
$$

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

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

$$
\begin{aligned}
& \mathit{missing} = \mathit{pfold}_L\ (\mathit{hnil}, \mathit{hcons}) \\
& \quad \textbf{where}\ \mathit{hnil}\ \_ = [\,] \\
& \\
& \qquad\qquad \mathit{hcons}\ ((\mathit{Use}_2\ \mathit{var}, \mathit{errs}), \mathit{env}) \\
& \qquad\qquad\quad = \textbf{if}\ (\mathit{var} \in \mathit{map}\ \pi_1\ \mathit{env})\ \textbf{then}\ \mathit{errs} \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{else}\ \mathit{var} : \mathit{errs} \\
& \\
& \qquad\qquad \mathit{hcons}\ ((\mathit{Dupl}_2\ \mathit{var}, \mathit{errs}), \mathit{env}) \\
& \qquad\qquad\quad = \mathit{var} : \mathit{errs} \\
& \\
& \qquad\qquad \mathit{hcons}\ ((\mathit{Block}_2\ (\mathit{lev}, \mathit{its}), \mathit{errs}), \mathit{env})
\end{aligned}
$$

$$
\begin{array}{l}
= \textbf{let}\ \mathit{errs}_1 = \mathit{missing} \circ (\mathit{duplicate}\ \mathit{lev}\ \mathit{env})\ \$\ \mathit{its} \\
\quad \textbf{in}\ \mathit{errs}_1 +\!\!\!+\ \mathit{errs}
\end{array}
$$

and *duplicate* in terms of buildp_L,

```
$\mathit{duplicate}\ \mathit{lev}\ \mathit{ds} = \mathit{buildp}_L\ (g\ \mathit{lev}\ \mathit{ds})$
  $\textbf{where}\ g\ \mathit{lev}\ \mathit{ds}\ (\mathit{nil}, \mathit{cons})\ [\,] = (\mathit{nil}, \mathit{ds})$

        $g\ \mathit{lev}\ \mathit{ds}\ (\mathit{nil}, \mathit{cons})\ (\mathit{Use}\ \mathit{var} : \mathit{its})$
          $= \textbf{let}\ (\mathit{its}_2, \mathit{ds}') = g\ \mathit{lev}\ \mathit{ds}\ (\mathit{nil}, \mathit{cons})\ \mathit{its}$
            $\textbf{in}\ \ (\mathit{cons}\ (\mathit{Use}_2\ \mathit{var}, \mathit{its}_2), \mathit{ds}')$

        $g\ \mathit{lev}\ \mathit{ds}\ (\mathit{nil}, \mathit{cons})\ (\mathit{Decl}\ \mathit{var} : \mathit{its})$
          $= \textbf{let}\ (\mathit{its}_2, \mathit{ds}') = g\ \mathit{lev}\ ((\mathit{var}, \mathit{lev}) : \mathit{ds})\ (\mathit{nil}, \mathit{cons})\ \mathit{its}$
            $\textbf{in}\ \ \textbf{if}\ ((\mathit{var}, \mathit{lev}) \in \mathit{ds})\ \textbf{then}\ (\mathit{cons}\ (\mathit{Dupl}_2\ \mathit{var}, \mathit{its}_2), \mathit{ds}')$
                                              $\textbf{else}\ (\mathit{its}_2, \mathit{ds}')$

        $g\ \mathit{lev}\ \mathit{ds}\ (\mathit{nil}, \mathit{cons})\ (\mathit{Block}\ \mathit{nested} : \mathit{its})$
          $= \textbf{let}\ (\mathit{its}_2, \mathit{ds}') = g\ \mathit{lev}\ \mathit{ds}\ (\mathit{nil}, \mathit{cons})\ \mathit{its}$
            $\textbf{in}\ \ (\mathit{cons}\ (\mathit{Block}_2\ (\mathit{lev} + 1, \mathit{nested}), \mathit{its}_2), \mathit{ds}')$
```

we can apply Law 2.5.1 to the program $\mathit{semantics} = \mathit{missing} \circ (\mathit{duplicate}\ 0\ [\,])$, since this program has just been expressed as an explicit composition between a pfold_L and a buildp_L. We obtain a deforested circular definition, which, when inlined, gives the following program:

```
$\mathit{semantics}\ p = \mathit{errs}$
  $\textbf{where}$
    $(\mathit{errs}, \boxed{\text{env}}) = \mathit{gk}\ 0\ [\,]\ p$

    $\mathit{gk}\ \mathit{lev}\ \mathit{ds}\ [\,] = ([\,], \mathit{ds})$

    $\mathit{gk}\ \mathit{lev}\ \mathit{ds}\ (\mathit{Use}\ \mathit{var} : \mathit{its})$
      $= \textbf{let}\ (\mathit{errs}, \mathit{ds}') = \mathit{gk}\ \mathit{lev}\ \mathit{ds}\ \mathit{its}$
        $\textbf{in}\ (\textbf{if}\ (\mathit{var} \in \mathit{map}\ \pi_1\ \boxed{\text{env}})\ \textbf{then}\ \mathit{errs}$
                                          $\textbf{else}\ \mathit{var} : \mathit{errs}, \mathit{ds}')$

    $\mathit{gk}\ \mathit{lev}\ \mathit{ds}\ (\mathit{Decl}\ \mathit{var} : \mathit{its})$
```

$$\begin{aligned}
&= \textbf{let}\ (errs, ds') = gk\ lev\ ((var, lev) : ds)\ its\\
&\quad \textbf{in if}\ ((var, lev) \in ds)\ \textbf{then}\ (var : errs, ds')\\
&\qquad\qquad\qquad\qquad\qquad\quad \textbf{else}\ (errs, ds')
\end{aligned}$$

$$\begin{aligned}
&gk\ lev\ ds\ (Block\ nested : its)\\
&\quad = \textbf{let}\ (errs_2, ds') = gk\ lev\ ds\ its\\
&\qquad \textbf{in}\ (\textbf{let}\ errs_1 = missing \circ (duplicate\ (lev + 1)\ \boxed{\text{env}})\ \$\ nested\\
&\qquad\quad \textbf{in}\ errs_1 +\!\!+\ errs_2, ds')
\end{aligned}$$

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 this 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 2.5.1. We may also notice, however, that, for nested blocks, in the definition

$$\begin{aligned}
&gk\ lev\ ds\ (Block\ nested : its)\\
&\quad = \textbf{let}\ (errs_2, ds') = gk\ lev\ ds\ its\\
&\qquad \textbf{in}\ (\textbf{let}\ errs_1 = missing \circ (duplicate\ (lev + 1)\ env)\ \$\ nested\\
&\qquad\quad \textbf{in}\ errs_1 +\!\!+\ errs_2, ds')
\end{aligned}$$

an intermediate structure is still being used in order to glue functions *missing* and *duplicate* together. This intermediate structure can easily be eliminated, since we have already expressed function *missing* in terms of $pfold_L$, and function *duplicate* in terms of $buildp_L$. Therefore, by direct application of Law 2.5.1 to the above function composition, we obtain:

$$\begin{aligned}
&gk\ lev\ ds\ (Block\ nested : its)\\
&\quad = \textbf{let}\ (errs_2, ds') = gk\ lev\ ds\ its\\
&\qquad \textbf{in}\ (\textbf{let}\ (errs_1, \boxed{env_2}) = g\ (lev + 1)\ env\ (knil, kcons)\ nested\\
&\qquad\qquad \textbf{where}\ knil = hnil\ \boxed{env_2}\\
&\qquad\qquad\qquad\quad kcons\ x = hcons\ (x, \boxed{env_2})\\
&\qquad\quad \textbf{in}\ errs_1 +\!\!+\ errs_2, ds')
\end{aligned}$$

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_2.

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

We should notice that, in general, this extra effort is not necessary. In this particular example, this manipulation effort was made since it is possible to calculate two circular definitions from the straightforward solution and both circular functions share almost the 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:

```
$semantics\ p = errs$
  $\textbf{where}\ (errs, \boxed{\text{env}}) = gk\ 0\ [\,]\ \boxed{\text{env}}\ p$

        $gk\ lev\ ds\ env\ [\,] = ([\,], ds)$

        $gk\ lev\ ds\ env\ (Use\ var : its)$
          $= \textbf{let}\ (errs, ds') = gk\ lev\ ds\ env\ its$
            $\textbf{in}\ (\textbf{if}\ (var \in map\ \pi_1\ env)\ \textbf{then}\ errs$
                                        $\textbf{else}\ var : errs, ds')$

        $gk\ lev\ ds\ env\ (Decl\ var : its)$
          $= \textbf{let}\ (errs, ds') = gk\ lev\ ((var, lev) : ds)\ env\ its$
            $\textbf{in}\ \textbf{if}\ ((var, lev) \in ds)\ \textbf{then}\ (var : errs, ds')$
                                     $\textbf{else}\ (errs, ds')$

        $gk\ lev\ ds\ env\ (Block\ nested : its)$
          $= \textbf{let}\ (errs_2, ds') = gk\ lev\ ds\ env\ its$
            $\textbf{in}\ (\textbf{let}\ (errs_1, \boxed{env_2}) = gk\ (lev + 1)\ env\ \boxed{env_2}\ nested$
```

$\textbf{in}\ errs_1 +\!\!+\ errs_2, ds')$

Regarding the above program, we may notice that it has two circular definitions. One such definition occurs in the *semantics* function, and makes it possible for the environment of the outer level of a block program to be used while still being constructed. For the example sentence that we have considered before,

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

this circularity makes the environment $[(\texttt{"x"}, 0), (\texttt{"x"}, 0), (\texttt{"y"}, 0)]$ available to the function that traverses the outer block. The other circular definition, occurring in the last definition of function *gk*, is used so that, for every traversal of a nested sequence of instructions, its environment may readily be used. This means that the function traversing the nested block in the above example sentence may use the environment $[(\texttt{"x"}, 0), (\texttt{"x"}, 0), (\texttt{"y"}, 0), (\texttt{"y"}, 1)]$ even though it still needs to be constructed.

The introduction of these circularities, by the application of our calculational method, completely eliminated the intermediate lists of instructions that were used in the straightforward *semantics* solution we started with. Furthermore, the derivation of this circular program made it possible to obtain a *semantics* program that computes the list of errors that occur in a *Block* program by traversing it only once.

2.6 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 of the composition of a producer and a consumer function. 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 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.

In the next chapter, we will extend the applicability scope of our rule to programs relying on monadic computations.

Chapter 3

Calculation of Monadic Circular Programs

Summary

In the previous chapter, we have extended standard shortcut fusion to programs that rely on both an intermediate structure and an additional context parameter. 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: we derive monadic circular programs from strict ones, maintaining the global effects.
An important feature of our extensions is that they can be uniformly defined for a wide class of data types and monads, using generic calculation rules.

3.1 Introduction

In the previous chapter, we have shown how circular programs can be used to achieve intermediate structure deforestation in programs such as $prog = cons \circ prod$, where $prod :: a \rightarrow (t, z)$ and $cons :: (t, z) \rightarrow b$. This means that the producer function may generate, besides the intermediate structure t, 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. The circular program we derive does not construct any intermediate structure and 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 programs, 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 the kind of programs we studied in chapter 2, but in the context of monadic programming. Our approach follows the recent studies conducted by Ghani and Johann (2008); Manzino and Pardo (2008), that proposed monadic extensions to standard shortcut fusion. Our 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. For both cases, fusion is achieved by transforming the original program into a circular one. We do not consider the case where the producer is given by a pure function (and the consumer is given by a monadic one) since it can already be fused using the pfold/buildp fusion rule presented before.

This chapter is organized as follows. Sections 3.2 and 3.3 present two motivating examples that serve to illustrate the applicability of our techniques. Indeed, for each of the examples, we will derive an equivalent monadic circular program. All the programs presented in this chapter (both the example programs we start with and their calculated equivalents) will be monadic. The generic constructions that give rise to the specific laws presented in those examples are developed in section 3.4. We present the generic formulation of the laws for calculating monadic circular programs in section 3.4. Finally, in section 3.5, we conclude the chapter.

3.2 Bit string transformation

To illustrate our techniques to derive circular monadic programs we first consider an example based on a simple bit string conversion that has applications in cryptography (Baier et al. 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 will define a function that relies on a monad. In general, a monad is a triple $(M, \mathit{return}, \gg\!=)$ consisting of a type constructor M and two polymorphic functions:

$$\begin{array}{l}\mathit{return} :: a \rightarrow M\ a \\ (\gg\!=) :: M\ a \rightarrow (a \rightarrow M\ b) \rightarrow M\ b\end{array}$$

that obey the following laws

$$\begin{array}{rcl} m \gg\!= \mathit{return} & = & m \\ \mathit{return}\ x \gg\!= f & = & f\ x \\ (m \gg\!= f) \gg\!= g & = & m \gg\!= (\lambda x \rightarrow f\ x \gg\!= g) \end{array}$$

In *Haskell*, we can capture the definition of a monad as a type class.

```
class Monad m where
  return :: a → m a
  (>>=) :: m a → (a → m b) → m b
```

To implement the parser for bit strings, we will adopt the usual definition of parser monad (see (Hutton and Meijer 1998) for more details):

$$\textbf{newtype}\ \mathit{Parser}\ a = P\ (\mathit{String} \to [(a, \mathit{String})])$$

$$\begin{aligned}
&\textbf{instance}\ \mathit{Monad}\ \mathit{Parser}\ \textbf{where}\\
&\quad \mathit{return}\ a \quad = P\ (\lambda cs \to [(a, cs)])\\
&\quad p\ \gg\!=\ f = P\ (\lambda cs \to \mathit{concat}\ [\mathit{parse}\ (f\ a)\ cs' \mid (a, cs') \leftarrow \mathit{parse}\ p\ cs])
\end{aligned}$$

$$\begin{aligned}
&\mathit{parse} :: \mathit{Parser}\ a \to \mathit{String} \to [(a, \mathit{String})]\\
&\mathit{parse}\ (P\ p) = p
\end{aligned}$$

$$\begin{aligned}
&(\uplus) :: \mathit{Parser}\ a \to \mathit{Parser}\ a \to \mathit{Parser}\ a\\
&(P\ p) \uplus (P\ q) = P\ (\lambda cs \to \textbf{case}\ p\ cs +\!\!+\ q\ cs\ \textbf{of}\ [\,] \quad\quad \to [\,]\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad (x : xs) \to [x])
\end{aligned}$$

$$\begin{aligned}
&\mathit{pzero} :: \mathit{Parser}\ a\\
&\mathit{pzero} = P\ (\lambda cs \to [\,])
\end{aligned}$$

$$\begin{aligned}
&\mathit{item} :: \mathit{Parser}\ \mathit{Char}\\
&\mathit{item} = P\ (\lambda cs \to \textbf{case}\ cs\ \textbf{of}\ [\,] \quad\quad \to [\,]\\
&\qquad\qquad\qquad\qquad\qquad\quad (c : cs) \to [(c, cs)])
\end{aligned}$$

Alternatives are represented by a deterministic choice operator ($\uplus$), 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*.

$$\textbf{data}\ \mathit{Bit} = \mathit{Zero} \mid \mathit{One}$$

$$\begin{aligned}
&\mathit{bit} :: \mathit{Parser}\ \mathit{Bit}\\
&\mathit{bit} = \textbf{do}\ c \leftarrow \mathit{item}\\
&\qquad\quad\ \ \textbf{case}\ c\ \textbf{of}\ \texttt{'0'} \to \mathit{return}\ \mathit{Zero}\\
&\qquad\qquad\qquad\quad\ \ \texttt{'1'} \to \mathit{return}\ \mathit{One}\\
&\qquad\qquad\qquad\quad\ \ \_ \quad \to \mathit{pzero}
\end{aligned}$$

$$
\begin{array}{l}
\mathit{bitstring} :: \mathit{Parser}\ ([\mathit{Bit}], \mathit{Bit}) \\
\mathit{bitstring} = (\mathbf{do}\ b \quad\ \ \leftarrow \mathit{bit} \\
\qquad\qquad\qquad\quad (bs, s) \leftarrow \mathit{bitstring} \\
\qquad\qquad\qquad\quad \mathit{return}\ (b : bs, b \oplus s)) \\
\qquad\qquad\quad\ \uplus\ \ \mathit{return}\ ([\,], \mathit{Zero})
\end{array}
$$

Now, we implement the transformation function:

$$
\begin{array}{l}
\mathit{transform} :: ([\mathit{Bit}], \mathit{Bit}) \rightarrow [\mathit{Bit}] \\
\mathit{transform}\ ([\,], \_) \qquad = [\,] \\
\mathit{transform}\ (b : bs, s) = (b \oplus s) : \mathit{transform}\ (bs, s)
\end{array}
$$

In summary, our bit string transformer consists of:

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

Regarding the above solution, we may notice that function *bitstring* constructs an intermediate list of bits (variable bs), that is later consumed by function *transform* in order to produce the desired result (the transformed list of bits). The construction of this intermediate structure may result in inefficiency of the presented program. We, therefore, would like to eliminate it; following the same strategy that we have used in the previous chapter, such elimination will be achieved by applying a specific fusion law to *shift*. The fusion law to be used is a law in the style of shortcut fusion, similar to the ones conceived in chapter 2 for the derivation of purely functional programs, but with the difference that now we are dealing with monadic functions. In this section, we present the specific instance of our law for lists (which is the type of the intermediate structure used in our running example), and in section 3.4 we show that the law is an instance of a generic one 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 our case) are expressed in terms of certain program schemes. In our law, we also require the consumer to be given by a structural recursive definition, but in terms of a *pfold*, which admits as input an additional constant parameter to be used along the recursive calls. The generic definition of *pfold* has been presented in section 2.3.3, and the specific case of *pfold* for lists, that we will use in the calculations presented in this section, has been given in sections 2.4.1 and 2.5.1.

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 a monadic computation. This is expressed by a function called $mbuildp_L$:

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

Having stated the forms required to the producer and the consumer it is now possible to formulate our laws and to use them in the calculation of a circular monadic program equivalent to the program *shift*.

We start by introducing a Law, similar to Law 2.5.1, but that applies to monadic program compositions. As Law 2.5.1, the new Law achieves program fusion and intermediate structure deforestation by calculating circular programs.

Law 3.2.1 (PFOLD/MBUILDP FOR LISTS) *Let m be a recursive monad.*

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

This law transforms a monadic composition, where the producer is an effectful function but the consumer may not necessarily 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 needs the use of a recursive binding within a monadic computation, and therefore requires the monad to be *recursive* (Erkök and Launchbury 2002). A recursive **do** (**mdo**-notation) is supported by *Haskell* for those monads that are declared an instance of the *MonadFix* class.

```
class Monad m ⇒ MonadFix m where
  mfix :: (a → m a) → m a
```

The monadic fixed-point operator *mfix* needs to obey the following semantic laws[1]:

$$f\ \bot = \bot \quad \Leftrightarrow \quad mfix\ f = \bot$$

$$mfix\ (return \circ h) \quad = \quad return\ (fix\ h) \tag{3.1}$$

$$\begin{aligned} mfix\ (\lambda x \rightarrow q \ggg\!= \lambda y \rightarrow f\ x\ y) \quad &= \quad q \ggg\!= (\lambda y \rightarrow mfix\ (\lambda x \rightarrow f\ x\ y)), \\ &\quad\ \ \textit{if } x \textit{ does not appear free in } q \end{aligned}$$

The parsing monad presented earlier can be declared an instance of the *MonadFix* class, for example, as follows:

```
instance MonadFix Parser where
  mfix f = P (λcs → mfix_L (λ~(x, y) → parse (f x) cs))
    where
      mfix_L f = case fix (f ∘ head) of
                   []      → []
                   (x : _) → x : mfix_L (tail ∘ f)
```

To see Law 3.2.1 in action, we write *transform* and *bitstring* in terms of

[1] *fix* is the usual fixed-point operator for pure functions $fix :: (a \rightarrow a) \rightarrow a$, $fix\ f = f\ (fix\ f)$.

$pfold_L$ and $mbuildp_L$, respectively:

$$
\begin{aligned}
&\mathit{transform} = pfold_L\ (hnil, hcons)\\
&\quad \textbf{where}\ hnil\ \_ = [\,]\\
&\qquad\qquad hcons\ ((b, r), s) = (b \oplus s) : r
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{bitstring} = mbuildp_L\ g\\
&\quad \textbf{where}\ g\ (nil, cons) = (\textbf{do}\ b \leftarrow bit\\
&\qquad\qquad\qquad\qquad\qquad\quad (bs, s) \leftarrow g\ (nil, cons)\\
&\qquad\qquad\qquad\qquad\qquad\quad return\ (cons\ (b, bs), b \oplus s)\\
&\qquad\qquad\qquad\qquad\qquad\ \uplus\ return\ (nil, Zero))
\end{aligned}
$$

Then, by applying the law we obtain:

$$
\begin{aligned}
&\mathit{shift} = \textbf{mdo}\ (bs, \boxed{s}) \leftarrow g\ ([\,], \lambda(b, r) \rightarrow (b \oplus \boxed{s}) : r)\\
&\qquad\qquad return\ bs
\end{aligned}
$$

Inlining, we get the following circular monadic program:

$$
\begin{aligned}
&\mathit{shift} = \textbf{mdo}\ (bs, \boxed{s}) \leftarrow \textbf{let}\ gk = (\textbf{do}\ b \leftarrow bit\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad (bs', s') \leftarrow gk\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad return\ ((b \oplus \boxed{s}) : bs', b \oplus s')\\
&\qquad\qquad\qquad\qquad\qquad\qquad\ \uplus\ return\ ([\,], Zero))\\
&\qquad\qquad\qquad\qquad\qquad \textbf{in}\ gk\\
&\qquad\qquad return\ bs
\end{aligned}
$$

The above program avoids the construction of the intermediate list of bits, by introducing a circular definition. Indeed, we may notice that s (the modulo 2 sum of an input sequence of bits) is used (in $b \oplus s$) in the function call to gk. However, s is also a result of that same call, hence the circularity.

In this section and in the previous one, we have calculated a circular program equivalent to the original *shift* program. This program consists of the composition of an effectful producer and a consumer given by a pure function. In the next section, we study fusion of monadic programs where both the consumer and the producer functions are effectful.

3.3 Algol 68 scope rules

In this section we consider an improvement on the semantic analyzer for the Algol 68 scope rules that we have been studying in this thesis.

We are still interested in developing a semantic function that analyzes a sequence of instructions and computes a list containing the variable identifiers of the instructions which do not obey to the scope rules of the Algol 68 language. However, when such an instruction is found, we now also want to output an error message explaining the programming error encountered, in order to make it easier to detect which identifiers are incorrect.

So, for example, if we consider the sentence presented earlier:

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

we still want to compute the list $[\,w, x]$: at the inner level, the use of variable w has no binding occurrence at all and the variable x has been declared twice, at the outer level. But, as a side-effect of computing this list, the following error messages must also be displayed:

Duplicate: decl x
Missing: decl w

The error messages must be displayed in a certain order: errors resulting from duplicate declarations are displayed first and only then we show the errors that result from missing declarations. Errors of the same type must be displayed in order of appearance. Furthermore, the errors occurring in nested blocks are displayed only after the errors occurring in the outer ones.

The original *semantics* program needs to be changed in order to accommodate the side-effect computations:

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

The function *duplicate* detects duplicate variable declarations by collecting all the declarations occurring in a program. It is now a monadic function since it needs to output error messages resulting from the errors it detects. The definition of this function, that closely follows the effect-free implementation presented in section 2.5, is as follows[2]:

```
$duplicate :: Int \rightarrow Env \rightarrow Prog \rightarrow IO\ (Prog_2, Env)$
$duplicate\ lev\ ds\ [\,] = return\ ([\,], ds)$

$duplicate\ lev\ ds\ (Use\ var : its)$
  $= \mathbf{do}\ (its_2, ds') \leftarrow duplicate\ lev\ ds\ its$
        $return\ (Use_2\ var : its_2, ds')$

$duplicate\ lev\ ds\ (Decl\ var : its)$
  $= \mathbf{if}\ ((var, lev) \in ds)$
      $\mathbf{then\ do}\ put\ (\texttt{"Duplicate: decl "} +\!\!+\ var)$
               $(its_2, ds') \leftarrow duplicate\ lev\ ((var, lev) : ds)\ its$
               $return\ (Dupl_2\ var : its_2, ds')$
      $\mathbf{else}\ duplicate\ lev\ ((var, lev) : ds)\ its$

$duplicate\ lev\ ds\ (Block\ nested : its)$
  $= \mathbf{do}\ (its_2, ds') \leftarrow duplicate\ lev\ ds\ its$
        $return\ (Block_2\ (lev + 1, nested) : its_2, ds')$
```

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 has been transformed into a monadic function as it also outputs error messages:

```
$missing :: (Prog_2, Env) \rightarrow IO\ [Var]$
$missing\ ([\,], \_) = return\ [\,]$

$missing\ (Use_2\ var : its_2, env)$
  $= \mathbf{if}\ (var \in map\ \pi_1\ env)$
      $\mathbf{then}\ missing\ (its_2, env)$
```

[2]We abbreviate *putStrLn* as *put*.

```
        else do $put$ ("Missing: decl " $+\!\!+$ $var$)
                $errs \leftarrow missing\ (its_2, env)$
                $return\ (var : errs)$

$missing\ (Dupl_2\ var : its_2, env)$
    $=$ do $errs \leftarrow missing\ (its_2, env)$
           $return\ (var : errs)$

$missing\ (Block_2\ (lev, its) : its_2, env)$
    $=$ do $errs_2 \leftarrow missing\ (its_2, env)$
           $errs_1 \leftarrow$ do $(p_2, env_2) \leftarrow duplicate\ lev\ env\ its$
                        $missing\ (p_2, env_2)$

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

Now, we want to eliminate the intermediate structure of type $Prog_2$ generated by function *duplicate* and consumed by function *missing*. We may notice that, if we attempted to directly apply Law 3.2.1 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 3.2.1 but is able to fuse effectful functions.

Law 3.3.1 (Effectful pfold/mbuildp for lists) *Let m be a recursive monad.*

```
    do $(xs, z) \leftarrow mbuildp_L\ g\ c$
       $pfold_L\ (hnil, hcons)\ (xs, z)$
$=$
    mdo $(m, \boxed{z}) \leftarrow$ let $knil = hnil\ \boxed{z}$
                             $kcons\ (x, y) = hcons\ ((x, y), \boxed{z})$
                         in  $g\ (knil, kcons)\ c$
        $m$
```

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

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

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 3.4 when we will define the generic formulation of the laws. However, in section 3.2 it was defined as a value of type $m\ ([a], z)$ because that form is more appropriate for writing monadic parsers.

For the present example we do not need to provide the instance of the *MonadFix* class for the *IO* monad as it is automatically provided by the Glasgow Haskell Compiler (GHC), which is the reference compiler we are using.

Now, if we write the monadic version of function *missing* in terms of $pfold_L$,

```
missing = pfold_L (hnil, hcons)
  where hnil _ = return []

        hcons ((Use_2 var, m_errs), env)
          = if (var ∈ map π_1 env)
              then m_errs
              else do put ("Missing: decl " ++ var)
                      errs ← m_errs
                      return (var : errs)

        hcons ((Dupl_2 var, m_errs), env)
          = do errs ← m_errs
               return (var : errs)

        hcons ((Block_2 (lev, its), m_errs), env)
          = do errs_2 ← m_errs
               errs_1 ← do (p_2, env_2) ← duplicate lev env its
```

```
                        $missing\ (p_2, env_2)$
                $return\ (errs_1 +\!\!+ errs_2)$
```

and the monadic version of *duplicate* in terms of $mbuildp_L$,

```
$duplicate\ lev\ ds = mbuildp_L\ (g\ lev\ ds)$
    **where** $g\ lev\ ds\ (nil, cons)\ [\,]$
              $= return\ (nil, ds)$

          $g\ lev\ ds\ (nil, cons)\ (Use\ var : its)$
              $=$ **do** $(its_2, ds') \leftarrow g\ lev\ ds\ (nil, cons)\ its$
                    $return\ (cons\ (Use_2\ var, its_2), ds')$

          $g\ lev\ ds\ (nil, cons)\ (Decl\ var : its)$
              $=$ **if** $((var, lev) \in ds)$
                **then do** $put$ ("Duplicate: decl " $+\!\!+\ var$)
                         $(its_2, ds') \leftarrow g\ lev\ ((var, lev) : ds)\ (nil, cons)\ its$
                         $return\ (cons\ (Dupl_2\ var, its_2), ds')$
                **else** $g\ lev\ ((var, lev) : ds)\ (nil, cons)\ its$

          $g\ lev\ ds\ (nil, cons)\ (Block\ nested : its)$
              $=$ **do** $(its_2, ds') \leftarrow g\ lev\ ds\ (nil, cons)\ its$
                    $return\ (cons\ (Block_2\ (lev + 1, nested), its_2), ds')$
```

we can apply Law 3.3.1 to *semantics* obtaining a deforested circular definition, which, when inlined, gives the following definition:

```
$semantics\ p$
    $=$ **mdo** $(m_{errs}, \boxed{\text{env}})$
            $\leftarrow$ **let** $gk\ lev\ ds\ [\,] = return\ (return\ [\,], ds)$

                $gk\ lev\ ds\ (Use\ var : its)$
                    $=$ **do** $(its_2, ds') \leftarrow gk\ lev\ ds\ its$
                          $return$ (**if** $(var \in map\ \pi_1\ \boxed{\text{env}})$)
                            **then** $its_2$
                            **else**
```

```
                                  do put ("Missing:decl" ++ var)
                                     errs ← its_2
                                     return (var : errs), ds')

              gk lev ds (Decl var : its)
                = if ((var, lev) ∈ ds)
                  then do put ("Duplicate:decl" ++ var)
                          (its_2, ds') ← gk lev
                                            ((var, lev) : ds)
                                            its
                          return (do errs ← its_2
                                     return (var : errs), ds')
                  else gk lev ((var, lev) : ds) its

              gk lev ds (Block nested : its)
                = do (its_2, ds') ← gk lev ds its
                     return
                       (do errs_2 ← its_2
                           errs_1
                             ← do (p_2, env_2)
                                     ← duplicate (lev + 1)
                                                 [env]
                                                 nested
                                  missing (p_2, env_2)
                           return (errs_1 ++ errs_2), ds')

            in gk 0 [] p
        m_errs
```

The calculations performed so far made it possible to eliminate some of the intermediate structures that were constructed by the original *semantics* program. We may notice, however, that an intermediate data structure is still constructed. Indeed, when traversing nested blocks, the calculated program defines:

```
$gk\ lev\ ds\ (Block\ nested : its)$
    $= \textbf{do}\ (its_2, ds') \leftarrow gk\ lev\ ds\ its$
          $return\ (\textbf{do}\ errs_2 \leftarrow its_2$
                    $errs_1 \leftarrow \textbf{do}\ (p_2, env_2)$
                                $\leftarrow duplicate\ (lev + 1)\ \boxed{\text{env}}\ nested$
                                  $missing\ (p_2, env_2)$

                    $return\ (errs_1 +\!\!+\ errs_2), ds')$
```

In order to deforest this intermediate structure as well, we may apply Law 3.3.1 again. Notice that the construction of such intermediate structure serves the purpose of gluing a composition between functions *missing* and *duplicate*, and that these functions have already been expressed in terms of the program schemes required by that law.

Additionally, we may apply the same strategy that we have applied in section 2.5.1 in order to use the definition of function *gk* to unify the two circular programs we obtain.

We finally obtain the program:

```
$semantics\ p$
    $= \textbf{mdo}\ (m_{errs}, \boxed{\text{env}})$
          $\leftarrow \textbf{let}\ gk\ lev\ ds\ env\ [\,] = return\ (return\ [\,], ds)$

                $gk\ lev\ ds\ env\ (Use\ var : its)$
                    $= \textbf{do}\ (its_2, ds') \leftarrow gk\ lev\ ds\ env\ its$
                          $return\ (\textbf{if}\ (var \in map\ \pi_1\ env)$
                                $\textbf{then}\ its_2$
                                $\textbf{else}$
                                  $\textbf{do}\ put$ ("Missing:decl"
                                          $+\!\!+\ var)$
                                      $errs \leftarrow its_2$
                                      $return\ (var : errs), ds')$

                $gk\ lev\ ds\ env\ (Decl\ var : its)$
                    $= \textbf{if}\ ((var, lev) \in ds)$
```

```
                          then do put ("Duplicate:decl" ++ var)
                                  (its_2, ds') ← gk lev ((var, lev) : ds)
                                                            env its
                                  return (do errs ← its_2
                                                  return (var : errs), ds')
                          else gk lev ((var, lev) : ds) env its

                    gk lev ds env (Block nested : its)
                       = do (its_2, ds') ← gk lev ds env its
                            return
                               (do errs_2 ← its_2
                                   errs_1
                                      ← mdo (m_errs_1, $\boxed{\text{env}_2}$)
                                                ← gk lev ds
                                                      $\boxed{\text{env}_2}$ nested
                                            m_errs_1
                                   return (errs_1 ++ errs_2), ds')

                in gk 0 [] $\boxed{\text{env}}$ p

            m_errs
```

Regarding the above program we may notice that it does not construct the intermediate *Prog*$_2$ structure that was used in the original *semantics* program to glue the functions *duplicate* and *missing* together. The deforestation of that structure was achieved introducing two circular definitions. The first one makes it possible to use the global environment of an input *Block* program while it is still being constructed. This is achieved with the circular definition on variable *env*. The second circular definition, defined over variable *env*$_2$, makes the environment of every nested sequence of *Block* instructions available while it is still under construction.

The calculated *semantics* program, as expected, produces the same results as the *semantics* program we started with. Furthermore, the monadic computations that come as side-effects of computing such results are also the same, for both versions of *semantics* and the same input *Block* program.

3.4 Calculating circular programs, generically

In this section, we show that the definition of Laws 3.2.1 and 3.3.1, presented in the previous sections, are instances of generic definitions valid for a wide class of data types and monads.

3.4.1 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 (Manzino and Pardo 2008; Ghani and Johann 2008). This extension is based on an extended form of build: Given a functor F (signature of a datatype) and another functor N, we can define

$$\begin{aligned}&\mathit{build}_N :: (\forall\ a\ .\ (F\ a \rightarrow a) \rightarrow c \rightarrow N\ a) \rightarrow c \rightarrow N\ \mu F\\&\mathit{build}_N\ g = g\ \mathit{in}_F\end{aligned}$$

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

Using build_N it is possible to state an extended form of shortcut fusion (see (Manzino and Pardo 2008; Ghani and Johann 2008) for a proof):

Law 3.4.1 (EXTENDED FOLD/BUILD) *For strict h and strictness preserving* N[3]

$$\mathit{map}_N\ (\mathit{fold}\ h) \circ \mathit{build}_N\ g = g\ h$$

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

$$\begin{aligned}\mathit{buildp}_N\ &:: (\forall\ a\ .\ (F\ a \rightarrow a) \rightarrow c \rightarrow N\ (a, z))\\&\quad \rightarrow c \rightarrow N\ (\mu F, z)\\\mathit{buildp}_N\ g &= g\ \mathit{in}_F\end{aligned}$$

[3]The strictness-preserving assumption on the functor means that map_N preserves strict functions, *i.e.*, if f is strict, then so is $\mathit{map}_N\ f$.

with the following shortcut fusion law:

Law 3.4.2 (EXTENDED FOLD/BUILDP) *For strict h and strictness-preserving N,*

$$map_N\ (fold\ h \times id) \circ buildp_N\ g = g\ h$$

Proof By considering $N'\ a = N\ (a, z)$, we have that $buildp_N\ g = build_{N'}\ g$ and $map_{N'}\ f = map_N\ (f \times id)$. Then, the left-hand side of the equation can be rewritten as: $map_{N'}\ (fold\ h) \circ build_{N'}\ g$. Finally, we apply Law 3.4.1. □

3.4.2 Monadic shortcut fusion

We are interested in studying Law 3.4.2 for the case when the functor N is the composition of a monad m with a product: For some type z,

$$N\ a = m\ (a, z) \qquad \text{and} \qquad map_N\ f = mmap\ (f \times id)$$

where

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

is the map function for the monad m. The producer then corresponds to a monadic version of buildp:

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

A monadic shortcut fusion law can be directly obtained as an instance of Law 3.4.2. We unfold the definition of $mmap$ so that we get a formulation in terms of **do**-notation:

Law 3.4.3 (FOLD/MBUILDP) *For strict k and strictness preserving $mmap$,*

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

This is a version for *mbuildp* of the shortcut fusion law introduced by Manzino and Pardo (2008).

Using this law we can state a first monadic extension of the pfold/buildp rule, that we have presented in section 2.4, and that applies to effect-free programs only. This extension provides the generic definition of Law 3.2.1, that was used to calculate a circular version of the bit string transformer.

Law 3.4.4 (PFOLD/MBUILDP) *Let m be a recursive monad with strictness-preserving mmap. For h with components* $(h_1, \ldots, h_n)$ *and strict,*

$$\begin{array}{l} \quad \textbf{do}\ \{(t,z) \leftarrow \mathit{mbuildp}\ g\ c; \mathit{return}\ (\mathit{pfold}\ h\ (t,z))\} \\ = \\ \quad \textbf{mdo}\{(v,\boxed{z}) \leftarrow \textbf{let}\ k_i\ \bar{x} = h_i\ (\bar{x},\boxed{z})\ \textbf{in}\ g\ k\ c; \mathit{return}\ v\} \end{array}$$

Proof

$$\begin{array}{ll} & \textbf{do}\ \{(t,z) \leftarrow \mathit{mbuildp}\ g\ c; \mathit{return}\ (\mathit{pfold}\ h\ (t,z))\} \\ = & \{\ (2.9)\ \} \\ & \textbf{do}\ (t,z) \leftarrow \mathit{mbuildp}\ g\ c \\ & \quad \textbf{let}\ k_i\ \bar{x} = h_i\ (\bar{x},z)\ \textbf{in}\ \mathit{return}\ (\mathit{fold}\ k\ t) \\ = & \{\ \text{definition of}\ \pi_1\ \} \\ & \textbf{do}\ (t,z) \leftarrow \mathit{mbuildp}\ g\ c \\ & \quad \textbf{let}\ k_i\ \bar{x} = h_i\ (\bar{x},z)\ \textbf{in}\ \mathit{return}\ (\pi_1\ (\mathit{fold}\ k\ t, z)) \\ = & \\ & \textbf{do}\ (t,z)\ \leftarrow \mathit{mbuildp}\ g\ c \\ & \quad (v,z') \leftarrow \textbf{let}\ k_i\ \bar{x} = h_i\ (\bar{x},z)\ \textbf{in}\ \mathit{return}\ (\mathit{fold}\ k\ t, z) \\ & \quad \mathit{return}\ v \\ = & \{\ \text{Law 3.4.5}\ \} \\ & \textbf{mdo}\ (t,z) \leftarrow \mathit{mbuildp}\ g\ c \\ & \qquad (v,\boxed{z'}) \leftarrow \textbf{let}\ k_i\ \bar{x} = h_i\ (\bar{x},\boxed{z'})\ \textbf{in}\ \mathit{return}\ (\mathit{fold}\ k\ t, z) \end{array}$$

$$\quad \textit{return } v$$

$$= \quad \{ \text{ Law 3.4.3 } \}$$

$$\textbf{mdo}\{ (v, \boxed{z}) \leftarrow \textbf{let } k_i \; \bar{x} = h_i \; (\bar{x}, \boxed{z}) \textbf{ in } g \; k \; c; \textit{return } v \} \qquad \Box$$

The introduction of a circular definition requires the monad to be *recursive* (Erkök and Launchbury 2002) as it requires the use of a circular binding within a monadic computation. In *Haskell* terms, this can be expressed using the **mdo**-notation provided that the monad is an instance of the *MonadFix* class. The definition of **mdo** in terms of **do** is as follows.

$$\textbf{mdo}\{ x \leftarrow e; e' \} = \textbf{do } \{ x \leftarrow \textit{mfix } (\lambda x \rightarrow e); e' \} \tag{3.2}$$

The circularity introduced in Law 3.4.4 is safe and therefore computations can be ordered under lazy evaluation. In order to prove this, we first need to prove the following property.

$$\forall \; z \; f \; . \; \textit{fix } (\lambda(v, z') \rightarrow (f \; z', z)) = (f \; z, z) \tag{3.3}$$

Proof

We prove this property by fixed-point induction with admissible predicate

$$P \; (x, y) = (x, y) \; \neq \; \bot \Rightarrow (x, y) = (f \; z, z)$$

Let us define $\phi \; (x, y) = (f \; y, z)$. The proof needs to consider two cases:

Base case

$P \; (\bot)$ is trivially true since the antecedent of P fails.

Inductive case

Assume that $P \; (x, y)$ holds. The inductive hypothesis is, therefore, that $(x, y) = (f \; z, z)$. We will prove that $P \; (\phi \; (x, y))$ holds.

$$
\begin{array}{ll}
 & \phi\ (x, y) \\
= & \quad \{\ \text{definition of } \phi\ \} \\
 & (f\ y, z) \\
= & \quad \{\ \text{inductive hypothesis}\ \} \\
 & (f\ z, z) \qquad \square
\end{array}
$$

Now, we can state and prove the following law, that guarantees the safe introduction of circular definitions in Law 3.4.4.

Law 3.4.5 (Monadic Local Recursion) *Let m be a recursive monad with strictness-preserving mmap. For h with components $(h_1, \ldots, h_n)$ and strict,*

$$
\begin{array}{l}
\mathbf{do}\ (v, z') \leftarrow \mathbf{let}\ k_i\ \bar{x} = h_i\ (\bar{x}, z)\ \mathbf{in}\ \mathit{return}\ (\mathit{fold}\ k\ t, z) \\
\qquad \mathit{return}\ v \\
= \\
\mathbf{mdo}\ (v, \boxed{z'}) \leftarrow \mathbf{let}\ k_i\ \bar{x} = h_i\ (\bar{x}, \boxed{z'})\ \mathbf{in}\ \mathit{return}\ (\mathit{fold}\ k\ t, z) \\
\qquad \mathit{return}\ v
\end{array}
$$

Proof

$$
\begin{array}{ll}
 & \mathbf{mdo}\ (v, \boxed{z'}) \leftarrow \mathbf{let}\ k_i\ \bar{x} = h_i\ (\bar{x}, \boxed{z'})\ \mathbf{in}\ \mathit{return}\ (\mathit{fold}\ k\ t, z) \\
 & \qquad \mathit{return}\ v \\
= & \quad \{\ (3.2)\ \} \\
 & \mathbf{do}\ (v, z') \leftarrow \mathit{mfix}\ (\lambda(v, z') \rightarrow \mathbf{let}\ k_i\ \bar{x} = h_i\ (\bar{x}, z') \\
 & \qquad\qquad\qquad\qquad\qquad\qquad \mathbf{in}\ \mathit{return}\ (\mathit{fold}\ k\ t, z)) \\
 & \quad \mathit{return}\ v \\
= & \quad \{\ (3.1)\ \} \\
 & \mathbf{do}\ (v, z') \leftarrow \mathit{return}\ (\mathit{fix}\ (\lambda(v, z') \rightarrow \mathbf{let}\ k_i\ \bar{x} = h_i\ (\bar{x}, z') \\
 & \qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{in}\ (\mathit{fold}\ k\ t, z))) \\
 & \quad \mathit{return}\ v
\end{array}
$$

$$
\begin{array}{l}
= \quad \{ \;(3.3)\; \} \\
\qquad \mathbf{do}\; (v, z') \leftarrow \mathit{return}\; (\mathbf{let}\; k_i\; \bar{x} = h_i\; (\bar{x}, z)\; \mathbf{in}\; (\mathit{fold}\; k\; t, z)) \\
\qquad\quad \mathit{return}\; v \\
= \\
\qquad \mathbf{do}\; (v, z') \leftarrow \mathbf{let}\; k_i\; \bar{x} = h_i\; (\bar{x}, z)\; \mathbf{in}\; \mathit{return}\; (\mathit{fold}\; k\; t, z) \\
\qquad\quad \mathit{return}\; v \qquad\qquad\qquad\qquad\qquad\qquad \Box
\end{array}
$$

Laws 3.4.3 and 3.4.4 handle monadic producers and purely functional consumers. When the consumer is also an effectful function, it is possible to state two other fusion laws. The formulation of these laws follows the approach presented by Chitil (2000) and Ghani and Johann (2008).

Law 3.4.6 (EFFECTFUL FOLD/MBUILDP) *For strict* $k :: F\;(m\;\;a) \rightarrow m\;\;a$ *and strictness preserving mmap,*

$$
\begin{array}{l}
\mathbf{do}\; \{ (t, z) \leftarrow \mathit{mbuildp}\; g\; c; v \leftarrow \mathit{fold}\; k\; t; \mathit{return}\; (v, z) \} \\
= \\
\mathbf{do}\; \{ (m, z) \leftarrow g\; k\; c; v \leftarrow m; \mathit{return}\; (v, z) \}
\end{array}
$$

Proof

$$
\begin{array}{l}
\mathbf{do}\; \{ (t, z) \leftarrow \mathit{mbuildp}\; g\; c; v \leftarrow \mathit{fold}\; k\; t; \mathit{return}\; (v, z) \} \\
\quad = \mathbf{do}\; (t, z) \;\;\leftarrow \mathit{mbuildp}\; g\; c \\
\qquad\quad\;\; (m, \_) \leftarrow \mathit{return}\; (\mathit{fold}\; k\; t, z) \\
\qquad\quad\;\; v \qquad\; \leftarrow m \\
\qquad\quad\;\; \mathit{return}\; (v, z) \\
\quad = \mathbf{do}\; (m, z) \leftarrow \mathbf{do}\; (t, z) \leftarrow \mathit{mbuildp}\; g\; c \\
\qquad\qquad\qquad\qquad\quad\; \mathit{return}\; (\mathit{fold}\; k\; t, z) \\
\qquad\quad\;\; v \qquad\; \leftarrow m \\
\qquad\quad\;\; \mathit{return}\; (v, z) \\
\quad = \mathbf{do}\; \{ (m, z) \leftarrow g\; k\; c; v \leftarrow m; \mathit{return}\; (v, z) \} \qquad \Box
\end{array}
$$

Using this law we can now state a shortcut fusion law for the derivation of monadic circular programs in those cases where both the producer and consumer are effectful functions. Again, like in Law 3.4.4, the monad is required to be recursive because of the introduction of a recursive binding within the monadic computation.

Law 3.4.7 (EFFECTFUL PFOLD/MBUILDP) *Let m be a recursive monad with strictness-preserving mmap. For* $h :: (F\ (m\ a), z) \rightarrow m\ a$ *with components* $(h_1, \ldots, h_n)$ *and strict,*

$$\begin{aligned}&\mathbf{do}\ \{(t,z) \leftarrow \mathit{mbuildp}\ g\ c; \mathit{pfold}\ h\ (t,z)\}\\ =\ &\\ &\mathbf{mdo}\{(m,\boxed{z}) \leftarrow \mathbf{let}\ k_i\ \bar{x} = h_i\ (\bar{x},\boxed{z})\ \mathbf{in}\ g\ k\ c; m\}\end{aligned}$$

Proof

$$\begin{aligned}&\mathbf{do}\ \{(t,z) \leftarrow \mathit{mbuildp}\ g\ c; \mathit{pfold}\ h\ (t,z)\}\\ =\ &\\ &\mathbf{do}\ (t,z) \leftarrow \mathit{mbuildp}\ g\ c\\ &\qquad m \quad\ \leftarrow \mathit{return}\ (\mathit{pfold}\ h\ (t,z))\\ &\qquad m\\ =\ &\quad \{\ \text{Law 3.4.4}\ \}\\ &\mathbf{mdo}\{(m,\boxed{z}) \leftarrow \mathbf{let}\ k_i\ \bar{x} = h_i\ (\bar{x},\boxed{z})\ \mathbf{in}\ g\ k\ c; m\} \qquad \Box\end{aligned}$$

Law 3.3.1 is the specific instance of Law 3.4.7, for the case where the intermediate structure is a list.

3.5 Conclusions

In this chapter, we have presented shortcut fusion rules for the derivation of circular monadic programs. Indeed, using an extension to shortcut fusion that captures the cases when the intermediate structures are generated as part of another structure given by a functor, we obtained shortcut fusion rules that transform compositions of monadic programs into monadic circular programs. This extends to monadic programs the work we have presented, for pure programs, in the previous chapter.

The rules we present are generic, as they can be instantiated for a wide class of algebraic data types and monads. We have also shown two example applications which demonstrate the practical interest of the rules.

Like for non-monadic programs, it is possible to obtain a prototype implementation of the rules using the rewrite rules (RULES pragma) of the Glasgow Haskell Compiler (GHC). The current implementation of the rewrite rules, however, does not allow matching on **do** blocks. Indeed, the left hand side of a rule must have the form $f\ x\ ...\ z$, where f is a function (and not a language construction like **do**). So, in order to implement our rules in GHC, we first need to encapsulate the parts to be matched on in functions (using, for example, the *mmap* function).

In the next chapter we will study an alternative solution to achieve intermediate structure deforestation for programs such as the ones we have been considering, by transforming them into higher-order programs.

Chapter 4

Calculation of Higher-order Programs

Summary

In this chapter, we present shortcut deforestation techniques to calculate higher-order programs. The techniques we propose have the same applicability scope and goals as the techniques we presented in the previous chapters. However, the elimination of the intermediate structures in the programs we transform will be achieved by introducing higher-order definitions instead of circular ones. We consider both pure and effectful programs.

4.1 Introduction

In the previous chapters we have studied calculational techniques to achieve intermediate structure deforestation in programs such as $\mathit{prog} = \mathit{cons} \circ \mathit{prod}$, where $\mathit{prod} :: a \rightarrow (t, z)$ and $\mathit{cons} :: (t, z) \rightarrow b$. Deforestation was achieved by transforming *prog* into a circular program, and was studied in the context of both pure programs, in chapter 2, and of monadic ones, in chapter 3.

An alternative solution to achieve intermediate structure deforestation for programs such as *prog* is to transform them into higher-order programs using a well-known program transformation technique called lambda-abstraction

(Pettorossi and Skowron 1987). In our particular context, the idea of the transformation is to derive a new function $prog' :: a \to (z \to b, z)$, which returns a function and the same value of type z that would be generated by *prod*, such that $prog\ a = f\ z$ **where** $(f, z) = prog'\ a$. Based on this idea, Voigtländer (2008) introduced a shortcut fusion rule for the derivation of pure, higher-order programs from compositions like *prog* when lists are the intermediate structure. In this chapter, we extend this result in two ways. First, we present a generic formulation of the shortcut fusion rule for the derivation of pure higher-order programs that can be applied to a wide range of datatypes as intermediate data structures. Second, we extend the generic rule to the context of monadic programming, obtaining shortcut fusion rules for the derivation of monadic higher-order programs. The rules we present in this chapter consider three cases: the case where the producer and the consumer are both pure functions, 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.

Obtaining higher-order programs is interesting since their execution is not restricted to a lazy evaluation setting as it happens with the execution of the circular ones. Furthermore, experimental benchmarks that we have conducted and that we present in chapter 6 show that the performance of the higher-order programs derived from programs like *prog* is significantly better than the performance of their circular or original equivalents, both in terms of time and memory consumption.

This chapter is organized as follows. Section 4.2 introduces the *higher-order pfold/buildp* rule, that we will use to calculate higher-order versions of pure programs. In that section, we also show how that rule can be used to obtain a higher-order version of the *repmin* program. In section 4.3, we transform the semantic analyzer presented in section 2.5 into a higher-order program, by direct application of the new rule. In section 4.4, we calculate monadic higher-order programs: in sections 4.4.1 and 4.4.2 we calculate higher-order versions of the bit string transformer (see section 3.2) and of the improved (monadic) version of the semantic analyzer (see section 3.3), respectively. The generic constructions that give rise to the specific program schemes and laws presented in those examples are developed in section 4.4.3. Finally, in section 4.5, we conclude the chapter.

4.2 The higher-order pfold/buildp rule

In section 2.4, we have presented the generic formulation of a calculation rule for deriving circular programs. The rule applies to programs consisting of the composition of a *pfold* and a *buildp*, and that do not rely on monadic computations. There exists, however, an alternative way to transform compositions between *pfold* and *buildp*. Indeed, in this section we derive higher-order programs from such compositions, instead of the circular programs we derived before.

The alternative transformation presented in this section is based on the fact that every *pfold* can be expressed in terms of a higher-order fold: For $h :: (F\ a, z) \to a$,

$$\mathit{pfold}\ h = \mathit{apply} \circ (\mathit{fold}\ \varphi_h \times \mathit{id}) \tag{4.1}$$

where $\varphi_h :: F\ (z \to a) \to (z \to a)$ is given by

$$\varphi_h = \mathit{curry}\ (h \circ ((\mathit{map}_F\ \mathit{apply} \circ \tau^F) \vartriangle \pi_2))$$

and $\mathit{apply} :: (a \to b, a) \to b$ by $\mathit{apply}\ (f, x) = f\ x$. Therefore, $\mathit{fold}\ \varphi_h :: \mu F \to (z \to a)$ is the curried version of $\mathit{pfold}\ h$.

With this relationship at hand we can state the following shortcut fusion law, which is the instance to our context of a more general program transformation technique called lambda abstraction (Pettorossi and Skowron 1987). The specific case of this law when lists are the intermediate structure was recently introduced by Voigtländer (2008).

Law 4.2.1 (HIGHER-ORDER PFOLD/BUILDP) *For left-strict h,*[1]

$$\mathit{pfold}\ h \circ \mathit{buildp}\ g = \mathit{apply} \circ g\ \varphi_h$$

[1]By left-strict we mean strict on the first argument, that is, $h\ (\perp, z) = \perp$.

Proof

$$\begin{array}{cl}
& \mathit{pfold}\ h \circ \mathit{buildp}\ g \\
= & \quad \{\ (4.1)\ \} \\
& \mathit{apply} \circ (\mathit{fold}\ \varphi_h \times \mathit{id}) \circ \mathit{buildp}\ g \\
= & \quad \{\ \text{Law 2.3.2}\ \} \\
& \mathit{apply} \circ g\ \varphi_h \qquad\qquad \Box
\end{array}$$

Like in the derivation of circular programs, $g\ \varphi_h$ returns a pair, but now composed of a function of type $z \to a$ and an object of type z. The final result then corresponds to the application of the function to the object. That is,

$$\mathit{pfold}\ h\ (\mathit{buildp}\ g\ c) = \mathbf{let}\ (f, z) = g\ \varphi_h\ c\ \mathbf{in}\ f\ z$$

To see an example of the application of Law 4.2.1, we consider again the straightforward solution to the *repmin* problem:

$$\begin{array}{l}
\mathit{transform}\ t = \mathit{replace} \circ \mathit{tmint}\ \$\ t \\[1ex]
\mathit{replace} = \mathit{pfold}_T\ (\mathit{Leaf} \circ \pi_2, \mathit{Fork} \circ \pi_1) \\[1ex]
\mathit{tmint} = \mathit{buildp}_T\ g \\
\quad \mathbf{where}\ g\ (\mathit{leaf}, \mathit{fork})\ (\mathit{Leaf}\ n) = (\mathit{leaf}\ n, n) \\
\qquad\qquad g\ (\mathit{leaf}, \mathit{fork})\ (\mathit{Fork}\ (l, r)) = \mathbf{let}\ (l', n_1) = g\ (\mathit{leaf}, \mathit{fork})\ l \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad (r', n_2) = g\ (\mathit{leaf}, \mathit{fork})\ r \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathbf{in}\ (\mathit{fork}\ (l', r'), \mathit{min}\ n_1\ n_2)
\end{array}$$

In order to apply Law 4.2.1 to *transform*, we need the expression of the algebra of the higher-order fold which corresponds to the curried version of *replace*:

$$\begin{array}{l}
\mathit{replace}_{ho} :: \mathit{LeafTree} \to (\mathit{Int} \to \mathit{LeafTree}) \\
\mathit{replace}_{ho} = \mathit{fold}_T\ (\varphi_{h_1}, \varphi_{h_2}) \\
\quad \mathbf{where}\ \varphi_{h_1}\ \_ = \lambda z \to \mathit{Leaf}\ z \\
\qquad\qquad \varphi_{h_2}\ (l, r) = \lambda z \to \mathit{Fork}\ (l\ z, r\ z)
\end{array}$$

Then, by direct application of Law 4.2.1 to *transform*, we obtain:

$$transform = apply \circ g\ (\varphi_{h_1}, \varphi_{h_2})$$

Inlining the above definition, we obtain the higher-order solution to *repmin* that we had already presented in chapter 1 (page 9):

```
$transform\ t = nt\ m$
  $\mathbf{where}$
    $(nt, m) = repmin\ t$
    $repmin\ (Leaf\ n) = (\lambda z \rightarrow Leaf\ z, n)$
    $repmin\ (Fork\ (l, r)) = \mathbf{let}\ (l', n_1)\ = repmin\ l$
                                   $(r', n_2) = repmin\ r$
                              $\mathbf{in}\ (\lambda z \rightarrow Fork\ (l'\ z, r'\ z), min\ n_1\ n_2)$
```

The transformation of the straightforward *repmin* solution into a higher-order program can easily be implemented under GHC. Indeed, like we have done for the calculation of circular programs, in section 2.4, we may express Law 4.2.1, that can be used for calculating higher-order programs, in terms of rewrite rules. The specific RULES pragma that applies to programs such as *transform*, that construct an intermediate leaf tree, is as follows:

```
{-# RULES
  "ho_pfoldT/buildpT"
    $\forall\ c\ h_1\ h_2\ (g :: \forall\ a\ .\ (Int \rightarrow a, (a, a) \rightarrow a) \rightarrow b \rightarrow (a, z))\ .$
      $pfold_T\ (h_1, h_2)\ (buildp_T\ g\ c)$
        $= \mathbf{let}\ (f, z) = g\ (\varphi_{h_1}, \varphi_{h_2})\ c$
              $\varphi_{h_1}\ x = \lambda z \rightarrow h_1\ (x, z)$
              $\varphi_{h_2}\ (l, r) = \lambda z \rightarrow h_2\ ((l\ z, r\ z), z)$
          $\mathbf{in}\ f\ z$
  #-}
```

4.3 Calculating a higher-order program

In this section we study the application of Law 4.2.1 to a real example: the Algol 68 scope rules, that we have introduced in detail in section 2.5. In that section, we defined the semantic analyzer for the scope rules of Algol as follows:

$$semantics = missing \circ (duplicate\ 0\ [\,])$$

The definition we presented constructs an intermediate list of instructions, that we woud like to eliminate with fusion. For this purpose, we will now use the specific instance of Law 4.2.1 for the case where the intermediate structure is a list:

Law 4.3.1 (HIGHER-ORDER PFOLD/BUILDP FOR LISTS)

$$pfold_L\ (hnil, hcons) \circ buildp_L\ g = apply \circ g\ (\varphi_{hnil}, \varphi_{hcons})$$

where $(\varphi_{hnil}, \varphi_{hcons})$ is the algebra of the higher-order fold which corresponds to the curried version of $pfold_L\ (hnil, hcons)$.

In section 2.5, we have already expressed function *missing* in terms of $pfold_L$,

```
missing = pfold_L (hnil, hcons)
  where hnil _ = []

        hcons ((Use_2 var, errs), env)
          = if (var ∈ map π_1 env) then errs
                                    else var : errs

        hcons ((Dupl_2 var, errs), env)
          = var : errs

        hcons ((Block_2 (lev, its), errs), env)
          = let errs_1 = missing ∘ (duplicate lev env) $ its
            in errs_1 ++ errs
```

and function *duplicate* in terms of $buildp_L$.

$$
\begin{array}{l}
\mathit{duplicate}\ \mathit{lev}\ \mathit{ds} = \mathit{buildp}_L\ (g\ \mathit{lev}\ \mathit{ds}) \\
\quad \textbf{where}\ g\ \mathit{lev}\ \mathit{ds}\ (\mathit{nil}, \mathit{cons})\ [\,] = (\mathit{nil}, \mathit{ds}) \\[1ex]
\qquad\quad g\ \mathit{lev}\ \mathit{ds}\ (\mathit{nil}, \mathit{cons})\ (\mathit{Use}\ \mathit{var} : \mathit{its}) \\
\qquad\qquad = \textbf{let}\ (\mathit{its}_2, \mathit{ds}') = g\ \mathit{lev}\ \mathit{ds}\ (\mathit{nil}, \mathit{cons})\ \mathit{its} \\
\qquad\qquad\quad \textbf{in}\ \ (\mathit{cons}\ (\mathit{Use}_2\ \mathit{var}, \mathit{its}_2), \mathit{ds}') \\[1ex]
\qquad\quad g\ \mathit{lev}\ \mathit{ds}\ (\mathit{nil}, \mathit{cons})\ (\mathit{Decl}\ \mathit{var} : \mathit{its}) \\
\qquad\qquad = \textbf{let}\ (\mathit{its}_2, \mathit{ds}') = g\ \mathit{lev}\ ((\mathit{var}, \mathit{lev}) : \mathit{ds})\ (\mathit{nil}, \mathit{cons})\ \mathit{its} \\
\qquad\qquad\quad \textbf{in}\ \ \textbf{if}\ ((\mathit{var}, \mathit{lev}) \in \mathit{ds})\ \textbf{then}\ (\mathit{cons}\ (\mathit{Dupl}_2\ \mathit{var}, \mathit{its}_2), \mathit{ds}') \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{else}\ (\mathit{its}_2, \mathit{ds}') \\[1ex]
\qquad\quad g\ \mathit{lev}\ \mathit{ds}\ (\mathit{nil}, \mathit{cons})\ (\mathit{Block}\ \mathit{nested} : \mathit{its}) \\
\qquad\qquad = \textbf{let}\ (\mathit{its}_2, \mathit{ds}') = g\ \mathit{lev}\ \mathit{ds}\ (\mathit{nil}, \mathit{cons})\ \mathit{its} \\
\qquad\qquad\quad \textbf{in}\ \ (\mathit{cons}\ (\mathit{Block}_2\ (\mathit{lev} + 1, \mathit{nested}), \mathit{its}_2), \mathit{ds}')
\end{array}
$$

Therefore, in order to apply Law 4.3.1 to the *semantics* program, we now only need the expression of the algebra $(\varphi_{hnil}, \varphi_{hcons})$ of the curried version of *missing*:

$$
\begin{array}{l}
\mathit{missing}_{ho} = \mathit{fold}_L\ (\varphi_{hnil}, \varphi_{hcons}) \\
\quad \textbf{where}\ \varphi_{hnil} = \lambda\_ \rightarrow [\,] \\[1ex]
\qquad\quad \varphi_{hcons}\ (\mathit{Use}_2\ \mathit{var}, \mathit{ferrs}) \\
\qquad\qquad = \lambda \mathit{env} \rightarrow \textbf{if}\ (\mathit{var} \in \mathit{map}\ \pi_1\ \mathit{env})\ \textbf{then}\ \mathit{ferrs}\ \mathit{env} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{else}\ \mathit{var} : (\mathit{ferrs}\ \mathit{env}) \\[1ex]
\qquad\quad \varphi_{hcons}\ (\mathit{Dupl}_2\ \mathit{var}, \mathit{ferrs}) \\
\qquad\qquad = \lambda \mathit{env} \rightarrow \mathit{var} : (\mathit{ferrs}\ \mathit{env}) \\[1ex]
\qquad\quad \varphi_{hcons}\ (\mathit{Block}_2\ (\mathit{lev}, \mathit{its}), \mathit{ferrs}) \\
\qquad\qquad = \lambda \mathit{env} \rightarrow \textbf{let}\ \mathit{errs}_1 = \mathit{missing} \circ (\mathit{duplicate}\ \mathit{lev}\ \mathit{env})\ \$\ \mathit{its} \\
\qquad\qquad\qquad\qquad\quad \textbf{in}\ \mathit{errs}_1 \mathbin{+\!\!+} (\mathit{ferrs}\ \mathit{env})
\end{array}
$$

After inlining the definition that we calculate by directly applying Law 4.3.1 to the *semantics* program, we obtain the program presented in the next page.

```
$semantics\ p = ferrs\ env$
  **where** $(ferrs, env) = g_\varphi\ 0\ [\,]\ p$
        $g_\varphi\ lev\ ds\ [\,] = (\lambda env \to [\,], ds)$

        $g_\varphi\ lev\ ds\ (Use\ var : its)$
          $=$ **let** $(ferrs, ds') = g_\varphi\ lev\ ds\ its$
            **in** $(\lambda env \to$ **if** $var \in map\ \pi_1\ env$
                          **then** $ferrs\ env$
                          **else** $var : (ferrs\ env), ds')$

        $g_\varphi\ lev\ ds\ (Decl\ var : its)$
          $=$ **let** $(ferrs, ds') = g_\varphi\ lev\ ((var, lev) : ds)\ its$
            **in if** $((var, lev) \in ds)$
              **then** $(\lambda env \to var : (ferrs\ env), ds')$
              **else** $(ferrs, ds')$

        $g_\varphi\ lev\ ds\ (Block\ nested : its)$
          $=$ **let** $(ferrs_2, ds') = g_\varphi\ lev\ ds\ its$
            **in** $(\lambda env \to$ **let** $errs_1 = missing$
                                    $\circ\ (duplicate\ (lev + 1)$
                                                  $env)\ \$\ nested$
                          **in** $errs_1 +\!\!+\ ferrs_2\ env, ds')$
```

Notice that the first component of the result produced by the call $g_\varphi\ 0\ [\,]\ p$ is now a function, instead of a concrete value. When this function is applied to *env*, it produces the list of variables that do not obey to the semantic rules of the language. The program we have calculated is, therefore, a higher-order program.

Regarding the above program, we may notice that it maintains the construction of an intermediate structure. This situation already occurred in sections 2.5.1 and 3.3, when different fusion rules were applied to different versions of the *semantics* program. Again, an intermediate structure is constructed whenever a nested sequence of instructions is traversed, in the definition presented next.

$$
\begin{aligned}
& g_{\varphi}\ \mathit{lev}\ \mathit{ds}\ (\mathit{Block}\ \mathit{nested} : \mathit{its}) \\
& \quad = \mathbf{let}\ (\mathit{ferrs}_2, \mathit{ds}') = g_{\varphi}\ \mathit{lev}\ \mathit{ds}\ \mathit{its} \\
& \qquad \mathbf{in}\ (\lambda \mathit{env} \rightarrow \mathbf{let}\ \mathit{errs}_1 = \mathit{missing} \\
& \qquad\qquad\qquad\qquad\qquad\qquad \circ\ (\mathit{duplicate}\ (\mathit{lev} + 1) \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{env})\ \$\ \mathit{nested} \\
& \qquad\qquad\qquad\qquad \mathbf{in}\ \mathit{errs}_1 +\!\!+\ \mathit{ferrs}_2\ \mathit{env}, \mathit{ds}')
\end{aligned}
$$

The *missing* ∘ *duplicate* composition in the above definition, however, may be eliminated by direct application of Law 4.3.1. This is due to the fact that functions *missing* and *duplicate* have already been expressed in terms of the appropriate program schemes. We obtain:

$$
\begin{aligned}
& g_{\varphi}\ \mathit{lev}\ \mathit{ds}\ (\mathit{Block}\ \mathit{nested} : \mathit{its}) \\
& \quad = \mathbf{let}\ (\mathit{ferrs}_2, \mathit{ds}') = g_{\varphi}\ \mathit{lev}\ \mathit{ds}\ \mathit{its} \\
& \qquad \mathbf{in}\ (\lambda \mathit{env} \rightarrow \mathbf{let}\ (\mathit{ferrs}_1, \mathit{env}_1) = g_{\varphi}\ (\mathit{lev} + 1)\ \mathit{env}\ \mathit{nested} \\
& \qquad\qquad\qquad\qquad \mathbf{in}\ \mathit{ferrs}_1\ \mathit{env}_1 +\!\!+\ \mathit{ferrs}_2\ \mathit{env}, \mathit{ds}')
\end{aligned}
$$

The higher-order version of *semantics* that we calculate in this section, by applying Law 4.3.1, twice, to the original *semantics* program avoids the construction of any intermediate structure. Furthermore, in this program, the appropriate (local or global) environment is passed to the correct block of instructions. Notice that, in order for this to happen, it was not necessary to post-process the calculated program, as it was in sections 2.5.1 and 3.3. The execution of the higher-order *semantics* program is not restricted to a lazy execution setting. Recall that the intermediate structure free program that we calculated in section 2.5.1 may only be executed in a lazy setting: it holds two circular definitions.

4.4 Calculation of monadic higher-order programs

In the previous sections, we have studied the use of generic calculation rules in the derivation of higher-order programs. The rules we presented, however, do not consider monadic functional programs. In the same way that we did for circular program derivation, we now wish to extend higher-order fusion to monadic programs. This is the purpose of this section.

We start by reviewing the two monadic examples we presented in chapter 3: the bit string transformer (see section 3.2) and the improved version of the Algol 68 semantic analyzer that outputs error messages (see section 3.3). Later, we present the formal definition of the constructions that give rise to the specific laws presented in those examples.

4.4.1 Bit string transformation

To illustrate our technique for deriving monadic higher-order programs, we first review the bit string transformer that we have defined before. Recall that we wanted 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 considered that the input sequence was given as a string of bits, which was parsed into a list and then transformed. Recall also that, for the parsing phase, we have used a monadic parser. In section 3.2, we defined the following solution to this problem:

$$
\begin{aligned}
\mathit{shift} = \mathbf{do}\ & (\mathit{bs}, s) \leftarrow \mathit{bitstring} \\
& \mathit{return}\ (\mathit{transform}\ (\mathit{bs}, s))
\end{aligned}
$$

This solution constructs an intermediate list of bits, that can be eliminated using a specific fusion rule that handles monadic programs. Indeed, the elimination of the intermediate list of bits that is constructed by *bitstring* and consumed by *transform* is going to be achieved by transforming *shift* into a higher-order program. This transformation closely follows the ideas introduced in section 4.2, except that, now, as we deal with monadic functions, we will obtain higher-order monadic programs.

The specific calculation rule that applies for programs such as *shift*, that construct an intermediate list, is as follows[2].

[2]In this Law, and in this section, we use the version of $\mathit{mbuildp}_L$ presented on page 58.

Law 4.4.1 (HIGHER-ORDER PFOLD/MBUILDP FOR LISTS)

$$
\begin{aligned}
&\quad \mathbf{do}\ \{(t,z) \leftarrow \mathit{mbuildp}_L\ g;\\
&\qquad\quad \mathit{return}\ (\mathit{pfold}_L\ (\mathit{hnil},\mathit{hcons})\ (t,z))\}\\
&=\\
&\quad \mathbf{do}\ \{(f,z) \leftarrow g\ (\varphi_{hnil},\varphi_{hcons});\\
&\qquad\quad \mathit{return}\ (f\ z)\}
\end{aligned}
$$

In this law, we use $(\varphi_{hnil}, \varphi_{hcons})$ as the algebra of the higher-order fold that corresponds to the curried version of $\mathit{pfold}_L\ (\mathit{hnil}, \mathit{hcons})$.

Since the functions *transform* and *bitstring* have already been expressed in terms of pfold_L and $\mathit{mbuildp}_L$, respectively,

```
$transform = pfold_L\ (hnil, hcons)$
  where $hnil\ \_ = [\,]$
        $hcons\ ((b, r), s) = (b \oplus s) : r$

$bitstring = mbuildp_L\ g$
  where $g\ (nil, cons) = ($do $b \quad\quad \leftarrow bit$
                              $(bs, s) \leftarrow g\ (nil, cons)$
                              $return\ (cons\ (b, bs), b \oplus s))$
                          $\uplus\ return\ (nil, Zero)$
```

we only need to express *transform* as a higher-order fold, in order to apply the law to the program *shift*:

```
$transform_{ho} :: [Bit] \rightarrow (Bit \rightarrow [Bit])$
$transform_{ho} = fold_L\ (\varphi_{hnil}, \varphi_{hcons})$
  where $\varphi_{hnil} = \lambda\_ \rightarrow [\,]$
        $\varphi_{hcons}\ (b, r) = \lambda s \rightarrow (b \oplus s) : r\ s$
```

Then, we may apply Law 4.4.1 to *shift*, obtaining the higher-order monadic program:

```
$shift =$ do $(f, s) \leftarrow g\ (\varphi_{hnil}, \varphi_{hcons})$
             $return\ (f\ s)$
```

Inlining the above definition, we obtain

```
$shift = \mathbf{do}\ (f, s) \leftarrow g_\varphi$
             $return\ (f\ s)$
   $\mathbf{where}\ g_\varphi = (\mathbf{do}\ b \quad \leftarrow bit$
                      $(f, s) \leftarrow g_\varphi$
                      $return\ (\lambda s' \rightarrow (b \oplus s') : f\ s', b \oplus s))$
                 $\uplus\ return\ (\lambda\_ \rightarrow [\,], Zero)$
```

The *shift* program that we have just derived is a higher-order program in the sense that f, one of the results produced by function g_φ, is itself a function. Once applied to s, function f produces, for the same input string, the same transformed list of bits produced by the original *shift* program.

4.4.2 Algol 68 scope rules

In this section, we calculate a higher-order monadic program that is equivalent to the improved version of the semantic analyzer for the Algol 68 scope rules that we presented in section 3.3. In that section, we defined the following program:

```
$semantics :: Prog \rightarrow IO\ [\,Var\,]$
$semantics\ p = \mathbf{do}\ (p', env) \leftarrow duplicate\ 0\ [\,]\ p$
                    $missing\ (p', env)$
```

In order to transform *semantics*, we use a calculation rule that is an instance of a more general law that we will introduce in section 4.4.3. The law we use is very similar to Law 4.4.1, that we have used to calculate a higher-order version of the program *shift*, which transforms bit strings. In that program, the producer is monadic but the consumer is given by a pure function. In the *semantics* program, however, both the consumer and the producer are monadic, so Law 4.4.1 does not (directly) apply. The new rule, that applies to programs such as *semantics*, is as follows[3].

[3]In this Law, and in this section, we use the version of $mbuildp_L$ presented on page 64.

Law 4.4.2 (EFFECTFUL HIGHER-ORDER PFOLD/MBUILDP FOR LISTS)

$$
\begin{array}{l}
\quad \textbf{do}\ \{(t, z) \leftarrow \mathit{mbuildp}_L\ g\ c;\\
\qquad\quad \mathit{pfold}_L\ (\mathit{hnil}, \mathit{hcons})\ (t, z)\}\\
=\\
\quad \textbf{do}\ \{(f, z) \leftarrow g\ (\varphi_{\mathit{hnil}}, \varphi_{\mathit{hcons}})\ c;\\
\qquad\quad f\ z\}
\end{array}
$$

Now, we want to apply Law 4.4.2 to the *semantics* program. In section 3.3, we have already expressed functions *duplicate* and *missing* in terms of $\mathit{mbuildp}_L$ and pfold_L, respectively. In particular, we have expressed *missing* as:

```
$missing = pfold_L\ (hnil, hcons)$
  where $hnil\ \_ = return\ [\,]$

        $hcons\ ((Use_2\ var, m_{errs}), env)$
          $= $ if $(var \in map\ \pi_1\ env)$
            then $m_{errs}$
            else do $put$ ("Missing: decl " $+\!\!+\ var$)
                    $errs \leftarrow m_{errs}$
                    $return\ (var : errs)$

        $hcons\ ((Dupl_2\ var, m_{errs}), env)$
          $= $ do $errs \leftarrow m_{errs}$
                $return\ (var : errs)$

        $hcons\ ((Block_2\ (lev, its), m_{errs}), env)$
          $= $ do $errs_2 \leftarrow m_{errs}$
                $errs_1 \leftarrow$ do $(p_2, env_2) \leftarrow duplicate\ lev\ env\ its$
                                   $missing\ (p_2, env_2)$
                $return\ (errs_1 +\!\!+\ errs_2)$
```

In order to apply Law 4.4.2 to *semantics*, we need to express *missing* in terms of a higher-order fold equivalent to the above pfold. Indeed, the algebra $(\varphi_{hnil}, \varphi_{hcons})$ of such a higher-order fold is necessary to apply the law.

We define:

```
$missing_{ho} :: Prog_2 \to (Env \to IO\ [Var])$
$missing_{ho} = fold_L\ (\varphi_{hnil}, \varphi_{hcons})$
  **where** $\varphi_{hnil} = \lambda\_ \to return\ [\,]$

        $\varphi_{hcons}\ (Use_2\ var, ferrs)$
          $= \lambda env \to$ **if** $(var \in map\ \pi_1\ env)$
                    **then** $ferrs\ env$
                    **else do** $put$ ("Missing: decl " $+\!\!+\ var$)
                          $errs \leftarrow ferrs\ env$
                          $return\ (var : errs)$

        $\varphi_{hcons}\ (Dupl_2\ var, ferrs)$
          $= \lambda env \to$ **do** $errs \leftarrow ferrs\ env$
                      $return\ (var : errs)$

        $\varphi_{hcons}\ (Block_2\ (lev, its), ferrs)$
          $= \lambda env \to$ **do** $errs_2 \leftarrow ferrs\ env$
                      $errs_1 \leftarrow$ **do** $(p_2, env_2) \leftarrow duplicate\ lev\ env\ its$
                                 $missing\ (p_2, env_2)$
                      $return\ (errs_1 +\!\!+\ errs_2)$
```

Then, by Law 4.4.2, we obtain:

```
$semantics\ p =$ **do** $(ferrs, env) \leftarrow g_\varphi\ 0\ [\,]\ p$
                 $ferrs\ env$

**where**
  $g_\varphi\ lev\ ds\ [\,] = return\ (\lambda\_ \to return\ [\,], ds)$

  $g_\varphi\ lev\ ds\ ((Use\ var) : its)$
    $=$ **do** $(ferrs, ds') \leftarrow g_\varphi\ lev\ ds\ its$
         $return\ (\lambda env \to$ **if** $(var \in map\ \pi_1\ env)$
                                        then ferrs env
                                        else do put ("Missing: decl " ++ var)
                                                errs ← ferrs env
                                                return (var : errs), ds')
    g_φ lev ds ((Decl var) : its)
      = if ((var, lev) ∈ ds)
          then do put ("Duplicate: decl " ++ var)
                  (ferrs, ds') ← g_φ lev ((var, lev) : ds) its
                  return (λenv → do errs ← ferrs env
                                    return (var : errs), ds')
          else g_φ lev ((var, lev) : ds) its

    g_φ lev ds ((Block nested) : its)
      = do (ferrs_2, ds') ← g_φ lev ds its
           return (λenv → do errs_2 ← ferrs_2 env
                             errs_1 ← do (p_2, env_2)
                                            ← duplicate (lev + 1)
                                                        env nested
                                         missing (p_2, env_2)
                             return (errs_1 ++ errs_2), ds')
```

Regarding the program we have just calculated we may notice that still it defines a composition between functions *duplicate* and *missing*. Indeed, one such composition occurs in the definition:

```
    g_φ lev ds ((Block nested) : its)
      = do (ferrs_2, ds') ← g_φ lev ds its
           return (λenv → do errs_2 ← ferrs_2 env
                             errs_1 ← do (p_2, env_2)
                                            ← duplicate (lev + 1)
                                                        env nested
                                         missing (p_2, env_2)
                             return (errs_1 ++ errs_2), ds')
```

This composition, however, may be fused into a single function by direct application of Law 4.4.2 (like in section 4.3, no post-processing is required). We obtain:

$$
\begin{aligned}
& g_{\varphi}\ lev\ ds\ ((Block\ nested) : its) \\
& \quad = \mathbf{do}\ (ferrs_2, ds') \leftarrow g_{\varphi}\ lev\ ds\ its \\
& \qquad\quad return\ (\lambda env \rightarrow \mathbf{do}\ errs_2 \leftarrow ferrs_2\ env \\
& \qquad\qquad\qquad\qquad\qquad\qquad errs_1 \leftarrow \mathbf{do}\ (ferrs_1, env_1) \leftarrow g_{\varphi}\ (lev + 1) \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad env\ nested \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad ferrs_1\ env_1 \\
& \qquad\qquad\qquad\qquad\qquad\qquad return\ (errs_1 +\!\!+\ errs_2), ds')
\end{aligned}
$$

In this section, we have calculated higher-order programs from the composition of particular monadic functions. In the next section we show that the laws we have used in our calculations are specific instances of more general ones. Indeed, the laws we present next are applicable to a wide range of data types and monads.

4.4.3 Calculating monadic higher-order programs, generically

In this section, we study the use of calculation laws to derive higher-order monadic programs. The laws we introduce may be considered as an extension to monadic programs of the law presented in section 4.2, that we review here:

Law 4.2.1 (HIGHER-ORDER PFOLD/BUILDP) For left-strict h,

$$pfold\ h \circ buildp\ g = apply \circ g\ \varphi_h$$

Recall that this law is based on the fact that every *pfold* can be expressed in terms of a higher-order fold: For $h :: (F\ a, z) \rightarrow a$,

$$pfold\ h = apply \circ (fold\ \varphi_h \times id)$$

where $\varphi_h :: F\ (z \rightarrow a) \rightarrow (z \rightarrow a)$ is given by

$$\varphi_h = curry\ (h \circ ((map_F\ apply \circ \tau^F) \vartriangle \pi_2))$$

and $apply :: (a \to b, a) \to b$ by $apply\ (f, x) = f\ x$. Therefore, $fold\ \varphi_h :: \mu F \to (z \to a)$ is the curried version of $pfold\ h$.

Law 4.2.1 can be generalized in the sense of extended shortcut fusion.

Law 4.4.3 *For left-strict h and strictness-preserving N,*

$$map_N\ (pfold\ h) \circ buildp_N\ g = map_N\ apply \circ g\ \varphi_h$$

Proof

$$\begin{array}{ll}
 & map_N\ (pfold\ h) \circ buildp_N\ g \\
= & \{\ (4.1), (2.3)\ \} \\
 & map_N\ apply \circ map_N\ (fold\ \varphi_h \times id) \circ buildp_N\ g \\
= & \{\ \text{Law 3.4.2}\ \} \\
 & map_N\ apply \circ g\ \varphi_h \qquad \Box
\end{array}$$

Like in section 3.4.2, we can consider the particular case when N is the functor of a monad. Unlike the transformation to circular programs, now we do not need to require the monad to be recursive. Again, in first place we state the case when the *pfold* is a pure function. This is the case of, for example, the bit string transformer that we have presented in section 4.4.1. Indeed, Law 4.4.1 that was used in that section to calculate a higher-order program that transforms bit strings is an instance of the following law:

Law 4.4.4 (HIGHER-ORDER PFOLD/MBUILDP) *For left-strict h and strictness-preserving mmap,*

$$\begin{array}{l}
\mathbf{do}\ \{(t, z) \leftarrow mbuildp\ g\ c; return\ (pfold\ h\ (t, z))\} \\
= \\
\mathbf{do}\ \{(f, z) \leftarrow g\ \varphi_h\ c; return\ (f\ z)\}
\end{array}$$

Notice that $mbuildp\ g$ is a function of type $mbuildp\ g :: c \to m\ (a, z)$. However, in section 4.4.1, it was defined as a value of type $m\ (a, z)$, where a is a list, because this form is more appropriate for writing monadic parsers.

Finally, we present the fusion rule that is able to deal with programs where the consumer is also an effectful function. One example of such a program is the analyzer of the Algol 68 scope rules that was presented in section 4.4.2. In that section we have used Law 4.4.2 to transform the original semantic analyzer into a higher-order program. Now, we present the definition of the generic rule that Law 4.4.2 is an instance of.

Law 4.4.5 (EFFECTFUL HIGHER-ORDER PFOLD/MBUILDP) *For left-strict* $h :: (F\ (m\ a), z) \to m\ a$ *and strictness-preserving mmap,*

$$\begin{array}{l} \quad \mathbf{do}\ \{(t, z) \leftarrow mbuildp\ g\ c; pfold\ h\ (t, z)\} \\ = \\ \quad \mathbf{do}\ \{(f, z) \leftarrow g\ \varphi_h\ c; f\ z\} \end{array}$$

Proof

$$\begin{array}{ll} & \mathbf{do}\ \{(t, z) \leftarrow mbuildp\ g\ c; pfold\ h\ (t, z)\} \\ = & \\ & \begin{array}{l} \mathbf{do}\ (t, z) \leftarrow mbuildp\ g\ c \\ \quad\ \ m \quad\ \leftarrow return\ (pfold\ h\ (t, z)) \\ \quad\ \ m \end{array} \\ = & \{\ \text{Law 4.4.4}\ \} \\ & \begin{array}{l} \mathbf{do}\ m \leftarrow \mathbf{do}\ \{(f, z) \leftarrow g\ \varphi_h\ c; return\ (f\ z)\} \\ \quad\ \ m \end{array} \\ = & \\ & \mathbf{do}\ \{(f, z) \leftarrow g\ \varphi_h\ c; f\ z\} \qquad \Box \end{array}$$

4.5 Conclusions

In this chapter we have studied the derivation of higher-order programs using generic calculation rules. We have generalized the shortcut fusion rule for deriving pure higher-order programs presented in (Voigtländer 2008) so that it can be applied to compositions of programs with an arbitrary data type as intermediate structure. We have also presented an extension of the above rule for the derivation of higher-order programs to the case of monadic programs. We considered programs consisting of the composition between a monadic producer and a pure consumer and programs where both the producer and the consumer functions are monadic.

Chapter 5

Strictification of Circular Programs

Summary

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. 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 grammar-based techniques. Moreover, we use standard slicing techniques to slice circular programs.

5.1 Introduction

Circular lazy programs elegantly and concisely model multiple traversal algorithms in a lazy language. 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 may lead 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). The presented techniques analyze the circular program and construct a dependency graph that records how each program variable depends on the others. Our analysis then builds a set of new relations between the program variables, breaking up the circular dependencies and establishing a new evaluation order for the program. The new order is finally translated into a strict multiple traversal program. Moreover, we use program specialisation techniques to derive deforested versions of the strict programs.

Because our techniques break up circular definitions into several strict functions, we can directly apply standard slicing techniques to slice circular lazy programs. Slicing is a program manipulation technique, that also relies on a program dependency graph (Horwits and Reps 1992; Tip 1995), and that is used to identify sub-programs of a program under study. Standard slicing techniques, however, do not directly apply to circular programs: they would construct a circular dependency graph, and the slicing operations performed on that graph could result in non-termination. Given a circular program, we will then 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).

In order to motivate the use of our techniques, we will define a circular program to format HTML style tables. Using our strictification technique, that circular program will initially be transformed into a two traversal program: the first traversal of the derived program will compute an intermediate data structure to be used by the second traversal to produce the desired formatting. Later, that intermediate data structure will be deforested, by the application of standard program specialisation techniques. We obtain a table formatter program consisting of the composition of two higher-order, data structure free, functions. Finally, we will present a fragment (or slice) of the original program that only computes the width of an input table. This fragment will be obtained by performing backward slicing on the original circular program using as slicing criteria the width of a table.

This chapter is organized as follows. Section 5.2 presents the notation and the running example used throughout the chapter. Section 5.3 presents the derivation of strict programs from circular ones. Section 5.4 presents the slicing of circular programs. In section 5.5 we discuss the class of circular programs considered. Section 5.6 shows our conclusions.

5.2 Notation

In order to make it easier to demonstrate our strictification techniques, we introduce the programming language given in Fig. 5.1. The rules of the language will be used, for example, in section 5.3, to infer dependency relations between the arguments and the results of a function call. Such relations are the building blocks of our method to analyze and eliminate the circular definitions that occur in a program.

A program in our language consists of 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.

5.2.1 The table formatter program

In this thesis, we have already presented several examples that demonstrate the power of circular programming. In this chapter, we consider the development of a circular program to solve a new programming problem.

Let us consider that we want to define a program that formats HTML style tables. Fig. 5.2 shows an example of a possible input (left) and correspondent output (right).

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
	$\mid$	**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 5.1: Abstract syntax

⟨TABLE⟩
 $\langle TR\rangle\langle TD\rangle^{1}_{14}$ `The first line` ⟨/TD⟩$\langle TD\rangle^{1}_{4}$ `of a` ⟨/TD⟩⟨/TR⟩

 $\langle TR\rangle\langle TD\rangle^{7}_{12}$⟨TABLE⟩
  $\langle TR\rangle\langle TD\rangle^{1}_{4}$ `This` ⟨/TD⟩$\langle TD\rangle^{1}_{2}$ `is` ⟨/TD⟩⟨/TR⟩

  $\langle TR\rangle\langle TD\rangle^{1}_{7}$ `another` ⟨/TD⟩⟨TD/⟩⟨/TR⟩

  $\langle TR\rangle\langle TD\rangle^{1}_{5}$ `table` ⟨/TD⟩⟨TD/⟩⟨/TR⟩
 ⟨/TABLE⟩
 ⟨/TD⟩$\langle TD\rangle^{1}_{5}$ `table` ⟨/TD⟩⟨/TR⟩
⟨/TABLE⟩

```
|--------------------|
|The first line|of a |
|--------------------|
||----------|  |table|
||This   |is|  |     |
||----------|  |     |
||another|  |  |     |
||----------|  |     |
||table  |  |  |     |
||----------|  |     |
|--------------------|
```

Figure 5.2: 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 row, 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 perform 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 to format tables. Note that to construct such a circular program the programmer does 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 this chapter.

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

$$\begin{aligned}
&\textbf{data } \textit{Table} &&= \textit{RootTable Rows}\\
&\textbf{data } \textit{Rows} &&= \textit{EmptyRows}\\
& &&\mid \textit{ConsRows}\ (\textit{Row}, \textit{Rows})\\
&\textbf{data } \textit{Row} &&= \textit{OneRow Elems}\\
&\textbf{data } \textit{Elems} &&= \textit{EmptyElems}\\
& &&\mid \textit{ConsElems}\ (\textit{Elem}, \textit{Elems})\\
&\textbf{data } \textit{Elem} &&= \textit{OneStr String}\\
& &&\mid \textit{OneTable Table}
\end{aligned}$$

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 an argument the list of available widths of the columns. This list is the computed list of widths. As we can see next, a circular dependency is defined.

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

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 $zipwith_{Max}$ is the list $[14, 5]$, that is, the maximum width of each column.

```
$evalRows\ (ConsRows\ (row, rows), aws) = (lines, mh, mws)$
  **where** $(lines_1, mh_1, mws_1) = evalRow\ (row, aws)$
        $(lines_2, mh_2, mws_2) = evalRows\ (rows, aws)$
        $mh\ \ = mh_1 + mh_2 + 1$   -- + 1 is for the separator
        $mws = zipwith_{Max}\ (mws_1, mws_2)$
        $lines = addSep\ (aws, lines_1, lines_2)$

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

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.

```
$evalRow\ (OneRow\ elems, aws) = (lines, mh, mws)$
  **where** $(lines_1, \boxed{\text{mh}}, mws) = evalElems\ (elems, \boxed{\text{mh}}, aws)$
        $lines = addBorder\ lines_1$
```

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

```
$evalElems\ (ConsElems\ (elem, elems), ah, aws) = (lines, mh, mws)$
  **where** $aws_2 = tail\ aws$
        $(lines_1, mh_1, mw_1)\ \ = evalElem\ elem$
        $(lines_2, mh_2, mws_2) = evalElems\ (elems, ah, aws_2)$
        $mws\ = mw_1 : mws_2$
        $mh\ \ = max\ (mh_1, mh_2)$
```

$$lines = glue\ (aws, mw_1, ah, mh_1, lines_1, lines_2)$$

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

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

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

$$evalElem\ (OneTable\ table) = (lines_1, mh_1, mw_1)$$
$$\quad \mathbf{where}\ (lines_1, mh_1, mw_1) = evalTable\ table$$

The functions *addSep*, *sepLine*, *addBorder* and *glue*, add line separators, horizontal and vertical borders, and glue table lines, respectively. We omit their (simple) definition here.

This table formatter is a *circular* program: circular definitions occur twice as we can see in the program. The lazy engine, however, will be able to schedule the computations and convey values between different traversal functions at execution time. In the next sections, we will show how to transform this circular program into a strict multiple traversal program that relies on intermediate structure to glue the different traversals. We will also show how such redundant data structures can be deforested.

5.3 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.

5.3.1 Detection of circular definitions

Let us analyze in detail one of the most intricate function alternatives of the Table Formatter program: the function *evalTable* applied at the node *RootTable*, where a circular definition occurs.

$$\begin{aligned}
&evalTable\ (RootTable\ rows) = (lines, mh, mw)\\
&\quad \textbf{where}\ (lines_1, mh_1, \boxed{\text{mws}}) = evalRows\ (rows, \boxed{\text{mws}})\\
&\qquad\quad mh \ \ = mh_1 + 2\\
&\qquad\quad mw \ \ = (sum\ mws) + (length\ mws) + 1\\
&\qquad\quad lines = sepLine\ (mws, lines_1)
\end{aligned}$$

Figure 5.3 shows the induced dependency relation (represented as a graph), which follows from a flow analysis of the total program.

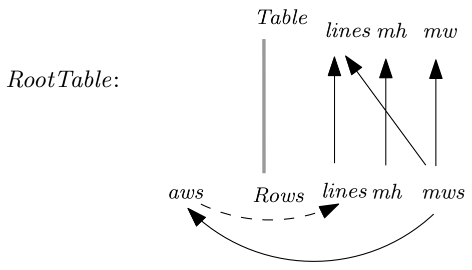


Figure 5.3: Dependency graph of function *evalTable*

For each alternative function definition a dependency graph is induced. Such graphs are labeled with the data type constructor that the alternative definition refers to. Furthermore, in these graphs we use undirected (solid) lines to connect the types involved in a tree-like structure: result type on top and arguments at the bottom. The variable names representing formal arguments (results) of the function definition are displayed at the left (right) of the resulting type. Such variable names are displayed in all occurrences of that data type in the different induced graphs. Notice, for example, that the results produced by *evalTable*: *lines* (the formatted lines of the table), *mh* (the minimal height of the table) and *mw* (the table's minimal width), 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* (indeed, a circular definition ensures that *mws* is copied to the second argument of function *evalRows*, i.e., copied to *aws*). We use black lines to represent direct dependencies and dashed-black lines to represent indirect dependencies. In Figure 5.3, the dashed arrow with origin in the variable *aws* and destination in the variable *lines* represents an induced dependency that is inferred in the definition of *evalTable* for constructor *ConsRows*, in a way that we will describe later. The formal process to calculate both direct and indirect dependencies is presented in the next sections.

As we can easily see in Figure 5.3, 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: they correspond to circular definitions in the program. The detection of circular definitions in the abstract syntax tree of the programs under consideration then corresponds to detecting dependencies, in one of the functions defined in the program, from one of its results to one of its arguments.

5.3.2 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 analysis we present adapts the static analysis that has already been proposed for attribute grammars (Kastens 1980).

The algorithms that compute the alternative evaluation order establish the number of visits and an *interface* for every data-type X of the circular program. We denote the interface of data-type X by *Interface* (X).

Interface (X), as computed by these algorithms, usually has the following shape:

$$\mathit{Interface}\ (X) = [(\mathit{args}_1, \mathit{results}_1), \ldots, (\mathit{args}_n, \mathit{results}_n)]$$

where

$$\mathit{args}_i = \{\mathit{arguments\ of\ the\ } i_{th}\ \mathit{function\ defined\ over\ } X\}$$

$$\mathit{results}_i = \{\mathit{results\ of\ the\ } i_{th}\ \mathit{function\ defined\ over\ } X\}$$

Thus, by computing *Interface* (X), for every data-type X, the scheduling algorithms specify, for every visit i 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:

$$DP\ (C) = \{\ \mathit{Var}_1\ \rightarrow\ \mathit{Var}_2\ \mid\ \mathit{Var}_2\ \mathit{depends\ on}\ \mathit{Var}_1\ \mathit{in}\ C\}$$

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 circular program. Indeed, if we consider the abstract syntax of our programs (Figure 5.1), we may define: in the first attribution rule ($v_1 = v_2$), the variable v_1 depends on the variable v_2, in the second attribution 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 occur in e. We present the direct dependencies induced by the abstract table formatter program in Figure 5.4 (solid directed lines were used to represent this type of dependencies). Next, we also present the derived *DP* relation, for the constructor *RootTable* of that same 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, constructor *RootTable* has type $RootTable :: Rows \rightarrow Table$, and the tuple $(RootTable, 1, lines)$ 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*, this value being represented by $(RootTable, 1, lines)$.

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

Definition 5.3.1 (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*

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

such that $\langle C, i\rangle = X_i$, is non-circular.

In this case we say that the interfaces are compatible.

A relation of dependencies between variables is non-circular if its transitive closure 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 by the similar concept for attribute grammars. In the literature, one finds a slightly different class of attribute grammars, the so-called *L-ordered* attribute grammars. The classes of *partitionable* and *L-ordered* attribute grammars are equal for attribute grammars in Backus Naur form (BNF). We may define a class of circular programs similar to the class of *L-ordered* attribute grammars.

Definition 5.3.2 (L-Ordered Circular Program) *A circular program is an L-ordered circular program if there exist total orders $TO\ (X)$ for every data-type X such that for every constructor $C :: (X_1, X_2, \ldots, X_n) \rightarrow X_0$, the relation*

$$DP\ (C)\ \cup \bigcup_{i=0}^{n}\ TO\ (\langle C, i\rangle)$$

such that $\langle C, i\rangle = X_i$, is cycle free.

The total orders $TO\ (X)$ are easily converted into interfaces, by cutting them into maximal segments of function arguments and function results.

However, Engelfriet and Filé (1982) proved that deciding whether an attribute grammar is *L-ordered* or not is an NP-complete problem. Kastens

(1980) defined a subclass of *L-ordered* 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. In the next section, we adapt Kastens' algorithm to work on circular programs.

5.3.3 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.

Our adaptation of Kasten's algorithm proceeds in three steps. First, we compute the direct dependencies between variable occurrences in the program. Next, we calculate the relation of induced dependencies. Finally, the interfaces for the data-type symbols are defined. These steps are presented in detail next.

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

The circular program is not ordered if DP is cyclic.

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

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

The circular program is not ordered if IDP is cyclic.

Figure 5.4 shows the IDP relation (dashed lines were used to represent it) induced by the Table formatter circular program[1].

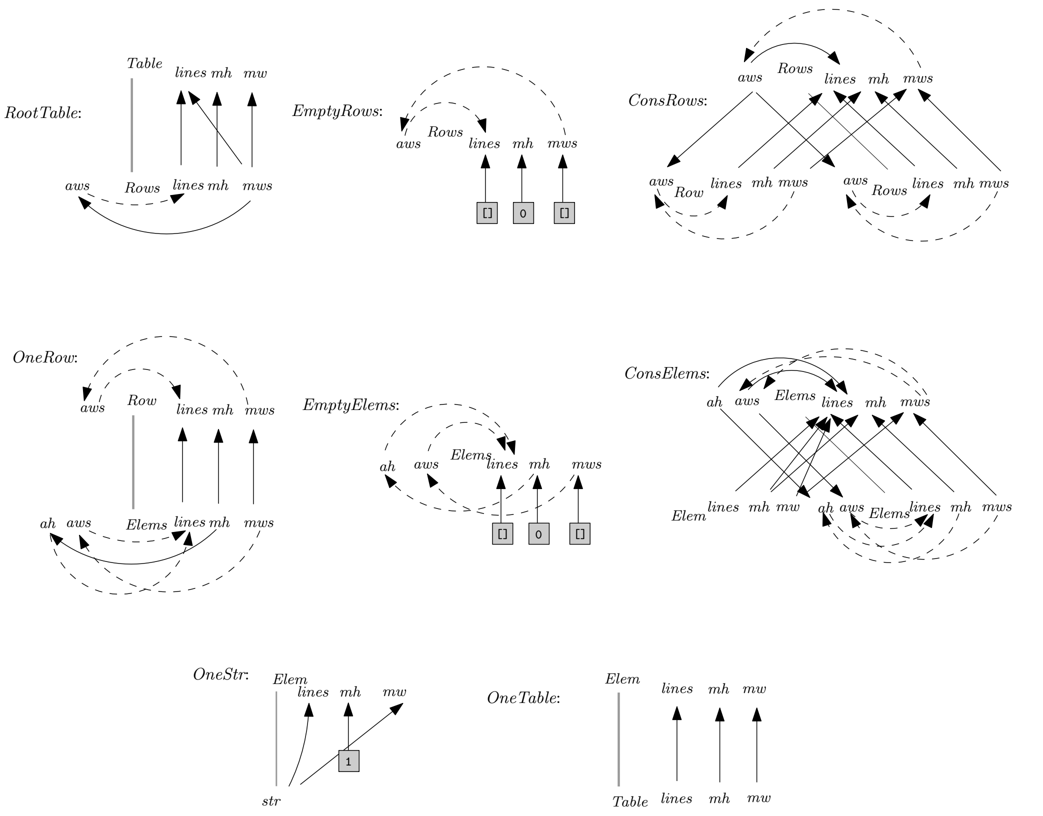


Figure 5.4: Dependency graph DP (solid arrows), IDP (dashed arrows)

[1]In fact, for simplicity and readability, Figure 5.4 omits the representation of the dependencies established, in IDP, between two argument variables and between two result variables, e.g., the dependency (*RootTable*, 1, *mws*) $\to$ (*RootTable*, 1, *lines*) is omitted.

Next, we compute the relation $\mathit{IDS} = \bigcup_{X \in DataTypes} \mathit{IDS}\ (X)$, where *DataTypes* is the set of the program's data-types, that defines the *Induced Dependencies* among variables:

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

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

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

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

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

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

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

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

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

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

Definition 5.3.3 (Ordered Circular Program) *A circular program is an ordered circular program if the relation*

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

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 in section 5.5. 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 the 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[2].

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

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

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

Let us now prove that the Table formatter circular 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 the program. We obtain

$$\begin{aligned} A_{\mathit{Table},1} &= \{\, \mathit{Table.lines}, \mathit{Table.mh}, \mathit{Table.mw} \} \\ A_{\mathit{Table},2} &= \{\,\} \\ A_{\mathit{Rows},1} &= \{\, \mathit{Rows.lines}, \mathit{Rows.mh} \} \end{aligned}$$

[2]Provided that the auxiliary functions used in the program also terminate.

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

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

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

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

As we can easily notice, all the *DS* dependency relations are cycle free. Furthermore, we can observe the graphs shown in Figure 5.4 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-type symbols:

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

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*).

5.3.4 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 node 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 v_{th} 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 $\mathit{Interface}\ (X)$. We denote the number of visits of data-type X by $\mathtt{v}(X)$. Furthermore, each visit v to X, with $1 \leq v \leq \mathtt{v}(X)$, has a fixed *interface*: the element in position v of sequence $\mathit{Interface}\ (X)$. 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 $\mathit{Args}_v(X)$ and $\mathit{Res}_v(X)$, where

$$\mathit{Args}_v(X) = A_{X,2*(\mathtt{v}(X)-v+1)}$$

and

$$\mathit{Res}_v(X) = A_{X,2*(\mathtt{v}(X)-v)+1}$$

The visit-sequence of a constructor is usually presented as a list of the two basic instructions. Visit-sequences, however, are the *input* of our techniques to derive purely functional programs. Thus, they are divided into *visit-sub-sequences vss* (C, v), delimited by **begin** v and **end** v, containing the instructions to be performed on visit v to the constructor C, where C is a constructor of X, and $1 \leq v \leq \mathtt{v}(X)$. In order to simplify the presentation, visit-sub-sequences are also annotated with *define* and *usage* variable directives. Every visit-sub-sequence *vss* (C, v) is annotated with the *interface* of visit v to X. Therefore *vss* (C, v) is annotated with `arg` $(\mathit{Args}_v(X))$ and `res` $(\mathit{Res}_v(X))$.

Every instruction `eval` (x) is annotated with the directive `uses` (bs) that specifies the list of variable occurrences used to evaluate x, *i.e.*, the occurrences that x depends on. The instruction `visit` (X_i, v) causes child i of constructor C, where $C :: (X_1, X_2, \ldots, X_n) \rightarrow X_0$, to be visited for the v_{th} time. The visit uses the variable occurrences of $Args_v(X_i)$ as arguments and returns the variable occurrences of $Res_v(X_i)$. Thus `visit` (X_i, v) is annotated with `inp` and `out` where `inp` is the list of the elements of $Args_v(X_i)$ and `out` is the list of elements of $Res_v(X_i)$.

Figure 5.5 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 these 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 5.2, in the multiple traversal evaluator of the table formatter, the height, the width and the formatted lines of the nested tables have to be passed from the first to the second traversal of the outer table. 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.

5.3.5 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 mimic the imperative approach: values needed later are stored in a new tree, called a *visit tree*. Such values have to be preserved from the traversal that creates them until the last traversal that uses them. Thus, each traversal builds a new visit tree containing in its nodes the values needed in future visits. The functions that

```
plan RootTable
begin^1  arg()
  visit  (Rows, 1)
         inp()
         out(Rows.mws),
  eval   (Table.mw)
         uses(Rows.mws),
  eval   (Rows.aws)
         uses(Rows.mws),
  visit  (Rows, 2)
         inp(Rows.aws)
         out(Rows.lines, Rows.mh),
  eval   (Table.lines)
         uses(Rows.mws, Rows.lines),
  eval   (Table.mh)
         uses(Rows.mh),
end^1    res(Table.lines, Table.mh, Table.mw)

plan OneRow
begin^1  arg()
  visit  (Elems, 1)
         inp()
         out(Elems.mws, $\boxed{Elems.mh}$),
  eval   (Row.mws)
         uses(Elems.mws),
end^1    res(Row.mws)
begin^2  arg(Row.aws)
  eval   (Row.mh)
         uses(Elems.mh),
  eval   (Elems.ah)
         uses($\boxed{Elems.mh}$),
  visit  (Elems, 2)
         inp(Elems.ah, Elems.aws)
         out(Elems.lines),
  eval   (Elems.aws)
         uses(Row.aws),
  eval   (Row.lines)
         uses(Elems.lines),
end^2    res(Row.mh, Row.lines)

plan OneStr
begin^1  arg()
  eval   (Elem.mh)
         uses()
  eval   (Elem.lines)
         uses(str)
  eval   (Elem.mw)
         uses(str)
end^1    res(Elem.lines, Elem.mh, Elem.mw)

plan EmptyRows
begin^1  arg()
  eval   (Rows.mws)
         uses()
end^1    res(Rows.mws)
begin^2  arg(Rows.aws)
  eval   (Rows.mh)
         uses()
  eval   (Rows.lines)
         uses()
end^2    res(Rows.mh, Rows.lines)

plan EmptyElems
begin^1  arg()
  eval   (Elems.mws)
         uses(),
  eval   (Elems.mh)
         uses(),
end^1    res(Elems.mh, Elems.mws)
begin^2  arg(Elems.ah, Elems.aws)
  eval   (Elems.lines)
         uses(),
end^2    res(Elems.lines)

plan OneTable
begin^1  arg()
  visit  (Table, 1)
         inp()
         out(Table.lines, Table.mh, Table.mw),
  eval   (Elem.mh)
         uses(Table.mh)
  eval   (Elem.mw)
         uses(Table.mw)
  eval   (Elem.lines)
         uses(Table.lines)
end^1    res(Elem.lines, Elem.mh, Elem.mw)

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)
begin^2  arg($Rows_1$.aws)
  eval   (Row.aws)
         uses($Rows_1$.aws),
  visit  (Row, 2)
         inp(Row.aws)
         out(Row.lines, Row.mh),
  eval   ($Rows_2$.aws)
         uses($Rows_1$.aws),
  visit  ($Rows_2$, 2)
         inp($Rows_2$.aws)
         out($Rows_2$.lines, $Rows_2$.mh),
  eval   ($Rows_1$.mh)
         uses(Row.mh, $Rows_2$.mh)
  eval   ($Rows_1$.lines)
         uses($Rows_1$.aws, Row.lines, $Rows_2$.lines)
end^2    res($Rows_1$.lines, $Rows_1$.mh)

plan ConsElems
begin^1  arg()
  visit  ($Elems_2$, 1)
         inp()
         out($Elems_2$.mh, $Elems_2$.mws),
  visit  (Elem, 1)
         inp()
         out($\boxed{Elem.lines}$, $\boxed{Elem.mh}$, $\boxed{Elems.mw}$),
  eval   ($Elems_1$.mh)
         uses(Elem.mh, $Elems_2$.mh),
  eval   ($Elems_1$.mws)
         uses(Elem.mw, $Elems_2$.mws)
end^1    res($Elems_1$.mh, $Elems_1$.mws)
begin^2  arg($Elems_1$.ah, $Elems_1$.aws)
  eval   ($Elems_2$.ah)
         uses($Elems_1$.ah),
  eval   ($Elems_2$.aws)
         uses($Elems_1$.aws),
  visit  ($Elems_2$, 2)
         inp($Elems_2$.ah, $Elems_2$.aws)
         out($Elems_2$.lines),
  eval   ($Elems_1$.lines)
         uses($Elems_1$.aws, $\boxed{Elem.mw}$, $\boxed{Elem.mh}$,
         $Elems_1$.ah, $\boxed{Elem.lines}$, $Elems_2$.lines),
end^2    res($Elems_1$.lines)
```

Figure 5.5: The visit-sub-sequences induced by the *Table* circular program.

represent the subsequent traversals 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 the *Table* program that is obtained by applying these 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 the type of the $\mathit{ConsElems}_2$ 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 5.5, is mapped to function *visitTable*.

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

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 5.2. 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*.

```
$visitRows_1\ (ConsRows\ (row, rows)) = (ConsRows_2\ (row_2, rows_2), mws)$
  **where** $(rows_2, mws_2) = visitRows_1\ rows$
        $(row_2\ , mws_1) = visitRow_1\ \ row$
        $mws = zipwith_{Max}\ (mws_1, mws_2)$

$visitRows_1\ EmptyRows = (EmptyRows_2, [\,])$

$visitRows_2\ (ConsRows_2\ (row, rows), aws) = (lines, mh)$
  **where** $(lines_1, mh_1) = visitRow_2\ (row, aws)$
        $(lines_2, mh_2) = visitRows_2\ (rows, aws)$
        $lines = addSep\ (aws, lines_1, lines_2)$
        $mh = mh_1 + mh_2 + 1$

$visitRows_2\ (EmptyRows_2, aws) = ([\,], -1)$
```

As for constructor *OneRow*, recall Figure 5.5 to notice the two *visit-sub-sequences* scheduled over it. The first one is mapped to function $visitRow_1$ and the second one to function $visitRow_2$.

```
$visitRow_1\ (OneRow\ elems) = (OneRow_2\ (mh_1, elems_2), mws_1)$
  **where** $(elems_2, mws_1, mh_1) = visitElems_1\ elems$

$visitRow_2\ (OneRow_2\ (mh_1, elems), aws) = (lines, mh_1)$
  **where** $lines_1 = visitElems_2\ (elems, mh_1, aws)$
        $lines = addBorder\ lines_1$
```

Constructors *ConsElems* and *EmptyElems* have also been scheduled two *visit-sub-sequences*, that we translate to the strict functions $visitElems_1$ and $visitElems_2$. Notice that this scheduling breaks up *evalElems*' circular invocation, inside function *evalRow*, into a two traversal strategy.

```
$visitElems_1\ (ConsElems\ (elem, elems))$
  $= (ConsElems_2\ (mh_1, mw_1, lines_1, elems_2), mh, mws)$
  **where** $(lines_1, mh_1, mw_1)\ \ \ = visitElem\ elem$
        $(elems_2, mh_2, mws_2) = visitElems_1\ elems$
        $mh = max\ mh_1\ mh_2$
        $mws = mw_1 : mws_2$
```

$$\mathit{visitElems}_1\ \mathit{EmptyElems} = (\mathit{EmptyElems}_2, 0, [\,])$$

$$\begin{array}{l}
\mathit{visitElems}_2\ (\mathit{ConsElems}_2\ (\mathit{lines}_1, \mathit{mh}_1, \mathit{mw}_1, \mathit{elems}), \mathit{ah}, \mathit{aws}) = \mathit{lines} \\
\quad \textbf{where}\ \mathit{aws}_2 = \mathit{tail}\ \mathit{aws} \\
\qquad\qquad \mathit{lines}_2 = \mathit{visitElems}_2\ (\mathit{elems}, \mathit{ah}, \mathit{aws}_2) \\
\qquad\qquad \mathit{lines} = \mathit{glue}\ (\mathit{aws}, \mathit{mw}_1, \mathit{ah}, \mathit{mh}_1, \mathit{lines}_1, \mathit{lines}_2)
\end{array}$$

$$\mathit{visitElems}_2\ (\mathit{EmptyElems}_2, \mathit{ah}, \mathit{aws}) = [\,]$$

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.

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

$$\begin{array}{l}
\mathit{visitElem}\ (\mathit{OneTable}\ \mathit{table}) = (\mathit{lines}_1, \mathit{mh}_1, \mathit{mw}_1) \\
\quad \textbf{where}\ (\mathit{lines}_1, \mathit{mh}_1, \mathit{mw}_1) = \mathit{visitTable}\ \mathit{table}
\end{array}$$

5.3.6 Deforestation by program specialisation

The strict programs derived using the techniques presented in the previous section rely on a (possibly) large number of gluing intermediate data structures to convey information between different traversals. Such redundant structures can, however, be eliminated by using program specialisation techniques (Peyton Jones 2007). 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 needed in future visits is passed on by partially parameterizing a more general function.

Next, we show the strict, deforested *Table* program obtained by program specialisation of the strict one. Function *visitTable* is transformed into the following higher-order function:

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

Notice that the calls $\mathit{visitRows}_1$ *rows* and $\mathit{visitRows}_2$ $(\mathit{rows}_2, \mathit{mws})$ 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 program specialization.

Next, functions $\mathit{consRows}_1$, $\mathit{emptyRows}_1$, $\mathit{consRows}_2$ and $\mathit{emptyRows}_2$ are presented. Functions $\mathit{consRows}_1$ and $\mathit{emptyRows}_1$ specialize the definition of function $\mathit{visitRows}_1$ over the constructors *ConsRows* and *EmptyRows*, respectively, while functions $\mathit{consRows}_2$ and $\mathit{emptyRows}_2$ specialize the definition of $\mathit{visitRows}_2$ over constructors $\mathit{ConsRows}_2$ and $\mathit{EmptyRows}_2$.

$$
\begin{aligned}
&\mathit{consRows}_1\ (\mathit{row}, \mathit{rows}) = (\mathit{consRows}_2\ (\mathit{row}_2, \mathit{rows}_2), \mathit{mws})\\
&\quad \textbf{where}\ (\mathit{rows}_2, \mathit{mws}_2) = \mathit{rows}\\
&\qquad\qquad (\mathit{row}_2, \mathit{mws}_1)\ \ = \mathit{row}\\
&\qquad\qquad \mathit{mws} = \mathit{zipwith\_max}\ (\mathit{mws}_1, \mathit{mws}_2)
\end{aligned}
$$

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

$$\mathit{consRows}_2\ (\mathit{row}, \mathit{rows}, \mathit{aws}) = (\mathit{lines}, \mathit{mh})$$

```
    where (lines_1, mh_1) = row aws
          (lines_2, mh_2) = rows aws
          lines = addSep (aws, lines_1, lines_2)
          mh = mh_1 + mh_2 + 1

emptyRows_2 aws = ([], -1)
```

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

```
oneRow_1 elems = (oneRow_2 (mh_1, elems_2), mws_1)
  where (elems_2, mws_1, mh_1) = elems

oneRow_2 (mh_1, elems, aws) = (lines, mh_1)
  where lines_1 = elems (mh_1, aws)
        lines   = addBorder lines_1
```

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

```
consElems_1 (elem, elems)
   = (consElems_2 (lines_1, mh_1, mw_1, elems_2), mh, mws)
  where (lines_1 , mh_1, mw_1)  = elem
        (elems_2, mh_2, mws_2) = elems
        mh = max mh_1 mh_2
        mws = mw_1 : mws_2

emptyElems_1 = (emptyElems_2, 0, [])

consElems_2 ((lines_1, mh_1, mw_1, elems_2), ah, aws) = lines
  where aws_2 = tail aws
        lines_2 = elems_2 (ah, aws_2)
        lines = glue (aws, mw_1, ah, mh_1, lines_1, lines_2)

emptyElems_2 (ah, aws) = []
```

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

```
oneStr str = ([str], 1, length str)

oneTable table = (lines_1, mh_1, mw_1)
  where (lines_1, mh_1, mw_1) = table
```

As a result of specialising the *Table* strict program presented in the previous section, we obtain a deforested program consisting of a set of higher-order functions. We should notice, however, that the execution of this higher-order program is different than that of the strict one. Indeed, in the strict *Table* program, function *visitTable* is applied to a concrete element of type *Table* to produce its formatting. One such element is as follows:

```
table = RootTable
  (ConsRows (OneRow (ConsElems (OneStr "simon",
                     ConsElems (OneStr "mary",
                                EmptyElems))),

   ConsRows (OneRow (ConsElems (OneStr "richard",
                     ConsElems (OneTable nested,
                                EmptyElems))),
             EmptyRows)))

nested = RootTable
  (ConsRows (OneRow (ConsElems (OneStr "john",
                     ConsElems (OneStr "steve",
                                EmptyElems))),
             EmptyRows))
```

In order to format the same table using the higher-order *Table* program derived in this section, we would now simply have to run *table*, where

```
table = rootTable
  (consRows_1 (oneRow_1 (consElems_1 (oneStr "simon",
                         consElems_1 (oneStr "mary",
                                      emptyElems_1))),

   consRows_1 (oneRow_1 (consElems_1 (oneStr "richard",
```

$$\begin{aligned}
&\qquad\qquad\qquad\qquad \mathit{consElems}_1\ (\mathit{oneTable}\ \mathit{nested},\\
&\qquad\qquad\qquad\qquad\qquad\qquad \mathit{emptyElems}_1))),\\
&\qquad\qquad \mathit{emptyRows}_1)))
\end{aligned}$$

$$\begin{aligned}
&\mathit{nested} = \mathit{rootTable}\\
&\quad (\mathit{consRows}_1\ (\mathit{oneRow}_1\ (\mathit{consElems}_1\ (\mathit{oneStr}\ \texttt{"john"},\\
&\qquad\qquad\qquad\qquad \mathit{consElems}_1\ (\mathit{oneStr}\ \texttt{"steve"},\\
&\qquad\qquad\qquad\qquad\qquad\qquad \mathit{emptyElems}_1))),\\
&\qquad\qquad \mathit{emptyRows}_1))
\end{aligned}$$

Alternatively, we can define a simple set of combinators for constructing tables. This works for both the strict and the higher-order versions of the *Table* program. For the higher-order version of that program we can define, for example, the following combinators:

$$\begin{aligned}
&\mathit{table}\ \mathit{rows} = \mathit{rootTable}\ (f\ \mathit{rows})\\
&\quad \textbf{where}\ f\ [\,] = \mathit{emptyRows}_1\\
&\qquad\qquad f\ (h:t) = \mathit{consRows}_1\ (\mathit{oneRow}_1\ (g\ h), f\ t)\\
&\qquad\qquad g\ [\,] = \mathit{emptyElems}_1\\
&\qquad\qquad g\ (h:t) = \mathit{consElems}_1\ (h, g\ t)\\
&\mathit{txt}\ \mathit{str} = \mathit{oneStr}\ \mathit{str}\\
&\mathit{nested}\ t = \mathit{oneTable}\ \$\ \mathit{table}\ t
\end{aligned}$$

We may now use this combinators to format the table that we have been considering. Indeed, we may define:

$$\begin{aligned}
&t = \mathit{table}\ [[\mathit{txt}\ \texttt{"simon"}, \mathit{txt}\ \texttt{"mary"}],\\
&\qquad\qquad [\mathit{txt}\ \texttt{"richard"}, \mathit{nested}\ [[\mathit{txt}\ \texttt{"john"}, \mathit{txt}\ \texttt{"steve"}]]]]
\end{aligned}$$

By running t, we obtain the expected formatting for this table, that we present in Figure 5.6.

Although we have used a first-order circular program as the running example, the techniques introduced by the higher-order extension to attribute grammars (Swierstra and Vogt 1991) directly apply to the transformation of higher-order circular functions, as well. Circular programs modelling algo-

```
|--------------------|
|simon  |mary        |
|--------------------|
|richard||----------||
|       ||john|steve||
|       ||----------||
|--------------------|
```

Figure 5.6: The formatted table.

rithms 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. 1997) 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. The calculation techniques that we have presented in chapter 2 may also be applied to some of the strict programs derived in this section. These programs are then transformed back into circular programs and into higher-order deforested programs. In the next section we present the use of program slicing techniques to slice circular programs.

5.4 Slicing circular programs

Although the programming language community has done a considerable amount of work on program slicing (Horwits and Reps 1992; Tip 1995), there is little work done on slicing of lazy functional languages. In this section, we 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 computing the width of the table. All other definitions are *sliced-out*.

We start by considering the top level constructor of that program, *i.e.*, the *RootTable* constructor. From the total visit sequence plan scheduled for this constructor (presented in Figure 5.5), 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, that 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 $ConsElems$
begin$^1$  arg()
  visit  $(Elems_2, 1)$
         inp()
         out$(Elems_2.mws)$,
  visit  $(Elem, 1)$
         inp()
         out$(\boxed{Elems.mw})$,
  eval   $(Elems_1.mws)$
         uses$(Elem.mw, Elems_2.mws)$
end$^1$    res$(Elems_1.mws)$
```

```
plan $EmptyElems$
begin$^1$  arg()
  eval   $(Elems.mws)$
         uses(),
end$^1$    res$(Elems.mws)$
```

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

```
plan $OneStr$
begin$^1$  arg()
  eval   $(Elem.mw)$
         uses$(str)$
end$^1$    res$(Elem.mw)$
```

```
plan $OneTable$
begin$^1$  arg()
  visit  $(Table, 1)$
         inp()
         out$(Table.mw)$,
  eval   $(Elem.mw)$
         uses$(Table.mw)$
end$^1$    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 just presented into valid *Haskell* definitions.

$visitTable :: Table \rightarrow Int$

```
visitTable (RootTable rows) = mw
  where mws_1 = visitRows rows
        mw    = (sum mws_1) + (length mws_1) + 1

visitRows (ConsRows (row, rows)) = mws
  where mws_2 = visitRows rows
        mws_1 = visitRow  row
        mws   = zipwith_max mws_1 mws_2

visitRows EmptyRows = [ ]

visitRow (OneRow els) = mws
  where mws = visitElems_1 els

visitElems (ConsElems (el, els)) = mw_1 : mws_2
  where mws_2 = visitElems els
        mw_1 = visitElem el

visitElems EmptyElems = [ ]

visitElem (OneStr str) = length str

visitElem (OneTable table) = mw_1
  where mw_1 = visitTable table
```

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 this chapter, that is circular, strict or deforested programs. This is the case if we consider, in our example, as the slicing criteria the result that computes the table (*lines*). The resulting programs are very similar to the ones we have presented, with the exception that the top function returns one result only: the formatted table.

5.5 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 it 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. In the next chapter, we will use this complex four traversal pretty printing algorithm to benchmark the different strict and lazy programs.

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 the attribute grammar 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 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 cannot be (directly) considered is the breadth-first numbering. However, the tricky example presented by Okasaki can be slightly modified and then expressed as an ordered circular program[3].

5.6 Conclusion

This chapter presented techniques and tools to model and manipulate circular programs. These techniques transform circular programs into strict, purely functional programs. Program specialisation 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.

The techniques presented in this chapter have been implemented in the first *Haskell* based library to express attribute grammars functionally. Furthermore, we have used this library to construct new tools to manipulate and transform circular programs. Both the library and the tools are described in chapter 6.

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

Chapter 6

Tools and Libraries to Model and Manipulate Circular Programs

Summary

The techniques presented in the previous chapter were implemented in a reusable library, that we present here. We use this library to construct other tools, that we also describe in this chapter, and that incoporate circular programming. The tools we developed are used to transform circular programs that solve real programming problems into strict and strict deforested programs. The performance of these three types of programs was then benchmarked, and the results obtained are also presented here.

6.1 Introduction

In the previous chapter, we have presented techniques to transform a circular lazy program into a strict, multiple traversal program. In this chapter, we present the implementation of these techniques as a reusable and concise *Haskell* library: the *CircLib* library. Using this library, we have developed a simple attribute grammar based system and two strictification tools: *HaCirc* and *OCirc*. The *HaCirc* tool accepts as input a *Haskell* circular program and generates the two different types of strict programs presented in chapter 5: the strict multiple traversal program that uses intermediate gluing structures to convey information between different traversals and the deforested, higher-order strict program that performs multiple traversals without building such gluing structures. The second tool, *OCirc*, is an *Ocaml* version of *HaCirc*. Instead of *Haskell* circular programs, it accepts as input a circular program written in *Ocaml* notation. It produces a correct strict multiple traversal *Ocaml* program. In this way, we make the concise and elegant style of expressing multiple traversal algorithms also available to non-lazy functional programmers. Circular lazy functional programs perform the scheduling of the computations at execution time. In a strict, multiple traversal programming setting, the scheduling is performed statically. Moreover, the strict programs rely on additional mechanisms (gluing data structures or partially parameterized functions) to convey information between traversals, while the circular programs do not. In order to compare the performance of these equivalent programs, we perform the first systematic benchmark of circular, strict and higher-order programs. The results show that for algorithms relying on a large number of traversals the strict, higher-order programs are more efficient than the lazy ones, both in terms of runtime and memory consumption.

The attribute grammar system processes attribute grammars written in a notation similar to one used in the definition of the attribute grammar presented on page 7, and it generates strict (deforested or not) and lazy attribute evaluators.

This chapter is organized as follows. In section 6.2.1 we present the *CircLib* library; in section 6.2.2 we describe the *HaCirc* and *OCirc* tools and in section 6.2.3 we present the attribute grammar system that we have built on top of *CircLib*. In section 6.3 we describe in detail the benchmark experiments we have conducted and we present the results obtained. Section 6.4 concludes the chapter.

6.2 Libraries and tools for circular programming

In this section we present the library and the tools that we have constructed and that incorporate circular programming.

6.2.1 The *CircLib* library

We have implemented the techniques presented in chapter 5 in a reusable library, the *CircLib* library. *CircLib* is the first concise, easy to maintain and extend *Haskell* based library implementing attribute grammar techniques, namely the static scheduling of computations based on the well-known Kastens (1980) algorithm. The API of *CircLib* is presented in appendix.

The library introduces two data types to model circular programs and visit sequences in *Haskell*, and defines functions that implement all the formal definitions presented in chapter 5. The *Haskell* code in *CircLib* very much resembles its formal definition. As an example, recall the formal definition of the relation DP, given on page 110:

$$DP = \bigcup_{C \in Constructors} DP\ (C)$$

In this definition, *Constructors* is the set of the underlying circular program's constructors, and DP establishes dependency relations between the variables of the program. In *CircLib*, this definition has been implemented as follows:

```
dp :: CP → Rel Var Var
dp cp = let cs = constrs cp
            dps c = lookup c (deps cp)
        in Set.union (map (λc → dp_c (dps c)) cs)
```

We can see that *dp* is obtained by translating, in a straightforward fashion, the relational notation used in the definition of DP, into *Haskell* notation.

The *CircLib* library is generic and reusable. Indeed, we have used it to build several tools, that we present in the next section.

6.2.2 Tools for circular program manipulation

The *CircLib* library was used as the building block of several tools that incorporate circular programming. The library was used to implement tools, such as *HaCirc* and *OCirc*, that transform circular programs into strict ones and in the construction of a simple attribute grammar system. These three tools are briefly described next.

The *HaCirc* tool

The *HaCirc* tool is a *Haskell* refactor. It refactors circular programs into their 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 structures.

HaCirc is also a 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 higher-order deforested programs (i.e., programs with no intermediate, traversal gluing, structures).

We may use the *HaCirc* tool, for example, to transform the circular version of the semantic analyzer of Algol 68 derived in chapter 2 or the abstract tables formatter introduced in chapter 5 into strict and strict deforested equivalent programs. We may also use *HaCirc* to transform the *repmin* circular program presented in chapter 1 into the following program:

```
data LeafTree = Leaf Int
              | Fork (LeafTree, LeafTree)

data ShapeTree = SLeaf
               | SFork (ShapeTree, ShapeTree)

transform :: LeafTree → LeafTree
transform t = replace (tminst t)

tminst :: LeafTree → (ShapeTree, Int)
tminst (Leaf n)     = (SLeaf, n)
```

$$\begin{aligned} &\mathit{tminst}\ (\mathit{Fork}\ (l, r)) = \mathbf{let}\ (\mathit{sl}, m_1) = \mathit{tminst}\ l \\ &\qquad\qquad\qquad\qquad\qquad\ (\mathit{sr}, m_2) = \mathit{tminst}\ r \\ &\qquad\qquad\qquad\qquad\ \mathbf{in}\ (\mathit{SFork}\ (\mathit{sl}, \mathit{sr}), \mathit{min}\ m_1\ m_2) \end{aligned}$$

$$\begin{aligned} &\mathit{replace} :: (\mathit{ShapeTree}, \mathit{Int}) \rightarrow \mathit{LeafTree} \\ &\mathit{replace}\ (\mathit{SLeaf}, m) \qquad\quad = \mathit{Leaf}\ \ m \\ &\mathit{replace}\ (\mathit{SFork}\ (l, r), m) = \mathit{Fork}\ (\mathit{replace}\ (l, m), \mathit{replace}\ (r, m)) \end{aligned}$$

In the generated program, a shape tree is used to glue functions *tminst* and *replace* together. Indeed, a shape tree is the minimal intermediate data structure that the strict *repmin* can use. The techniques presented in chapter 5 always try to derive the minimal intermediate structures that can be used in the strict programs they derive.

The *OCirc* tool

In order to allow *Ocaml* programmers to write advanced and complex multiple traversal algorithms as elegant and concise circular programs, we have implemented a tool similar to *HaCirc* that transforms circular programs written in the *Ocaml* notation, into correct strict *Ocaml* programs.

We can use *OCirc* for example, to transform the *repmin* circular *program* expressed in *Ocaml* notation:

```
type tree = Leaf  of int
          | Fork of tree*tree
and r     = Root of tree

let rec transform t = match t with
        Root t -> let (replace,m) = repmin (t,m) in
                  replace

and repmin (t,m) = match t with
        Leaf n        -> (Leaf m, n)

      | Fork (t1,t2) -> let (replace1,m1) = repmin (t1,m) in
                        let (replace2,m2) = repmin (t2,m) in
                        (Fork (replace1,replace2),min (m1, m2));;
```

into a correct and executable *Ocaml* program that solves *repmin*:

```
type tree = Leaf  of int
          | Fork of tree*tree
and r     = Root of tree

type tree2 = Fork2  of tree2*tree2
           | Leaf2

let rec visit_t1 t = match t with
        Fork (t1, t2) -> let (t1', m1) = visit_t1 t1 in
                         let (t2', m2) = visit_t1 t2 in
                         let m = min m1 m2 in
                         (Fork2 (t1', t2'), m)
      | Leaf n -> (Leaf2, n)

and visit_t2 t m = match t with
        Fork2 (t1, t2) -> let t1' = visit_t2 t1 m in
                          let t2' = visit_t2 t2 m in
                          Fork (t1', t2')
      | Leaf2 -> Leaf m

and transform t = match t with
        Root t1 -> let (t2, m) = visit_t1 t1 in
                   let replace = visit_t2 t2 m in
                   replace
```

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 online[1]. The execution of such interactive versions of the tool(s) requires no further installation.

6.2.3 The AG system

The *CircLib* library was also used to construct a simple attribute grammar system. This system was implemented as briefly described next. First, at-

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

tribute grammars were modelled by a *Haskell* data-type. Then, we have constructed a generalized LR parser, using the *HaGlr* tool (Fernandes et al. 2004), to parse concrete attribute grammars into instances of the previously defined attribute grammar abstract data-type. Once attribute grammars are constructed, they can easily be transformed into elements of the *CP* data-type defined in *CircLib*. This means that it is then possible to transform them into *Haskell* and *Ocaml* based strict multiple traversal programs and *Haskell* lazy programs. Assembling all these components together, we obtain a very simple attribute grammar system.

In fact, the *Haskell* and *Ocaml* programs presented in the two previous sections are also the result of processing the attribute grammar of page 7.

6.3 Benchmarks

In this thesis, we have presented three different approaches to solve programming problems in a functional setting: using circular programs, strict multiple traversal programs and higher-order programs. Indeed, in chapters 2 and 3, strict programs were transformed, by calculation, into circular programs; in chapter 4, we studied the transformation of strict programs into higher-order ones and in chapter 5 circular programs were transformed into strict multiple traversal programs and into higher-order ones.

Circular programs are elegant and concise and do not use intermediate data structures. However, their computations need to be scheduled dynamically, at runtime, by the lazy engine. Strict programs use data structures to convey information between traversals and the scheduling of their computations is done statically. Higher-order programs are also scheduled statically, but they do not use intermediate structures. Instead, they use higher-order definitions to construct intermediate trees of function calls.

In this section, we present the first detailed benchmark comparing the execution performance of these three types of programs. Our benchmarks considered three examples that differ in the number of traversals they execute and also in the computational effort required to schedule their computations.

We compared the running performance of the circular *Table* formatter introduced in chapter 5 with the running performances of the strict and higher-order programs we derived from it. The *Table* circular program induces a two traversal program, that may be considered as a simple one. The details of the comparison and the results we obtained are presented in section 6.3.1.

We have also tested the performance of a circular program that processes a tiny subset of the *C* language, called *MicroC*, against the performance of the derived strict and higher-order programs. The *MicroC* processor induces a six traversal program, that is more demanding on the scheduling of computations than the *Table* formatter. The description of the conducted test and the results obtained are presented in detail in section 6.3.2.

Furthermore, we also compared circular, strict and higher-order versions of the pretty-printing combinators defined by Swierstra et al. (1999). The combinators offer the possibility of defining multiple printing layouts, and finding the optimal one may be an extremely hard-working task, as we will see in section 6.3.3. The performance benchmarks for this example are also presented in that section.

The results presented in sections 6.3.1 and 6.3.2 were obtained by running the programs in an Intel Centrino 1.4GHz with 512MB of RAM memory, under a Linux Mandrake 10.0 Operative System, and using the GHC 6.4 *Haskell* compiler. The results presented in section 6.3.3 were obtained in an Intel Core 2 Duo 1.83GHz with 1GB of RAM memory, running the Ubuntu Operative System and using the GHC 6.8.2 *Haskell* compiler.

6.3.1 The table formatter

The three *Table* formatters presented in chapter 5 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 6.1. The runtimes (in seconds) correspond to the accumulation of 5 executions. The memory consumption refers to the memory used in one run, and it was obtained with the built-in GHC memory profiler.

The results show that the three implementations have similar running

		Circular		Strict		Higher-Order	
	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 6.1: Performance results of the three different Table formatters

times, although the higher-order programs are always slightly faster than the others. In terms of memory consumption, the higher-order programs consume half of the memory needed by the circular programs. 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.

6.3.2 The *MicroC* processor

The *MicroC* language processor generates assembly for a simple stack-based machine and it includes an advanced pretty-printing algorithm. As input we consider typical *MicroC* programs, with 1360, 2720 and 4080 lines. The runtimes (in seconds) are the accumulation of 10 executions. The memory consumption refers to the memory used in one run, and it was obtained with the built-in GHC memory profiler.

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

Table 6.2: Performance results of the three *MicroC* processors.

The results presented in Table 6.2 show that the higher-order programs have the best running times of the different implementations of the *MicroC* processor: they are 2.8 times faster than the lazy equivalents. The higher-order implementations are also always more efficient than the strict ones: 2.6 times faster. One would expect, however, that the aggressive optimizations performed by GHC would be able to perform deforestation of the intermediate structures used in the strict implementations automatically. Indeed, several deforestation techniques, like the standard *fold/build* rule, are implemented in GHC. However, as one can see, for example, in the definition of the strict *Table* program, the function that builds the intermediate structure also returns additional results. Thus, the *fold/build* rule does not apply and the compiler is not able to perform such optimizations. In chapter 4, we have introduced a new rule, Law 4.2.1, defined in the style of *fold/build*, and that applies to programs whose consumer can be expressed in terms of a *pfold*. These programs may now get automatically transformed under GHC.

6.3.3 The pretty printing combinators

In this section, we benchmark different versions of the pretty printing combinators presented in (Swierstra et al. 1999). Pretty printing is a relevant problem in the compiler construction task, and consists in representing tree-based structures in a width-bounded area. The representation is performed in a top-down, left to right order, and is done in such a way that the layout clearly represents the logical structure of the tree. An example of this is presented next.

Suppose we want to pretty-print an **if then else** structure. This simple structure may be displayed with different layouts, depending on the page width. A layout is said to be optimal, or prettiest, if it takes the least number of lines, while still not overflowing the right page margins. Thus, with page width at least 18, the prettiest representation for the conditional structure is:

if c **then** t **else** e

If the page width is between 11 and 18, the prettiest representation for the same structure is now:

if c **then** t
  else e

Notice that it is impossible to display the **else** branch besides the **then** branch in less than 18 characters. If the page width is between 6 and 11, it is still possible to find a representation for the **if then else** structure. Indeed, it may be displayed as follows.

if c
then t
else e

The above representation, however, is the only one we can find for that structure, with such a page width. This also means that the conditional structure cannot be displayed on pages less than 6 characters wide.

In (Swierstra et al. 1999), the authors define a set of combinators for describing such layouts. We may use them to represent the three possible layouts for the conditional structure, defining:

$$
\begin{array}{ll}
pp = & \mathit{if}_c \Leftrightarrow \mathit{then}_t \Leftrightarrow \mathit{else}_e \\
\bigvee & \mathit{if}_c \Leftrightarrow \mathit{then}_t \Updownarrow \mathit{else}_e \\
\bigvee & \mathit{if}_c \Updownarrow \mathit{then}_t \Updownarrow \mathit{else}_e
\end{array}
$$

where

$\mathit{if}_c \quad = \mathit{txt}$ `"if c"`
$\mathit{then}_t = \mathit{txt}$ `"then t"`
$\mathit{else}_e = \mathit{txt}$ `"else e"`

The function *txt* converts strings into layouts, $\Leftrightarrow$ places its two arguments beside each other, $\Updownarrow$ places them above each other and $\bigvee$ combines two possible layouts. The implementation of such combinators receives as input a structure to display and the width of the page where the structure is to be displayed in. Then, it finds the prettiest representation for the structure in that page.

In our benchmarks, we have tested different implementations of these combinators: the circular lazy version and its strict multiple traversal and higher-order equivalents. We have used these different implementations to pretty-print the *Haskell* module List.hs. Every data-type and function defined in that module are pretty-printed using a strategy similar to the one we presented for the conditional structure. The definition of a function is displayed beside or below its type definition. Furthermore, the specification of a function (its type and its definition) is also displayed beside or below the specification of the function that precedes it in the module. This means that the implementation of the combinators, in order to resolve which layout is the prettiest, needs to consider the representation of the entire module. This is indeed a very hard-working task.

Our benchmark considered the pretty-printing of the first 11, 12 and 13 functions of List.hs. The results we obtained are presented in Table 6.3, where execution times are shown in seconds.

	Number of functions	Circular	Strict	Higher Order
Haskell	11	8.2	3.99	1.51
	12	129	26.89	3.85
	13	4216	67.02	6.31

Table 6.3: Performance results of the three implementations of the pretty printing combinators.

The results confirm that higher-order programs are more efficient than equivalent circular versions (up to 668 times!) and strict ones as well (up to 10.5 times). The results also show that the circular programs are the slowest among the three different implementations. Furthermore, the results obtained with this benchmark show an exponential growth of the running times of circular programs: pretty printing 12 functions is 16 times slower than doing it for 11 functions, and pretty printing 13 functions is 33 times slower than doing it for 12 functions. In fact, results also show that the circular implementation of the pretty printing combinators is the only one that exhibits this exponential growth.

The results presented in Table 6.3 also illustrate well the combinatorial explosion of possible layouts for representing the list module. Indeed, if we restrict the number of possible layouts by forcing the specification of a function to be displayed below the specification of the function that precedes it, the different implementations of the pretty printing combinators can determine the optimal layout in a straighforward way. This can be seen in the performance results presented in Table 6.4 (execution times are presented in seconds).

	Number of functions	Circular	Strict	Higher Order
Haskell	11	0.03	0.03	0.02
	12	0.04	0.04	0.02
	13	0.04	0.04	0.02

Table 6.4: Performance results of the three implementations of the pretty printing combinators on a simple example.

6.4 Conclusions

In this chapter, we have presented the implementation of the techniques introduced in chapter 5 as a *Haskell* library: the *CircLib* library. *CircLib* consists of a generic and reusable library, and it was used to construct two tools, *HaCirc* and *OCirc* that manipulate circular programs, and to implement an attribute grammar system. *HaCirc* manipulates *Haskell* circular programs and *OCirc* manipulates circular programs expressed in *Ocaml* notation. Both tools transform circular programs into strict multiple traversal programs and into higher-order programs. The relationship between circular, strict and higher-order programs was also explored in other chapters of this thesis: indeed, in chapters 2, 3 and 4, we have studied the transformation, by calculation, of strict programs into circular and into higher-order programs.

In this chapter, we have also presented the performance results of the experimental benchmark conducted comparing the three type of programs studied in this thesis. For this purpose, we have compared the execution performances of circular, strict and multiple traversal equivalent solutions to several programming problems. The results we obtained show that higher-order programs are more efficient than their strict multiple traversal and circular equivalents.

Chapter 7

Conclusions

Summary

In this chapter, we briefly review the work presented in this thesis and we draw our conclusions. We also discuss possible directions for future research in areas related to the work described in the thesis.

This thesis discussed the design, implementation and calculation of circular programs. In chapter 2, we have presented a new program transformation technique for intermediate structure elimination. The programs we are able of dealing with are defined as the composition of a producer with a consumer function. 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 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).

In chapter 3, 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 practical interest: our rules were used to calculate single traversal, circular programs in the context of monadic parsing and in the context of a programming environment.

In chapter 4, we have studied an alternative solution to achieve intermediate structure deforestation and multiple traversal elimination such that higher-order definitions are introduced instead of the circular definitions introduced in chapters 2 and 3. The programs we obtain are, therefore, higher-order programs, and they are obtained by the application of generic calculation rules. We have generalized the shortcut fusion rule for deriving pure higher-order programs presented in (Voigtländer 2008) so that it can be applied to compositions of programs with an arbitrary data type as intermediate structure. We have also presented an extension of the above rule for the derivation of higher-order programs to the case of monadic programs. We considered programs consisting of the composition between a monadic producer and a pure consumer and programs where both the producer and the consumer functions are monadic.

In chapter 5, we have presented techniques to model and manipulate circular programs. These techniques transform circular programs into strict, purely functional programs. Program specialisation and slicing techniques are used to improve the performance of the evaluators and to slice circular lazy programs, respectively.

The techniques presented in chapter 5 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 without having to define complex traversal scheduling strategies. Circular definitions are well-known and heavily used in the attribute grammar community. With this work we make this powerful style of programming available to other programming paradigms, namely the non-lazy functional one. *CircLib* was also used to construct a simple and purely functional attribute grammar system.

Finally, the first experimental benchmarks comparing the execution performance of circular, strict and higher-order programs have also been conducted. The results show that higher-order *Haskell* programs are more efficient than circular and strict equivalents. This is more evident for increasingly complex algorithms and for programs that rely on multiple traversal strategies and that define several intermediate data structures.

The *CircLib* library, the *HaCirc* and *OCirc* tools, the attribute grammar system and the benchmark results were presented in chapter 6.

7.1 Future work

The work presented in this thesis may be summarized by the following diagram:

$$\begin{tikzcd}
 & \textit{circular programs} \arrow[dl, "a)"'] \arrow[dr, "b)"] & \\
\textit{strict programs} \arrow[ur, dashed, "c)"'] \arrow[rr, dashed, "d)"] & & \textit{deforested programs}
\end{tikzcd}$$

The arrow *c*) corresponds to the calculational methods we presented in chapter 2, for pure programs, and in chapter 3, for monadic ones. The arrow *d*) corresponds to the methods for calculating higher-order programs presented in chapter 4, for both pure and monadic programs. Arrows *a*) and *b*) correspond to the attribute grammar based program manipulation techniques presented in chapter 5.

Various aspects of the ideas presented in this thesis deserve further elaboration.

Multiple intermediate structure elimination The examples we presented for motivating the calculational methods described in this thesis consist of compositions of a single producer and a single consumer function. We would like, however, to be able to achieve the same fusion goals for programs consisting of an arbitrary number of function compositions. Indeed, we are now studying how to generalize our work in order to optimize programs 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. We will describe such a generalization in a forthcoming paper, already under preparation.

Relation with Attribute Grammars Circular programs and attribute grammars (AGs) are closely related (Swierstra 2003). Indeed, AG techniques are used to model and manipulate circular programs in order to derive efficient non-lazy equivalent programs (Fernandes and Saraiva 2007), and several circular-based AG systems have been developed (Swierstra et al. 2004; Wyk et al. 2006). In particular, we would like to express the transformations presented in chapter 5 (arrows *a*) and *b*) in the diagram above) in a calculational form, so that their correctness can be proved. Indeed, although these techniques are largely used by the AG community, their correctness remains to be formally proved! The techniques studied in this thesis also serve the purpose of increasing the knowledge on the relationship between circular programs and AGs, and, therefore, bring us close to our goal of building a proved correct AG system.

Incremental Lazy Functional Programming Incremental computation is about maintaining the input-output relationship of a program, as the input undergoes changes. The changes in the input may be such that one cannot avoid a complete recomputation of the output. However, in many cases, one can reuse results of the previous computation to update the output more efficiently than by performing a complete recomputation from scratch. Obviously, incremental computation is more efficient for cases where changes in the input cause small changes in the output.

The investigation field of incremental computation has proven to be an exciting one, as, over the years, several researchers have studied and proposed techniques to reuse previously computed results, in order to improve efficiency of computer programs. Incremental Computation is, indeed, essential to the implementation of programming environments (Reps and Teitelbaum 1989; Michiel 2004) or spreadsheets, for example.

Change Propagation (Reps 1982), Adaptive Programming (Acar et al. 2002, 2006b,a) and Function Memoization (Hughes 1985; Pugh and Teitelbaum 1989; Saraiva et al. 2000) are among the techniques proposed to achieve Incremental Computation. However, the Change Propagation and Function Memoization approaches do not handle circular programs. Furthermore, Reps' techniques do not handle circular attribute grammars. Thus, the incremental functional implementations derived from incremental attribute grammars will never be circular programs.

Acar's ingenious Adaptive Programming technique is proposed in the strict functional setting ML and its implementation in *Haskell* (Carlsson 2002) does not support *lazyness*. Thus, his technique also does not allow to combine incrementality with the circular definitions that may occur in a *lazy* setting.

As for Memoization, Hughes' *lazy* memo-functions are especially suitable to manipulate circular (infinite) structures. However, these circularities are not of the same kind as the ones we want to be able to deal with: we exploit the use of function call results as some of the same call's arguments with the purpose of eliminating multiple traversals over data structures. It is still not clear how to memoize such circular function calls.

In the future, we aim to develop techniques that make it possible to combine incrementality with circular programming.

Circular Bidirectional Transformations There are many situations in which one data structure, called *source*, is transformed to another, called *view*, in such a way that changes on the view can be transformed back to those on the original data structure. This is called Bidirectional Transformation (BT), and practical examples include synchronization of replicated data in different formats (Foster et al. 2005), presentation-oriented structured document development (Hu et al. 2004; Michiel 2004), interactive user interface design (Meertens 1998), and the well-known view updating mechanism which has been intensively studied in the database community (Bancilhon and Spyratos 1981; Dayal and Bernstein 1982; Gottlob et al. 1988; Lechtenbörger and Vossen 2003).

Bidirectional transformations can also benefit from the properties of circular programs. Indeed, circular programs may provide an ideal setting to compute a new *source*, given the original one and it's *view*, but submitted to a particular change.

In the future, we intend to fully explore this promising research direction. Our plan is to formally establish how circular programs can be integrated within bidirectional transformations, namely how circular programs can be used in the backward transformation of BTs.

Bibliography

Samson Abramsky and Achim Jung. Domain theory. In *Handbook of Logic in Computer Science*, pages 1–168. Clarendon Press, 1994.

Umut A. Acar, Guy E. Blelloch, and Robert Harper. Adaptive functional programming. In *POPL'02: Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 247–259, New York, NY, USA, 2002. ACM.

Umut A. Acar, Guy E. Blelloch, Matthias Blume, and Kanat Tangwongsan. An experimental analysis of self-adjusting computation. In *PLDI'06: Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 96–107, New York, NY, USA, 2006a. ACM.

Umut A. Acar, Guy E. Blelloch, and Robert Harper. Adaptive functional programming. *ACM Transactions on Programming Languages and Systems (journal version of the POPL'02 article)*, 28(6):990–1034, 2006b.

Lex Augusteijn. Sorting morphisms. In Doaitse Swierstra, Pedro Henriques, and José Oliveira, editors, *Third Summer School on Advanced Functional Programming*, volume 1608 of *LNCS*, pages 1–27. Springer-Verlag, September 1998.

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

François Bancilhon and Nicolas Spyratos. Update semantics of relational views. *ACM Transactions on Database Systems (TODS)*, 6(4):557–575, 1981.

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

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

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

Magnus Carlsson. Monads for incremental computing. In *ICFP'02: Proceedings of the seventh ACM SIGPLAN International Conference on Functional Programming*, pages 26–35, New York, NY, USA, 2002. ACM.

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

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

Robin Cockett and Dwight 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.

Umeshwar Dayal and Philip A. Bernstein. On the correct translation of update operations on relational views. *ACM Transactions on Database Systems (TODS)*, 7(3):381–416, 1982.

Oege de Moor, Kevin Backhouse, and S. Doaitse Swierstra. First-class attribute grammars. *Informatica (Slovenia)*, 24(3), 2000.

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.

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

João Fernandes, João Saraiva, and Joost Visser. Generalized LR parsing in Haskell. In *Informal Proceedings of the Summer School on Advanced Functional Programming, students' presentation, pages 24-37*, 2004.

João Paulo Fernandes and João Saraiva. Tools and Libraries to Model and Manipulate Circular Programs. In *PEPM'07: Proceedings of the ACM SIGPLAN 2007 Symposium on Partial Evaluation and Program Manipulation*, pages 102–111. ACM Press, 2007.

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 Press.

Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. Combinators for bi-directional tree transformations: a linguistic approach to the view update problem. In *POPL'05:*

Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pages 233–246, New York, NY, USA, 2005. ACM.

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

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

Andrew 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.

Georg Gottlob, Paolo Paolini, Roberto Zicari, and Roberto Zicari. Properties and update semantics of consistent views. *ACM Transactions on Database Systems (TODS)*, 13(4):486–524, 1988.

Ralf Hinze and Johan Jeuring. Generic Haskell: Practice and theory. In *Summer School on Generic Programming*, 2002.

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.

Zhenjiang Hu, Shin-Cheng Mu, and Masato Takeichi. A programmable editor for developing structured documents based on bidirectional transformations. In *PEPM'04: Proceedings of the 2004 ACM SIGPLAN Symposium on Partial Evaluation and Program Manipulation*, pages 178–189, New York, NY, USA, 2004. ACM.

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

Graham Hutton and Erik 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.

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

Uwe Kastens, Peter Pfahler, and Matthias T. Jung. The eli system. In *CC '98: Proceedings of the 7th International Conference on Compiler Construction*, pages 294–297, London, UK, 1998. Springer-Verlag.

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 João Saraiva. Lrc - A Generator for Incremental Language-Oriented Tools. In Kay Koskimies, editor, *7th International Conference on Compiler Construction*, volume 1383 of *LNCS*, pages 298–301. Springer-Verlag, April 1998.

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 *Conference Record 7th ACM SIGPLAN/SIGARCH International Conference 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)*, volume 2053 of *LNCS*. Springer-Verlag, May 2001.

Jens Lechtenbörger and Gottfried Vossen. On the computation of relational view complements. *ACM Transactions on Database Systems (TODS)*, 28 (2):175–208, 2003.

Cecilia Manzino and Alberto 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. 1999.

Lambert Meertens. Designing constraint maintainers for user interaction. http://www.cwi.nl/∼lambert, 1998.

Martijn Michiel. *Proxima : a presentation-oriented editor for structured documents*. PhD thesis, Department of Computer Science, Utrecht University, The Netherlands, 2004.

Oege de Moor, Simon L. Peyton Jones, and Eric Van Wyk. Aspect-oriented compilers. In *GCSE '99: Proceedings of the First International Symposium on Generative and Component-Based Software Engineering*, pages 121–133, London, UK, 2000. Springer-Verlag.

Shin-Cheng Mu, Zhenjiang Hu, and Masato Takeichi. Bidirectional scripting for structured documents. In *Japanese Society for Software Science and Technology*, 22th Taikai, 2005.

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.

Yoshiyuki Onoue, Zhenjiang Hu, Hideya Iwasaki, and Masato 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 1997.

Jukka Paakki. Attribute grammar paradigms—a high-level methodology in language implementation. *ACM Comput. Surv.*, 27(2):196–255, 1995.

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

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

Alberto Pardo, João Paulo Fernandes, and João Saraiva. Shortcut fusion rules for the derivation of circular and higher-order monadic programs. In *PEPM'09: Proceedings of the 2009 ACM SIGPLAN Symposium on Partial Evaluation and Program Manipulation*, pages 81–90, New York, NY, USA, 2008. ACM Press.

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

Alberto Pettorossi and Andrzej Skowron. The lambda abstraction strategy for program derivation. In *Fundamenta Informaticae XII*, pages 541–561, 1987.

Simon Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003. Also in *Journal of Functional Programming*, 13(1).

Simon Peyton Jones. Call-pattern specialisation for haskell programs. In *ICFP'07: Proceedings of the the twelveth ACM SIGPLAN International Conference on Functional Programming*, pages 327–337, New York, NY, USA, 2007. ACM.

Simon Peyton Jones, John Hughes, Lennart Augustsson, et al. Report on the programming language Haskell 98. Technical report, February 1999.

William Pugh and Tim Teitelbaum. Incremental computation via function caching. In *POPL'89: Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 315–328, New York, NY, USA, 1989. ACM.

Thomas Reps. Optimal-time incremental semantic analysis for syntax-directed editors. In *POPL'82: Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 169–176, New York, NY, USA, 1982. ACM.

Thomas Reps and Tim 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.

João Saraiva, Doaitse Swierstra, and Matthijs Kuiper. Functional Incremental Attribute Evaluation. In David Watt, editor, *9th International*

Conference on Compiler Construction, CC/ETAPS'2000, volume 1781 of *LNCS*, pages 279–294. Springer-Verlag, March 2000.

Doaitse Swierstra. Tutorial on attribute grammars. In *Second International Conference on Generative Programming and Component Engineering (GPCE'03)*, September 2003.

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

Doaitse Swierstra and Olaf Chitil. Linear, bounded, functional pretty-printing. *Journal of Functional Programming*, 19(01):1–16, January 2009.

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.

Doaitse Swierstra, Arthur Baars, and Andres Löh. The UU-AG attribute grammar system, 2004. `http://www.cs.uu.nl/groups/ST`.

Akihiko Takano and Erik Meijer. Shortcut deforestation in calculational form. In *In Proc. Conference on Functional Programming Languages and Computer Architecture*, pages 306–313. ACM Press, 1995.

Frank Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3:121–189, 1995.

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.

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

Philip 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.

Eric Van Wyk, Lijesh Krishnan, Derek Bodin, Eric Johnson, August Schwerdfeger, and Phil Russell. Tool Demonstration: Silver Extensible Compiler Frameworks and Modular Language Extensions for Java and C. In *SCAM*, page 161, 2006.

Appendix A

The CircLib *Haskell* library

In this section we present the API of the *Haskell* library that implements the re-schedulling of the circular definitions. We start by defining a data-type CP, to represent circular programs, and the functions that manipulate it[1]:

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

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

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

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

$$\begin{aligned}
&dp\ \ :: CP \ \rightarrow\ Rel\ Var\ Var\\
&idp :: CP \ \rightarrow\ Rel\ Var\ Var\\
&ids :: CP \ \rightarrow\ Rel\ (DT, Name)\ (DT, Name)
\end{aligned}$$

[1]These functions correspond to the *Haskell* versions of the formal definitions presented in Section 5.3.3.

```
a    :: CP → DT → Int → Set (DT, Name)
ds   :: CP → DT → Rel (DT, Name) (DT, Name)
edp  :: CP → Rel Var Var
isOrdered :: CP → Bool
interface :: CP → DT → Interface

type Interface = [(Set (DT, Name), Set (DT, Name))]
```

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

```
data VisitSequences = VS (Map Constr [VisitSubSequence])

data VisitSubSequence = VSS{ n    :: Int,
                             prod :: [DT],
                             arg  :: [VarName],
                             res  :: [VarName],
                             instructions :: [Instruction]}

data Instruction = Eval { variable :: Var,
                          uses :: [Var]}
                 | Visit{ visit :: (Int, Int),
                          inp   :: [Name],
                          out   :: [Name]}

visit_sequences :: CP → VisitSequences
```

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

```
backward_slice :: CP → Criteria → VisitSequences

forward_slice  :: CP → Criteria → VisitSequences

type Criteria = [VarName]
```