Strictification of Circular Programs *

João Paulo Fernandes
Universidade do Porto &
Universidade do Minho, Portugal
jpaulo@fe.up.pt

João Saraiva
Universidade do Minho,
Portugal
jas@di.uminho.pt

Daniel Seidel Janis Voigtländer
Universität Bonn,
Germany
{ds,jv}@iai.uni-bonn.de

Abstract

Circular functional programs (necessarily evaluated lazily) have been used as algorithmic tools, as attribute grammar implementations, and as target for program transformation techniques. Classically, Richard Bird [1984] showed how to transform certain multi-traversal programs (which could be evaluated strictly or lazily) into one-traversal ones using circular bindings. Can we go the other way, even for programs that are not in the image of his technique? That is the question we pursue in this paper. We develop an approach that on the one hand lets us deal with typical examples corresponding to attribute grammars, but on the other hand also helps to derive new algorithms for problems not previously in reach.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features; D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming

General Terms Design, Languages

Keywords program transformation

1. Introduction

Circular programs were first introduced by Bird [1984] to avoid multiple traversals originating from nested function calls. He fuses several traversals of the same input data structure by tupling their results and applying *unfold*/*fold*-transformation steps [Burstall and Darlington 1977]. Possible intra-traversal dependencies—if information gathered in one traversal is used in another—are captured by circular definitions in the transformed program, which given certain conditions are well-behaved under a lazy evaluation strategy. Subsequently, this kind of transformation was recast in terms of attribute grammars [Johnsson 1987; Kuiper and Swierstra 1987], and indeed circular programs have become one successful implementation technique for attribute grammars [de Moor et al. 2000; Saraiva 1999]. Circular programs have also been used as an algorithmic tool [Jones and Gibbons 1993; Okasaki 2000] and as target for transformation techniques other than pure elimination of multiple traversals [Fernandes et al. 2007; Pardo et al. 2009; Voigtländer 2004].

In this paper, we are interested in transforming circular programs into non-circular ones. In essence, we want to go in the opposite direction of the transformation that Bird [1984] proposed (and that Chin et al. [1999] systematized, and made more effective by exploiting strictness analysis). Why would we be interested in that, other than out of curiosity? We do care about efficiency: it is well known that circular programs, while nominally avoiding multiple traversals, can actually lead to high space and time costs through introduction of extra thunks, countermanding any potential benefit. Specifically, when looking for an implementation strategy for attribute grammar systems, lazily evaluated circular programs are an easy, but not necessarily the most practical route. Instead, one may ultimately want to go for a strict functional language as target. An early approach for such strictification is already inherent in the work of Kuiper and Swierstra [1987]. They provide two mappings from attribute grammars to functional programs: one that leads to a possibly multi-traversal, non-circular program and one that leads to a single-traversal, typically circular program. To get from a circular program to a non-circular one, it may be possible to apply the second mapping in reverse and then the first mapping in forward mode. Actually, that is exactly the opposite of the use that Kuiper and Swierstra propose to make of their mappings, and of course, it will only work if the circular program at hand is indeed the image of some non-circular program under the respective mappings in the opposite directions. Similarly, trying to somehow "simply" invert the original transformation technique of Bird [1984] would up front limit the class of programs we could hope to deal with. Hence, we are instead looking for an independent approach to eliminating circular definitions.

The latter also sets our work here apart from earlier work of Fernandes and Saraiva [2007]. They used attribute grammar techniques to transform lazy circular programs into programs executable both lazily (e.g., in Haskell) and strictly (e.g., in OCaml). Essentially, they recover the attribute grammar (dependencies) that correspond to a given circular program, ostensibly only evaluable in a lazy language, via syntactic analysis. Then, they use a (complex) scheduling algorithm from the attribute grammar world [Kastens 1980] to statically determine an admissible evaluation order, and implement it via a non-circular functional program. Thus, the need for a lazy evaluation engine to determine an admissible evaluation order dynamically at runtime is avoided. Again, this approach only works for circular programs that already correspond to an attribute grammar in a rather direct way. Instead, our aim here is to deal more generally with circular programs, however arising, also ones which do not correspond to an attribute grammar or are images of some non-circular program under any of the known techniques for going from non-circular to circular. A case in point is our dealing with a circular breadth-first tree numbering program due to Jones and Gibbons [1993] and Okasaki [2000].

* This work was supported as a joint project by Fundação para a Ciência e Tecnologia (FCT), grant No. FCT/Proc 441.00, and Deutscher Akademischer Austausch Dienst (DAAD), grant No. 50106501. The Portuguese partners are also supported under the SSaaPP research project, PTDC/EIA-CCO/108613/2008. Additionally, João Paulo Fernandes was supported by FCT, grant No. SFRH/BPD/46987/2008, and Daniel Seidel was supported by Deutsche Forschungsgemeinschaft (DFG), grant No. VO 1512/1-1.

We start in Section 2 by considering the classical *repmin* example and using it to introduce our approach to transforming circular programs into non-circular ones. We employ type-based analysis and general program transformation techniques. Our transformation in this example could be completely automatic, and indeed this is the case for typical examples that would also be in the reach of attribute grammar techniques. To emphasize this point, we consider a practical example from a programming language environment in Section 3. But automation, or even just formal presentation of a fixed transformation technique, is not our goal here. Rather, we are interested in the interplay and connection of program manipulation techniques with the aim of transforming circular into non-circular programs. Indeed, we perform an extended case study in Section 4 for an algorithmic circular programming idea originally due to Jones and Gibbons [1993] and then used by Okasaki [2000] in a comparison of breadth-first numbering algorithms. The circular program considered there is particular in that it does not correspond to an attribute grammar, nor can it be seen as an image of a multi-traversal program under the transformation of Bird [1984]. Instead, it embodies an independent algorithmic use of circular definitions. As such, it poses a challenging problem, and it turns out that in deriving a non-circular version from it we have to invest some creativity. The overarching approach, however, will still be the one from Section 2, and the bits of creativity we invest will pay off in terms of interesting algorithmic variations we arrive at. Interestingly, Okasaki [2000] discussed various implementations, in a strict language, of breadth-first numbering that he and others had come up with. He only gave the circular program for comparison, without relating it in any way to any of the strict algorithms. With our derivations, we can actually bridge the gap and get, from the circular program, new alternatives for strict implementation of breadth-first numbering.

As languages in this paper, we use Haskell and—to emphasize when programs are non-circular and can be evaluated strictly—OCaml. To also be able to make concrete statements about efficiency, we provide measurements and discussion in Section 5. We conclude in Section 6 and give a perspective for using our transformation approach from this paper as a facilitator for further optimization techniques.

Before we start, it seems worth pointing out that our approach here is orthogonal to the lambda-abstraction strategy of Pettorossi and Proietti [1988]. They avoid circular definitions by transforming multi-traversal, non-circular programs into single-traversal, non-circular programs that use higher-order. We instead go from single-traversal, circular programs to multi-traversal, non-circular programs that stay first-order. Of course, one could envision combining these transformation routes to replace circularity by higher-orderness while preserving single-traversal behavior.

2. Our General Strategy, and an Example

Our general strategy for transforming a lazy circular into a strict non-circular program is:

1. Find out which parts of the output of a circular call depend on which parts of its input. (Several approaches would be possible to do so, for example classical syntactic dependency analysis. Our main approach will be type-based.)
2. Naively split the circular call into several ones, each computing only one of the outputs, and exploit the information gained in the first step to decouple these different calls.
3. Specialize the different calls (using slicing, partial evaluation, ...) to work only with those pieces of input and output that are actually relevant in each case.

We demonstrate it based on the following circular program, due to Bird [1984]:

$$\mathbf{data}\ \mathsf{Tree}\ a = \mathsf{Leaf}\ a \mid \mathsf{Fork}\ (\mathsf{Tree}\ a)\ (\mathsf{Tree}\ a)$$

$$\begin{aligned}
&\mathit{repmin} :: \mathsf{Tree\ Int} \to \mathsf{Int} \to (\mathsf{Tree\ Int}, \mathsf{Int})\\
&\mathit{repmin}\ (\mathsf{Leaf}\ n)\quad m = (\mathsf{Leaf}\ m, n)\\
&\mathit{repmin}\ (\mathsf{Fork}\ l\ r)\ m = (\mathsf{Fork}\ l'\ r', \mathit{min}\ m_1\ m_2)\\
&\qquad\qquad\qquad\qquad \mathbf{where}\ (l', m_1) = \mathit{repmin}\ l\ m\\
&\qquad\qquad\qquad\qquad\qquad\quad (r', m_2) = \mathit{repmin}\ r\ m
\end{aligned}$$

$$\begin{aligned}
&\mathit{run} :: \mathsf{Tree\ Int} \to \mathsf{Tree\ Int}\\
&\mathit{run}\ t = \mathbf{let}\ (nt, m) = \mathit{repmin}\ t\ m\ \mathbf{in}\ nt
\end{aligned}$$

Our goal is to remove the circularity in run. So we need to find out which parts of the output of the circular call

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

depend on which parts of its input. For this example, there is a very easy way to exploit type information. Note that the type $\mathsf{Tree\ Int} \to \mathsf{Int} \to (\mathsf{Tree\ Int}, \mathsf{Int})$ assigned to repmin above is not the most general one that would be possible. Indeed, if we omit the explicit type signature and ask Haskell to infer a type from the defining equations of repmin, the answer will be:

$$\mathit{repmin} :: \mathsf{Ord}\ a \Rightarrow \mathsf{Tree}\ a \to b \to (\mathsf{Tree}\ b, a)$$

If we had used a non-polymorphic version min specialized to type Int, the answer would have been:

$$\mathit{repmin} :: \mathsf{Tree\ Int} \to b \to (\mathsf{Tree}\ b, \mathsf{Int})$$

In any case, we can see that the second output of repmin cannot possibly depend on the second input of repmin (unless Haskell's arguably impure strict evaluation primitive seq would be used, which we assume not to be the case for the programs we transform). The argument is that there is no way how the polymorphic input of unknown type b could be used to influence the computation of an Int. (We will later make use of a more precise formulation of this kind of reasoning.) What we can deduce from this is that for any $t :: \mathsf{Tree\ Int}$ and m_1, m_2 of the same (but arbitrary) type,

$$\mathit{snd}\ (\mathit{repmin}\ t\ m_1)\ \equiv\ \mathit{snd}\ (\mathit{repmin}\ t\ m_2)$$

Indeed, for any t and m,

$$\mathit{snd}\ (\mathit{repmin}\ t\ m)\ \equiv\ \mathit{snd}\ (\mathit{repmin}\ t\ \bot) \tag{1}$$

for the undefined value $\bot$.

This information comes in very useful. Note that we could always equivalently (but less efficiently, with two traversals, and still circular) have written the definition of run above as:

$$\begin{aligned}
&\mathit{run} :: \mathsf{Tree\ Int} \to \mathsf{Tree\ Int}\\
&\mathit{run}\ t = \mathbf{let}\ (nt, _) = \mathit{repmin}\ t\ m\\
&\qquad\qquad\ (_, m) = \mathit{repmin}\ t\ m\\
&\qquad \mathbf{in}\ nt
\end{aligned}$$

But now we can use (1) to deduce that this is equivalent to:

$$\begin{aligned}
&\mathit{run} :: \mathsf{Tree\ Int} \to \mathsf{Tree\ Int}\\
&\mathit{run}\ t = \mathbf{let}\ (nt, _) = \mathit{repmin}\ t\ m\\
&\qquad\qquad\ (_, m) = \mathit{repmin}\ t\ \bot\\
&\qquad \mathbf{in}\ nt
\end{aligned}$$

which is a non-circular program.

It is not a particularly efficient program, of course, because it twice uses the full repmin though only parts of the inputs and outputs are actually relevant in each case. Let us try to remedy this. First for the second call, namely $(\_, m) = \mathit{repmin}\ t\ \bot$. Clearly, it would be beneficial to have a more specialized function, $\mathit{repmin}_{\mathit{snd}}$, which takes only a tree argument and produces only

the second output of the original $repmin$. Of course, we can easily define such a function:

$$
\begin{array}{l}
repmin_{snd} :: \mathsf{Tree\ Int} \to \mathsf{Int} \\
repmin_{snd}\ t = snd\ (repmin\ t\ \bot)
\end{array}
$$

and then replace the above call by simply $m = repmin_{snd}\ t$. Moreover, using standard techniques (general unfold/fold-transformations [Burstall and Darlington 1977], or even an algorithmic variant [Pettorossi and Proietti 1996]), one can derive from the above a direct definition of $repmin_{snd}$:

$$
\begin{array}{l}
repmin_{snd} :: \mathsf{Tree\ Int} \to \mathsf{Int} \\
repmin_{snd}\ (\mathsf{Leaf}\ n)\ \ = n \\
repmin_{snd}\ (\mathsf{Fork}\ l\ r) = min\ (repmin_{snd}\ l)\ (repmin_{snd}\ r)
\end{array}
$$

Similarly, we can replace the other call in the above definition of run, namely $(nt, \_) = repmin\ t\ m$, by $nt = repmin_{fst}\ t\ m$ with:

$$
\begin{array}{l}
repmin_{fst} :: \mathsf{Tree\ Int} \to b \to \mathsf{Tree}\ b \\
repmin_{fst}\ (\mathsf{Leaf}\ n)\ \ m = \mathsf{Leaf}\ m \\
repmin_{fst}\ (\mathsf{Fork}\ l\ r)\ m = \mathsf{Fork}\ (repmin_{fst}\ l\ m) \\
\qquad\qquad\qquad\qquad\qquad\quad (repmin_{fst}\ r\ m)
\end{array}
$$

Ultimately, by simple inlining, this leads to a program consisting of $repmin_{fst}$, $repmin_{snd}$, and:

$$
\begin{array}{l}
run :: \mathsf{Tree\ Int} \to \mathsf{Tree\ Int} \\
run\ t = repmin_{fst}\ t\ (repmin_{snd}\ t)
\end{array}
$$

which is non-circular, can be evaluated strictly, and indeed corresponds exactly to the program from which Bird [1984] derived the circular version that we started this section with. It can trivially be rewritten in OCaml as:

```
type 'a tree = Leaf of 'a | Fork of ('a tree) * ('a tree)

let rec repmin_fst t m =
  match t with
    Leaf n      -> Leaf m
  | Fork (l, r) -> Fork (repmin_fst l m, repmin_fst r m)

let rec repmin_snd t =
  match t with
    Leaf n      -> n
  | Fork (l, r) -> min (repmin_snd l) (repmin_snd r)

let run t = repmin_fst t (repmin_snd t)
```

Our plan is to apply the approach demonstrated on the $repmin$ example to more complicated programs. As seen above, the key is the discovery of dependencies between inputs and outputs, in a preferably lightweight manner. In the example, we have used a type-based argument, namely that from the inferred function type

$$repmin :: \mathsf{Tree\ Int} \to b \to (\mathsf{Tree}\ b, \mathsf{Int})$$

we can see that the second output cannot depend on the second input. We also promised that there is a precise formulation for such reasoning. Indeed, we next review work in short that provides the desired information systematically.

2.1 Type-Based Useless-Variable Elimination

Kobayashi [2001] proposed a method for detecting dead code via type inference. The basic idea is that if some subexpression in a program can be replaced by a special value () of a special type () without affecting the type of the "$main$-expression" of the program, then it is guaranteed that the subexpression in question has no impact whatsoever on the result computed by the program. Instead of the special value () of the special type () we employ the undefined value $\bot$ of polymorphic type, otherwise our procedure is exactly as Kobayashi's. Our intended use is a bit different: rather than detecting dead code as such, we want to discover input-output-dependencies. But, of course, the latter problem can be reduced to the former: we simply surround a function call of interest with a projection onto some of its output components, pose the resulting expression as $main$-expression, and any pieces of the input that will be detected as useless then are known to not influence the part of the output we are interested in. Concretely, for the $repmin$ example we invoke Kobayashi's method on both

$$
\begin{array}{l}
\mathbf{let}\ repmin\ (\mathsf{Leaf}\ n)\ \ m = (\mathsf{Leaf}\ m, n) \\
\quad\ repmin\ (\mathsf{Fork}\ l\ r)\ m = (\mathsf{Fork}\ l'\ r', min\ m_1\ m_2) \\
\qquad\qquad\qquad\qquad\qquad\quad \mathbf{where}\ (l', m_1) = repmin\ l\ m \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad (r', m_2) = repmin\ r\ m \\
\mathbf{in}\ fst\ (repmin\ t\ m)
\end{array}
$$

and

$$
\begin{array}{l}
\mathbf{let}\ repmin\ (\mathsf{Leaf}\ n)\ \ m = (\mathsf{Leaf}\ m, n) \\
\quad\ repmin\ (\mathsf{Fork}\ l\ r)\ m = \ldots \\
\mathbf{in}\ snd\ (repmin\ t\ m)
\end{array}
$$

The output (with $\bot$ instead of (), as mentioned above) is:

$$
\begin{array}{l}
\mathbf{let}\ repmin\ (\mathsf{Leaf}\ n)\ \ m = (\mathsf{Leaf}\ m, \bot) \\
\quad\ repmin\ (\mathsf{Fork}\ l\ r)\ m = (\mathsf{Fork}\ l'\ r', \bot) \\
\qquad\qquad\qquad\qquad\qquad\quad \mathbf{where}\ (l', m_1) = repmin\ l\ m \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad (r', m_2) = repmin\ r\ m \\
\mathbf{in}\ fst\ (repmin\ t\ m)
\end{array}
$$

and

$$
\begin{array}{l}
\mathbf{let}\ repmin\ (\mathsf{Leaf}\ n)\ \ m = (\bot, n) \\
\quad\ repmin\ (\mathsf{Fork}\ l\ r)\ m = (\bot, min\ m_1\ m_2) \\
\qquad\qquad\qquad\qquad\qquad\quad \mathbf{where}\ (l', m_1) = repmin\ l\ m \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad (r', m_2) = repmin\ r\ m \\
\mathbf{in}\ snd\ (repmin\ t\ \bot)
\end{array}
$$

respectively. That is how we learn that the second output of $repmin$ does not depend on its second input. Also, Kobayashi's method comes with an optimality statement, in the sense that it finds the maximum of what is possible in terms of replacing subexpressions with () while preserving the type of the overall expression. So from the above result we can also deduce that the first output of $repmin$ indeed depends on both its inputs.

Moreover, Kobayashi [2001] also presents an algorithm that actually eliminates the detected dead code. Basically, it simply removes all function arguments and results that were singled out as special during type inference. Continuing our example, this leads exactly to the functions $repmin_{fst}$ and $repmin_{snd}$ shown earlier in Section 2, for use in replacements of the calls $fst\ (repmin\ t\ m)$ and $snd\ (repmin\ t\ \bot)$.

We see that much of what we need for our strictification toolbox does already exist. We add the ideas of splitting a circular call into separate ones, decoupling, and putting all the pieces together. In fact, our contribution is a way of combining known approaches for general program analysis and transformation such that strictification of circular programs becomes possible. As we will see with more complicated examples later on, one cannot always use Kobayashi's method and/or unfold/fold-transformations "out of the box", though.

3. A Simple Programming Environment

In this section, we apply our approach to a circular program of practical interest, one that deals with the scope rules of a sim-

ple programming language.[1] A program in that language consists of a sequence of instructions, where each instruction may either be the declaration or the use of a variable, e.g., $p = [\mathbf{use}\ x; \mathbf{decl}\ x; \mathbf{decl}\ x; \mathbf{use}\ y;]$. Such programs may be described by the following data type:

$\mathbf{type}\ \mathsf{Prog} = [\mathsf{It}]$
$\mathbf{data}\ \mathsf{It} = \mathsf{Decl}\ \mathsf{Var} \mid \mathsf{Use}\ \mathsf{Var}$
$\mathbf{type}\ \mathsf{Var} = \mathsf{String}$

Now, in order to be well formed, programs in the language, or Prog values, should obey the following scope rules:

1. all variables used must be declared. The declaration of a variable, however, may occur after its first use.
2. a variable must be declared at most once.

We aim to develop a semantic function that analyzes a sequence of instructions and computes a list containing the variable identifiers of the instructions which do not obey the above rules. 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 $[x, y]$: variable x has been declared twice, and the use of variable y has no binding occurrence at all.

The list of semantic errors encountered in a program (representable as $\mathbf{type}\ \mathsf{Errors} = [\mathsf{Var}]$), is obtained by checking, for each variable declaration, whether it has already appeared or not. For this, our implementation needs to go on accumulating (in an element of $\mathbf{type}\ \mathsf{Env} = [\mathsf{Var}]$) the variables that are declared in a program. Furthermore, each variable that is used must be declared somewhere in the program, so we need to know the global environment of the program (the list of all variables declared in it).

The following program implements the desired semantic analysis. A circular call is defined in run so that the global environment of an instruction sequence is used while still being constructed.

```
sem :: Prog → Env → Env → (Errors, Env)
sem []              dcls _ = ([], dcls)
sem (Decl var : p) dcls env_g
  = let (errs_p, env_p) = sem p (var : dcls) env_g
        errs_prog = if var ∈ dcls then var : errs_p else errs_p
    in (errs_prog, env_p)
sem (Use var : p)  dcls env_g
  = let (errs_p, env_p) = sem p dcls env_g
        errs_prog = if var ∈ env_g then errs_p else var : errs_p
    in (errs_prog, env_p)

run :: Prog → Errors
run prog = let (errs, env) = sem prog [] env in errs
```

Our goal is now to transform this program into a non-circular one. We follow the same derivation procedure as in the previous section, and obtain:

```
sem_snd :: Prog → Env → Env
sem_snd []              dcls = dcls
sem_snd (Decl var : p) dcls = sem_snd p (var : dcls)
sem_snd (Use var : p)  dcls = sem_snd p dcls

sem_fst :: Prog → Env → Env → Errors
sem_fst []              _    _ = []
sem_fst (Decl var : p)  dcls env_g
  = let errs_p = sem_fst p (var : dcls) env_g
    in if var ∈ dcls then var : errs_p else errs_p
sem_fst (Use var : p)  dcls env_g
  = let errs_p = sem_fst p dcls env_g
    in if var ∈ env_g then errs_p else var : errs_p

run :: Prog → Errors
run prog = sem_fst prog [] (sem_snd prog [])
```

As we see, the strictification procedure was able to realize that in a non-circular setting the global environment needs to be available (totally, due to the use-before-declare discipline) before semantic errors can be computed; it also tells us *how* that environment can be obtained.

The above program makes no essential use of lazy evaluation, and can be rewritten as an OCaml program (which we omit here due to space restrictions). In the next section we show that the principles we have been using so far still apply to circular programs that do not correspond directly to attribute grammars.

4. Breadth-First Numbering

Inspired by the work of Jones and Gibbons [1993] on breadth-first labelling, Okasaki [2000] gives the following circular program for numbering the inner nodes of a tree in breadth-first order:[2]

```
data Tree a = Empty | Fork a (Tree a) (Tree a)

bfn :: Tree a → [Int] → (Tree Int, [Int])
bfn Empty        ks        = (Empty, ks)
bfn (Fork _ l r) ~(k : ks) = (Fork k l' r', (k + 1) : ks'')
                             where (l', ks')  = bfn l ks
                                   (r', ks'') = bfn r ks'

run :: Tree a → Tree Int
run t = let (nt, ks) = bfn t (1 : ks) in nt
```

4.1 A First Approach: Offsets

As mentioned in the introduction, a bit of creativity is needed to deal with the above circular program. Driven by the observation that the second output of bfn is "somehow" obtained from its second input by incrementing list elements, potentially repeatedly, we first derive a variant of bfn which in its second output returns just those increments/offsets, rather than the result of actually adding them to the second input. The desired relationship between the two functions is:

$$bfn\ t\ ks\ \equiv\ \mathbf{let}\ (nt, ds) = bfn_{Off}\ t\ ks\ \mathbf{in}\ (nt, zipPlus\ ks\ ds)$$

where

```
zipPlus :: [Int] → [Int] → [Int]
zipPlus []       ds       = ds
zipPlus ks       []       = ks
zipPlus (k : ks) (d : ds) = (k + d) : (zipPlus ks ds)
```

The desired function is obtained pretty straightforwardly as follows:

```
bfn_Off :: Tree a → [Int] → (Tree Int, [Int])
bfn_Off Empty        ks        = (Empty, [])
bfn_Off (Fork _ l r) ~(k : ks) = (Fork k l' r',
                                  1 : (zipPlus ds ds'))
  where (l', ds)  = bfn_Off l ks
        (r', ds') = bfn_Off r (zipPlus ks ds)
```

[1] Due to space limitations, we consider a simplified version of the Algol 68 rules only. The complete definition is given by de Moor et al. [2000], and is used in the Eli attribute grammar-based system [Kastens et al. 2007].

[2] We use a lazy pattern match (notation: $\sim(k : ks)$) in the second equation of bfn, where Okasaki uses a strict one. The lazy version is more convenient for our derivation later on.

and can be used inside run as follows:

$$
\begin{array}{l}
run :: \mathsf{Tree}\ a \to \mathsf{Tree\ Int} \\
run\ t = \mathbf{let}\ (nt, ds) = bfn_{Off}\ t\ (1 : ks) \\
\qquad\qquad\ ks\ \ \ \ \ \ = zipPlus\ (1 : ks)\ ds \\
\qquad\ \mathbf{in}\ nt
\end{array}
$$

We see that there are now essentially two apparent circular dependencies: ds appears to depend on ks and ks on ds, plus ks depends on itself. Let us first deal with the former. Splitting the call to bfn_{Off} as follows:

$$
\begin{array}{l}
run :: \mathsf{Tree}\ a \to \mathsf{Tree\ Int} \\
run\ t = \mathbf{let}\ (nt, \_) = bfn_{Off}\ t\ (1 : ks) \\
\qquad\qquad\ (\_, ds) = bfn_{Off}\ t\ (1 : ks) \\
\qquad\qquad\ ks\ \ \ \ \ \ = zipPlus\ (1 : ks)\ ds \\
\qquad\ \mathbf{in}\ nt
\end{array}
$$

and applying the "type-based analysis plus specialization" approach from Section 2 (more specifically, involving Kobayashi's analysis as discussed in Section 2.1, since pure Haskell type inference is not enough to provide the required information here, due, e.g., to the call $zipPlus\ ks\ ds$ in bfn_{Off}) leads to:[3]

$$
\begin{array}{l}
run :: \mathsf{Tree}\ a \to \mathsf{Tree\ Int} \\
run\ t = \mathbf{let}\ nt = fst\ (bfn_{Off}\ t\ (1 : ks)) \\
\qquad\qquad\ ds = bfn_{Off,snd}\ t \\
\qquad\qquad\ ks = zipPlus\ (1 : ks)\ ds \\
\qquad\ \mathbf{in}\ nt
\end{array}
$$

$$
\begin{array}{l}
bfn_{Off,snd} :: \mathsf{Tree}\ a \to [\mathsf{Int}] \\
bfn_{Off,snd}\ \mathsf{Empty}\ \ \ \ \ \ = [\,] \\
bfn_{Off,snd}\ (\mathsf{Fork}\ \_\ l\ r) = 1 : (zipPlus\ ds\ ds') \\
\quad \mathbf{where}\ ds\ = bfn_{Off,snd}\ l \\
\qquad\qquad\ ds' = bfn_{Off,snd}\ r
\end{array}
$$

Note that it was not possible to specialize the call $fst\ (bfn_{Off}\ t\ (1 : ks))$ to some function $bfn_{Off,fst}$ with fewer input dependencies. On the good side, we have managed to eliminate the circularity between ks and ds, being left with only the circular dependency of ks on itself in the equation $ks = zipPlus\ (1 : ks)\ ds$. Let us look at that equation in a bit more detail, in particular "expanding" the lists to see how their elements relate to each other:

$$
\begin{array}{ll}
 & [k_0, k_1, \ldots] \\
\equiv & zipPlus\ [1, k_0, k_1, \ldots]\ [d_0, d_1, \ldots, d_n] \\
\equiv & (1 + d_0) : (zipPlus\ [k_0, k_1, \ldots]\ [d_1, \ldots, d_n]) \\
\equiv & (1 + d_0) : ((1 + d_0) + d_1) : (zipPlus\ [k_1, \ldots]\ [d_2, \ldots, d_n]) \\
\equiv & (1 + d_0) : ((1 + d_0) + d_1) : (((1 + d_0) + d_1) + d_2) : \\
 & (zipPlus\ [k_2, \ldots]\ [d_3, \ldots, d_n]) \\
\equiv & \ldots \\
\equiv & (tail\ (scanl\ (+)\ 1\ [d_0, d_1, \ldots, d_n])) \\
 & +\!\!+ (zipPlus\ [k_n, \ldots]\ [\,]) \\
\equiv & (tail\ (scanl\ (+)\ 1\ ds)) +\!\!+ [k_n, \ldots] \\
\equiv & (tail\ (scanl\ (+)\ 1\ ds)) +\!\!+ [last\ (scanl\ (+)\ 1\ ds), \ldots] \\
\equiv & (tail\ (scanl\ (+)\ 1\ ds)) +\!\!+ (repeat\ (last\ (scanl\ (+)\ 1\ ds)))
\end{array}
$$

Note that the last line contains no elements from $[k_0, k_1, ...]$, so we have discovered a non-circular definition for ks. Using it instead of the equation $ks = zipPlus\ (1 : ks)\ ds$ leads to a version of run that does not anymore contain circular definitions at all. Admittedly, arriving at the above takes some creativity. But since both $scanl$ and $zipPlus$ / $zipWith$ are pretty well known functions, the discovery that the circular binding involving $zipPlus$ can be replaced with straight calls to $scanl$ is actually not all too far-fetched.

The only thing that now seems to prevent us from executing run in a strict language is the call to $repeat$, which creates an infinite list. Actually, in OCaml this is not a real problem, because despite being strict, OCaml has some simple support for infinite lists. However, we can actually do away with infiniteness completely, because it is easy to see that this part of the list ks will not actually ever be needed. After all, bfn_{Off} never consumes more elements from its second argument than $bfn_{Off,snd}$ produces (for the same input tree, in the list ds). Hence, we can finally rewrite run into:

$$
\begin{array}{l}
run :: \mathsf{Tree}\ a \to \mathsf{Tree\ Int} \\
run\ t = \mathbf{let}\ nt = fst\ (bfn_{Off}\ t\ (1 : ks)) \\
\qquad\qquad\ ks = tail\ (scanl\ (+)\ 1\ (bfn_{Off,snd}\ t)) \\
\qquad\ \mathbf{in}\ nt
\end{array}
$$

where bfn_{Off} and $bfn_{Off,snd}$ are the functions shown earlier in this subsection.

The version of the program that we have now arrived at reads as follows when transliterated to OCaml:

```
open List

type 'a tree = Fork of 'a * ('a tree) * ('a tree) | Empty

let rec zipPlus ks ds =
  match ks with
    []        -> ds
  | (k :: ks') -> match ds with
                    []         -> ks
                  | (d :: ds') -> (k + d) :: zipPlus ks' ds'

let rec bfn_Off t ks =
  match t with
    Empty          -> (Empty, [])
  | Fork (_, l, r) ->
      let ks' = tl ks in
      let (l', ds) = bfn_Off l ks' in
      let (r', ds') = bfn_Off r (zipPlus ks' ds) in
      (Fork (hd ks, l', r'), 1 :: (zipPlus ds ds'))

let rec bfn_Off_snd t =
  match t with
    Empty          -> []
  | Fork (_, l, r) ->
      let ds  = bfn_Off_snd l and
          ds' = bfn_Off_snd r
      in 1 :: (zipPlus ds ds')

let rec scanl f n xs =
  match xs with
    []        -> [n]
  | (x :: xs') -> n :: (scanl f (f n x) xs')

let run t = fst (bfn_Off t (scanl (+) 1 (bfn_Off_snd t)))
```

While having succeeded in turning a lazy, circular into a strict, non-circular program, there is an unpleasant thing about the result: we see recomputation of the same intermediate results $zipPlus\ ds\ ds'$ in $bfn_{Off,snd}$ and bfn_{Off}. That is the price so far of replacing a single (though circular) traversal by two separate ones. Fortunately,

[3] Here is why we used a lazy pattern match above. Without it, we would have a bogus dependency of the second output of bfn_{Off} on its second input (namely requiring that the second input is not the empty list when the first input is a Fork). It would be possible to work around that by using that bfn_{Off} is never called with the empty list inside run and ensuing recursive calls. But working with a lazy pattern match right away is more convenient.

it is easy to avoid the recomputations by changing $bfn_{Off,snd}$ to store all the relevant intermediate results:

$$
\begin{array}{l}
\textbf{let } top\ t = \\
\quad \textbf{match } t \textbf{ with} \\
\quad\quad \mathsf{Empty} \qquad\quad \rightarrow [\,] \\
\quad\quad \mid \mathsf{Fork}\ (ds, \_, \_) \rightarrow ds \\
\\
\textbf{let rec } bfn_{Off,snd}\ t = \\
\quad \textbf{match } t \textbf{ with} \\
\quad\quad \mathsf{Empty} \qquad\quad \rightarrow \mathsf{Empty} \\
\quad\quad \mid \mathsf{Fork}\ (\_, l, r) \quad \rightarrow \\
\quad\quad\quad \textbf{let } tds\ = bfn_{Off,snd}\ l \textbf{ and} \\
\quad\quad\quad\quad\ tds' = bfn_{Off,snd}\ r \\
\quad\quad\quad \textbf{in } \mathsf{Fork}\ (1 :: (zipPlus\ (top\ tds)\ (top\ tds')), tds, tds')
\end{array}
$$

and then reusing them in bfn_{Off}. The latter means that the tree result of $bfn_{Off,snd}$ should be passed as an additional argument to bfn_{Off}. But since that tree has exactly the same shape as the input tree, and since bfn_{Off} already recurses over that input tree *while completely ignoring its content* (only using the tree's *shape*), it is actually possible to avoid introducing an extra argument, instead directly using the result of $bfn_{Off,snd}$ to drive the computation of bfn_{Off}:

$$
\begin{array}{l}
\textbf{let rec } bfn_{Off}\ t\ ks = \\
\quad \textbf{match } t \textbf{ with} \\
\quad\quad \mathsf{Empty} \qquad\quad \rightarrow (\mathsf{Empty}, [\,]) \\
\quad\quad \mid \mathsf{Fork}\ (ds'', l, r) \rightarrow \\
\quad\quad\quad \textbf{let } ks' = tl\ ks \textbf{ in} \\
\quad\quad\quad \textbf{let } (l', ds) = bfn_{Off}\ l\ ks' \textbf{ in} \\
\quad\quad\quad \textbf{let } (r', \_) = bfn_{Off}\ r\ (zipPlus\ ks'\ ds) \textbf{ in} \\
\quad\quad\quad (\mathsf{Fork}\ (hd\ ks, l', r'), ds'') \\
\\
\textbf{let } run\ t = \textbf{let } tds = bfn_{Off,snd}\ t \\
\qquad\qquad\quad \textbf{in } fst\ (bfn_{Off}\ tds\ (scanl\ (+)\ 1\ (top\ tds)))
\end{array}
$$

or, alternatively:

$$
\begin{array}{l}
\textbf{let rec } bfn_{Off}\ t\ ks = \\
\quad \textbf{match } t \textbf{ with} \\
\quad\quad \mathsf{Empty} \qquad\quad \rightarrow \mathsf{Empty} \\
\quad\quad \mid \mathsf{Fork}\ (\_, l, r) \rightarrow \\
\quad\quad\quad \textbf{let } ks' = tl\ ks \textbf{ in} \\
\quad\quad\quad \textbf{let } l'\ = bfn_{Off}\ l\ ks' \textbf{ and} \\
\quad\quad\quad\quad\ r' = bfn_{Off}\ r\ (zipPlus\ ks'\ (top\ l)) \textbf{ in} \\
\quad\quad\quad \mathsf{Fork}\ (hd\ ks, l', r') \\
\\
\textbf{let } run\ t = \textbf{let } tds = bfn_{Off,snd}\ t \\
\qquad\qquad\quad \textbf{in } bfn_{Off}\ tds\ (scanl\ (+)\ 1\ (top\ tds))
\end{array}
$$

Efficiency-wise, we have found that (the Haskell analogons of) these two alternatives just given are on a par. But an interesting difference between the two is that the second one, as opposed also to the original, circular program, has very good potential for parallel evaluation: in it, the two bfn_{Off}-calls are independent of each other. However, we have not explored this aspect further, yet.

A completely different alternative for avoiding $zipPlus$-recomputations, instead of introducing an intermediate data structure to store results, is to use the relationship

$$bfn\ t\ ks\ \equiv\ \textbf{let } (nt, ds) = bfn_{Off}\ t\ ks \textbf{ in } (nt, zipPlus\ ks\ ds)$$

with which we started the derivation in this subsection. Through it, we can rewrite the Haskell definition of run above the transliterated OCaml program (starting with **open** List) into:

$$
\begin{array}{l}
run :: \mathsf{Tree}\ a \rightarrow \mathsf{Tree\ Int} \\
run\ t = \textbf{let } nt = fst\ (bfn\ t\ (1 : ks)) \\
\qquad\qquad\ \ ks = tail\ (scanl\ (+)\ 1\ (bfn_{Off,snd}\ t)) \\
\qquad\quad \textbf{in } nt
\end{array}
$$

After all, we have by the above relationship that bfn and bfn_{Off} compute the same value in the first component of their output pair. The essence with this solution (which would equally well be possible in OCaml, of course) is that we have originally refactored bfn into bfn_{Off} to facilitate the removal of the circular dependency, but after we have done the specialization to/for $bfn_{Off,snd}$, we can, for the other traversal, switch back to the original function.

In either case (using bfn or bfn_{Off} for the second traversal in run, without or with employing an intermediate structure), we have now a two phase solution instead of the original circular definition. The first phase computes a list of "level beginnings", e.g., with

```
t = Fork 'a' (Fork 'b' Empty Empty)
     (Fork 'c' (Fork 'd' (Fork 'e' Empty Empty) Empty)
      (Fork 'f' Empty Empty))
```

we get:

$$scanl\ (+)\ 1\ (bfn_{Off,snd}\ t)\ \equiv\ [1, 2, 4, 6, 7]$$

The second phase uses such a list to do the actual numbering, either relying on $zipPlus$-calls (but with potential for independent, parallel processing of subtrees) or without (but with a necessarily more sequential processing). In the next subsection now, we derive an alternative for the *first* phase.

4.2 A Second Approach: Prefixes

Instead of using, as in the previous subsection, that the second output of the original bfn is obtained from its second input (ks) by element-wise adding a finite list (ds) to a finite prefix (of ks), we can also start from just the observation that exactly a finite prefix will be changed, without taking into account that this happens by repeatedly incrementing. So we now start again from the original bfn and first derive a variant which in its second output returns that finite prefix, rather than the whole second input with that prefix changed. The desired relationship between the two functions is:

$$bfn\ t\ ks\ \equiv\ \textbf{let } (nt, ps) = bfn_{Pre}\ t\ ks \textbf{ in } (nt, merge\ ks\ ps)$$

where

$$
\begin{array}{l}
merge :: [\mathsf{Int}] \rightarrow [\mathsf{Int}] \rightarrow [\mathsf{Int}] \\
merge\ [\,] \qquad ps \qquad\ = ps \\
merge\ ks \qquad [\,] \qquad\ = ks \\
merge\ (\_ : ks)\ (p : ps) = p : (merge\ ks\ ps)
\end{array}
$$

The desired function is obtained pretty straightforwardly as follows:

$$
\begin{array}{l}
bfn_{Pre} :: \mathsf{Tree}\ a \rightarrow [\mathsf{Int}] \rightarrow (\mathsf{Tree\ Int}, [\mathsf{Int}]) \\
bfn_{Pre}\ \mathsf{Empty} \qquad ks \qquad\ = (\mathsf{Empty}, [\,]) \\
bfn_{Pre}\ (\mathsf{Fork}\ \_\ l\ r)\ \tilde{}(k : ks) = (\mathsf{Fork}\ k\ l'\ r', \\
\qquad\qquad\qquad\qquad\qquad\qquad (k + 1) : (merge\ ps\ ps')) \\
\quad \textbf{where } (l', ps)\ = bfn_{Pre}\ l\ ks \\
\qquad\qquad (r', ps') = bfn_{Pre}\ r\ (merge\ ks\ ps)
\end{array}
$$

and can be used inside run as follows:

$$
\begin{array}{l}
run :: \mathsf{Tree}\ a \rightarrow \mathsf{Tree\ Int} \\
run\ t = \textbf{let } (nt, ps) = bfn_{Pre}\ t\ (1 : ks) \\
\qquad\qquad\ \ ks \qquad\ = merge\ (1 : ks)\ ps \\
\qquad\quad \textbf{in } nt
\end{array}
$$

Similar expansion and calculation as for the equation $ks\ =\ zipPlus\ \ (1 : ks)\ \ ds$ in Section 4.1 establishes that the equation $ks = merge\ (1 : ks)\ ps$ means $ks = ps +\!\!+ (repeat\ (last\ (1 : ps)))$.

Moreover, since bfn_{Pre} never consumes more elements from its second argument than it produces in its second output, we know that no element of list ks beyond those from ps will ever be needed, so we can directly write:

$$\begin{aligned}&run :: \mathsf{Tree}\ a \to \mathsf{Tree\ Int}\\&run\ t = \mathbf{let}\ (nt, ps) = bfn_{Pre}\ t\ (1 : ks)\\&\qquad\quad\ \ ks\ \ \ = ps\\&\quad\ \mathbf{in}\ nt\end{aligned}$$

Inlining, and splitting the call to bfn_{Pre} as follows:

$$\begin{aligned}&run :: \mathsf{Tree}\ a \to \mathsf{Tree\ Int}\\&run\ t = \mathbf{let}\ (nt, \_) = bfn_{Pre}\ t\ (1 : ps)\\&\qquad\quad\ (\_, ps) = bfn_{Pre}\ t\ (1 : ps)\\&\quad\ \mathbf{in}\ nt\end{aligned}$$

leaves us with a circular dependency of ps on itself. Applying the "type-based analysis plus specialization" approach from Section 2 leads to:

$$\begin{aligned}&bfn_{Pre,snd} :: \mathsf{Tree}\ a \to [\mathsf{Int}] \to [\mathsf{Int}]\\&bfn_{Pre,snd}\ \mathsf{Empty}\ \ \ \ ks\ \ \ \ = [\,]\\&bfn_{Pre,snd}\ (\mathsf{Fork}\ \_\ l\ r)\ \tilde{}(k : ks) = (k+1) : (merge\ ps\ ps')\\&\quad\mathbf{where}\ ps\ = bfn_{Pre,snd}\ l\ ks\\&\qquad\ \ ps' = bfn_{Pre,snd}\ r\ (merge\ ks\ ps)\end{aligned}$$

$$\begin{aligned}&run :: \mathsf{Tree}\ a \to \mathsf{Tree\ Int}\\&run\ t = \mathbf{let}\ nt = fst\ (bfn_{Pre}\ t\ (1 : ps))\\&\qquad\quad\ ps = bfn_{Pre,snd}\ t\ (1 : ps)\\&\quad\ \mathbf{in}\ nt\end{aligned}$$

but *fails* to discover any limits on input-output-dependencies. In particular, the circular dependency of ps on itself persists. Our best bet now is to again expand the list ps and try to discover internal relationships between list elements from

$$[p_0, p_1, \ldots, p_n] \ \equiv\ bfn_{Pre,snd}\ t\ [1, p_0, p_1, \ldots, p_n]$$

However, this clearly depends dynamically on the concrete tree t (in a certain way which we do not want to simply take for granted, though). So what can we do?

Well, conceptually at least we can still apply our approach of splitting a circular equation into several ones in the hope of discovering limited dependencies. The equation $ps = bfn_{Pre,snd}\ t\ (1{:}ps)$ thus becomes:

$$\begin{aligned}[p_0, \_, \ldots, \_] &= bfn_{Pre,snd}\ t\ [1, p_0, p_1, \ldots, p_n]\\[\_, p_1, \ldots, \_] &= bfn_{Pre,snd}\ t\ [1, p_0, p_1, \ldots, p_n]\\&\ldots\\[\_, \_, \ldots, p_n] &= bfn_{Pre,snd}\ t\ [1, p_0, p_1, \ldots, p_n]\end{aligned}$$

By inspection, in particular observing the behavior of $merge$, we find that the ith position of the output list of $bfn_{Pre,snd}$ only ever depends on the ith position of its second argument. Hence, the above becomes:

$$\begin{aligned}[p_0, \_, \ldots, \_] &= bfn_{Pre,snd}\ t\ [1, \bot, \bot, \ldots, \bot]\\[\_, p_1, \ldots, \_] &= bfn_{Pre,snd}\ t\ [\bot, p_0, \bot, \ldots, \bot]\\&\ldots\\[\_, \_, \ldots, p_n] &= bfn_{Pre,snd}\ t\ [\bot, \ldots, p_{n-1}, \bot]\end{aligned}$$

Note that this is of course not something we could write in the program, because even the length n of the target list can and will vary dynamically with t. But assume we had a function h which given an i and p_{i-1} (or 1 if $i = 0$) gives us the value bound to p_i in the relevant (if even existing) line above. More specifically, we seek a function

$$h :: \mathsf{Tree}\ a \to (\mathsf{Int}, \mathsf{Int}) \to \mathsf{Maybe\ Int}$$

such that $h\ t\ (i, p_{i-1})$ is Nothing if $bfn_{Pre,snd}\ t\ (1 : ps)$ contains no p_i, otherwise is Just p_i. Then we could rewrite run as follows:

$$\begin{aligned}&run :: \mathsf{Tree}\ a \to \mathsf{Tree\ Int}\\&run\ t = \mathbf{let}\ go\ (i, p) = \mathbf{case}\ h\ t\ (i, p)\ \mathbf{of}\\&\qquad\qquad\qquad\qquad\ \ \mathsf{Nothing} \to [\,]\\&\qquad\qquad\qquad\qquad\ \ \mathsf{Just}\ p' \ \ \to p' : (go\ (i+1, p'))\\&\qquad\quad ps = go\ (0, 1)\\&\quad\ \mathbf{in}\ fst\ (bfn_{Pre}\ t\ (1 : ps))\end{aligned}$$

The desired h-function can be derived from $bfn_{Pre,snd}$ by using that:

1. Instead of an input list we only need to pass in the single value that would have resided in the ith position (starting counting from zero). That is, an input (i, p) to h corresponds to an input list to $bfn_{Pre,snd}$ consisting of i occurrences of $\bot$, then p, then filled up with further $\bot$s.
2. Instead of an output list we only need to return information about whether an ith position exists in it, and if so, the value of that list element.
3. Lookup in lists interacts in a very simple way with the $merge$-function. Namely, an ith position exists in $merge\ xs\ ys$ if and only if it is so in at least one of xs and ys; and moreover, if both xs and ys contain an ith position, then the value from ys takes precedence.

The resulting function looks as follows:

$$\begin{aligned}&h :: \mathsf{Tree}\ a \to (\mathsf{Int}, \mathsf{Int}) \to \mathsf{Maybe\ Int}\\&h\ \mathsf{Empty}\ \ \ \ \ (\_, \_) = \mathsf{Nothing}\\&h\ (\mathsf{Fork}\ \_\ l\ r)\ (0, k) = \mathsf{Just}\ (k+1)\\&h\ (\mathsf{Fork}\ \_\ l\ r)\ (i, k) = \mathbf{case}\ h\ l\ (i-1, k)\ \mathbf{of}\\&\qquad\qquad\qquad\qquad\ \ \mathsf{Nothing} \to \mathbf{case}\ h\ r\ (i-1, k)\ \mathbf{of}\\&\qquad\qquad\qquad\qquad\qquad\qquad\ \ \mathsf{Nothing} \to \mathsf{Nothing}\\&\qquad\qquad\qquad\qquad\qquad\qquad\ \ \mathsf{Just}\ p' \ \ \to \mathsf{Just}\ p'\\&\qquad\qquad\qquad\qquad\ \ \mathsf{Just}\ p \ \ \ \to \mathbf{case}\ h\ r\ (i-1, p)\ \mathbf{of}\\&\qquad\qquad\qquad\qquad\qquad\qquad\ \ \mathsf{Nothing} \to \mathsf{Just}\ p\\&\qquad\qquad\qquad\qquad\qquad\qquad\ \ \mathsf{Just}\ p' \ \ \to \mathsf{Just}\ p'\end{aligned}$$

Its first equation corresponds to the first equation of $bfn_{Pre,snd}$. Its second equation corresponds to the second equation of $bfn_{Pre,snd}$ in the case that we are focussed on the 0th position in input and output. Finally, its third equation corresponds to the second equation of $bfn_{Pre,snd}$ in the case that we are focussed on a later position, i.e., the k in (i, k) corresponds to some element of the tail ks in $bfn_{Pre,snd}$'s equation, and correspondingly the output, if any, is to come from the call $merge\ ps\ ps'$ with $ps \ \equiv\ bfn_{Pre,snd}\ l\ ks$ and $ps' \ \equiv\ bfn_{Pre,snd}\ r\ (merge\ ks\ ps)$. Then, of the four branches of the nested **case**-expressions in the definition of h,

- the first corresponds to the case where neither ps nor ps' (which is actually equivalent to $bfn_{Pre,snd}\ r\ ks$ in this case as far as the $(i-1)$st position is concerned) contains an $(i-1)$st position;
- the second corresponds to the case where ps does not contain an $(i-1)$st position, but ps' (essentially equivalent to $bfn_{Pre,snd}\ r\ ks$ as before) does;
- the third corresponds to the case where ps does contain an $(i-1)$st position, with value p, but ps' (now equivalent to $bfn_{Pre,snd}\ r\ ps$ as far as the $(i-1)$st position is concerned) does not; and
- the fourth corresponds to the case that both ps and ps' (again essentially equivalent to $bfn_{Pre,snd}\ r\ ps$) contain values in the $(i-1)$st position, of which the one from the call on r takes precedence due to the call $merge\ ps\ ps'$.

Thus, we have arrived at a non-circular program suitable for use in OCaml. However, this time we right away replace, via the relationship

$$bfn\ t\ ks\ \equiv\ \textbf{let}\ (nt, ps) = bfn_{Pre}\ t\ ks\ \textbf{in}\ (nt, merge\ ks\ ps)$$

from the beginning of this subsection, bfn_{Pre} by the original bfn (for computing the first component of the output pair):

```
let rec bfn t ks =
  match t with
    Empty          → (Empty, ks)
  | Fork (_, l, r) →
      let (k, ks') = (List.hd ks, List.tl ks) in
      let (l', ks'') = bfn l ks' in
      let (r', ks''') = bfn r ks'' in
      (Fork (k, l', r'), (k + 1) :: ks''')

let rec h t ip =
  match t, ip with
    Empty, _              → None
  | Fork (_, l, r), (0, k) → Some (k + 1)
  | Fork (_, l, r), (i, k) → match h l (i − 1, k) with
                               None    → h r (i − 1, k)
                             | Some p →
                                 (match h r (i − 1, p) with
                                    None    → Some p
                                  | Some p' → Some p')

let run t = let rec go (i, p) =
                match h t (i, p) with
                  None    → []
                | Some p' → p' :: (go (i + 1, p')) in
            let ps = go (0, 1) in fst (bfn t (1 :: ps))
```

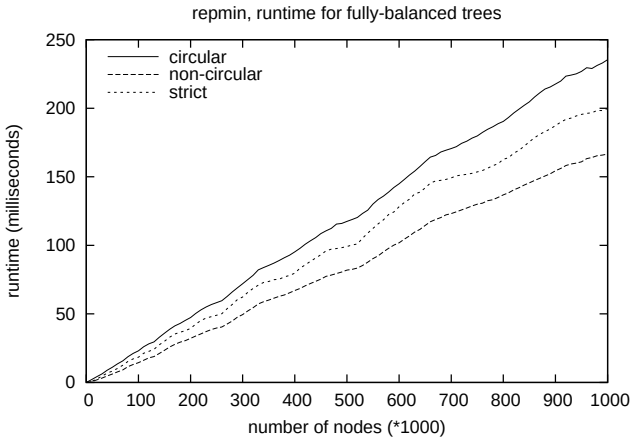
Essentially, we have arrived at an implementation of a breadth-first task via iterative deepening! Note that the Haskell version of it also works well on infinite trees, just as the Haskell versions from Section 4.1 do.

5. Analysis

Being able to transform circular into non-circular programs is certainly nice on a conceptual level, but ultimately of course the question is what such a transformation does to program efficiency. The two major factors of interest are runtime and heap consumption. We have performed a whole range of experiments in Haskell and OCaml. In order to really compare the impact of strictification (rather than the power of different compilers), we decided to plot here the results of systematic measurements in Haskell only. To simulate strict evaluation, rather than only evaluating a non-circular program in a lazy fashion, we employed Haskell's strict evaluation primitives (*seq* and friends). So generally we compare three program versions: a circular one, a non-circular one derived from it (and evaluated lazily), and an explicitly strictified version of the latter (with strictness primitives judiciously added where Haskell would otherwise deviate from OCaml's evaluation order). Measurements were performed on a Dell Precision Workstation T3400 with an Intel® Core™2 Q9550 processor (4 x 2.83GHz) and 3.8GB memory available. Programs were compiled with ghc-6.12.3, optimizing with `-O2`. The Criterion library (`http://hackage.haskell.org/package/criterion`) was used for runtime measurements and GHC's built-in profiler for heap measurements (for the latter, with stack size increased to 500 MBytes via the runtime option `-K500M`). Where appropriate and interesting, we also comment on the relative efficiency of OCaml vs. Haskell, as observed via wall-clock measurements. (For OCaml we used native-code compilation via `ocamlopt` version 3.11.2, stack size set with `ulimit -s 500000`.) We analyze the programs from Sections 2 and 4, and variants/algorithms that have come up in the literature [Chin et al. 1999; Okasaki 2000; Pettorossi and Skowron 1987]. We do not show results for the programs from Section 3, even though we have measured them as well. Those measurements show that the non-circular versions are on a par with, or better than (sometimes considerably, depending on the distribution of variable declarations and uses in the input sequence), the circular version.

5.1 Repmin

First the relation between circular and non-circular variants of the simple *repmin*-example is measured using three program versions: the circular program we started from in Section 2, the non-circular program we ended up with in Section 2, and a completely strict version of the latter. It turns out that the circular version is slowest, despite the fact that it ostensibly saves traversal work compared to the two non-circular versions. The relative efficiency of the two non-circular versions depends on the shape of trees. On fully balanced trees we find that the lazily evaluated version is better (also than the OCaml version, which has about the same performance as the strict Haskell version):



while for strongly left-leaning trees we observed that the completely strict version had a small advantage.

An interesting point of comparison is the results of Chin et al. [1999]. They consider *strictness-guided tupling* to prevent the introduction of extra thunks, and also look at circular tupling. In particular, they perform measurements for various versions of *repmin* (which they call *mintip*) in their Section 6. It turns out that, possibly due to changes/advances in compiler technology for lazy languages, our findings today differ from what they observed. In a nutshell, their results are summarized in Table 3 on page 127, replicated here in part:[4]

100 times of *mintip* on a tree of depth 12					
	Heap	Time(s)			
	(bytes)	INIT	MUT	GC	Total
No Tupling (!)	29,679,824	0.02	19.49	0.57	20.08
Tupling with (!)	52,620,228	0.01	27.81	4.14	31.96
No Tupling ($\star$)	24,762,928	0.03	14.51	0.57	15.11
Tupling with ($\star$)	18,211,728	0.02	11.95	0.34	12.31

[4] Of the six lines of measurements there, we consider only the first two and the last two, because the middle two concern a "medium-strict" version of *repmin* that we do not have otherwise present in our repertoire.

The first line here corresponds to our non-circular version. The second line corresponds to our circular version. The third line corresponds to our completely strictified non-circular version. And finally, the fourth line is the outcome of Chin et al.'s strictness-guided, circular tupling, i.e., a circular program with extra strictness annotations to prevent the detrimental effects of tupling on efficiency. In contrast to Chin et al.'s measurements, our plot above situates "No Tupling ($\star$)" between "No Tupling (!)" and "Tupling with (!)", and if we include "Tupling with ($\star$)" in the picture, we find that it performs almost identically to "No Tupling ($\star$)" (actually, slightly worse).

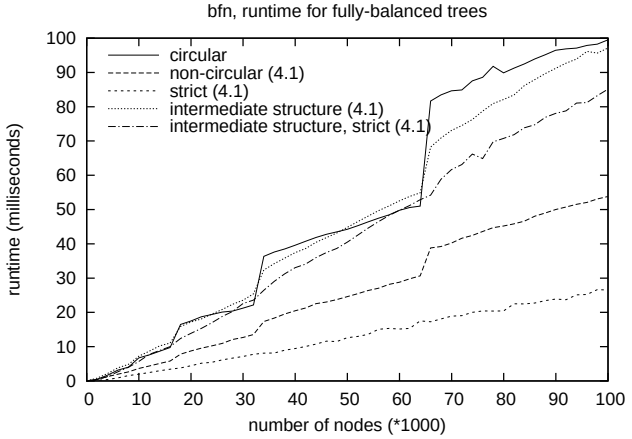
We have also run lazily and strictly evaluated versions of $repmin$ that Pettorossi and Skowron [1987] obtained by applying the lambda-abstraction strategy [Pettorossi and Proietti 1988]. We found them to perform worse than all the program versions considered above. In OCaml, the multi-traversal and the single-traversal, higher-order program had almost indistinguishable performance, and were not considerably faster than the original circular version in Haskell. As in the case of the results of Chin et al. [1999], we do not really have a ready explanation for these differences we observed from what the literature suggests about the relative performance to be expected when comparing different flavors of circular and non-circular programs.

5.2 Breadth-First Numbering

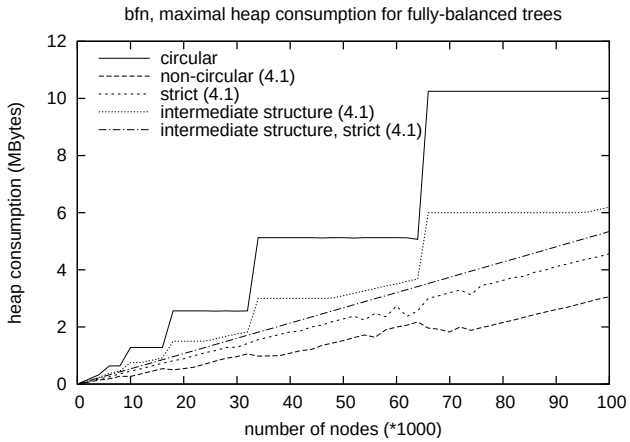
Here, let us begin by studying the programs from Section 4.1. We measure five program versions:

- the circular program we started from in Section 4;
- the non-circular Haskell program we finally ended up with in Section 4.1, with bfn_{Off} replaced by the original bfn, and with the main call in run changed to fit with the OCaml versions;
- a completely strict version of the latter;
- a Haskell version of the last OCaml program in Section 4.1, using an intermediate structure; and
- a completely strict version of the latter.

It turns out that the versions using an intermediate structure are not really a consistent/substantial runtime improvement over the original, circular program (but recall that we identified parallelization potential for these versions, which might ultimately change the picture), while the versions doing without an intermediate structure (but coming without potential for parallelization) do quite well in sequential evaluation:

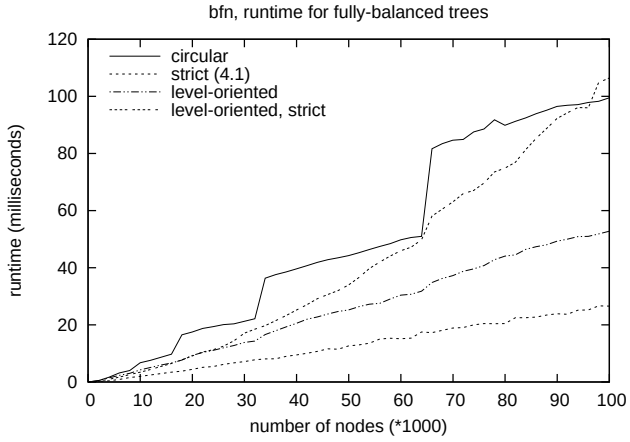


An interesting further observation is that, in contrast to what we saw in Section 5.1, use of Haskell's strictness primitives pays off here, as both the strict programs perform faster than their corresponding non-strict versions.[5] Moreover, if we look at Haskell heap consumption even the program versions using an intermediate structure are better than the original, circular program:



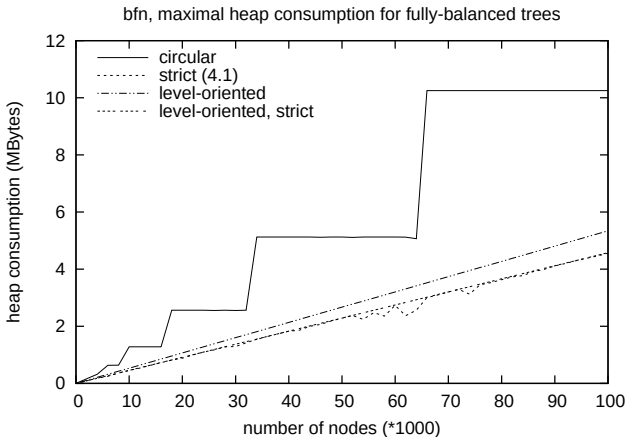
We also measured the final non-circular Haskell program derived in Section 4.2, with bfn_{Pre} replaced by the original bfn (and with h changed slightly to fit with the OCaml version), and a completely strict version of it. We found that the strictified version is about equally good as the completely strict version arising from Section 4.1 (which was the best one above), and that the same relative statement holds for wall-clock measurements in OCaml. In terms of Haskell heap consumption, we found that the lazily evaluated non-circular program arising from Section 4.2 performs almost exactly like the corresponding one from Section 4.1, and similarly for the completely strict Haskell versions.

We have already mentioned that Okasaki [2000] studied breadth-first numbering from an algorithmic perspective. As non-circular programs, he presents two different algorithms, one level-oriented (his Figure 5), the other forest/queue-based (his Figure 3). If we run a Haskell implementation of the level-oriented solution and a completely strict version of it against the circular breadth-first numbering program and against our own best version, we get the following timings (showing that our own program still performs the best; the same holds in OCaml):

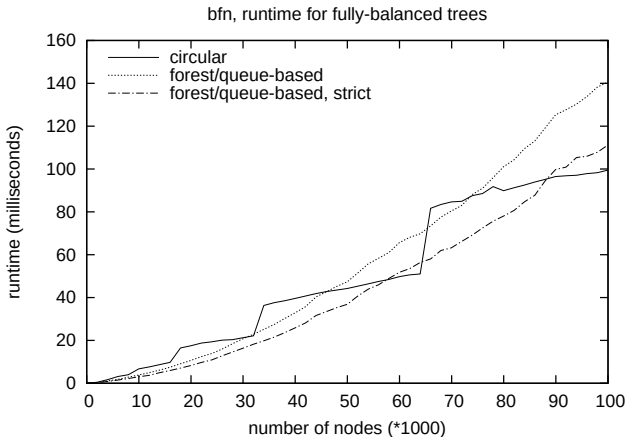


[5] And if we move to OCaml, the runtimes of the strict versions are cut by about another half.

For Haskell heap consumption, the situation is similar:



Measuring a Haskell implementation of Okasaki's forest/queue-based solution and a completely strict version of it, we found that both perform similarly to the original, circular program in terms of runtime:



while the heap consumption lines are similar to those for the level-oriented programs above. Surprisingly, we observed the forest/queue-based solution in OCaml to take about 50% more time than the corresponding strict Haskell version. (We have not tried to fine-tune Okasaki's OCaml version to potentially invert the situation, which might well be possible.)

6. Conclusion

We have proposed an approach to eliminating circular definitions from traversal programs in a lazy functional language, and performed benchmarking that shows it effective in practice. One further potential use of this kind of transformation is as a preprocessing step for other optimization techniques. For example, elimination of intermediate results (deforestation) from compositions of circular programs (with other, circular or non-circular) programs is a challenging problem. By first eliminating circularity, we could reduce this problem to one in a more standard setting. Using techniques like those of Voigtländer [2004] and Fernandes et al. [2007] we could even end up with circular programs again in the end. Similarly, we could try to benefit from the technique of Chin et al. [1999] for the optimization of circular programs. As developed, that technique applies to a non-circular program as a starting point. In fact, the authors emphasize that the same effects cannot be obtained by directly applying strictness analysis to a circular program. But using our approach, a possible route for optimization of circular programs would be to first transform into a non-circular program and then use Chin et al.'s technique.

Acknowledgments

We thank the anonymous reviewers for their comments and suggestions, and Chris Okasaki for remarks on *bfn* and our variants.

References

R.S. Bird. Using circular programs to eliminate multiple traversals of data. *Acta Informatica*, 21(3):239–250, 1984.

R.M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, 1977.

W.N. Chin, A.H. Goh, and S.C. Khoo. Effective optimisation of multiple traversals in lazy languages. In *Partial Evaluation and Semantics-Based Program Manipulation, Proceedings*, Technical Report, University of Aarhus, pages 119–130, 1999.

O. de Moor, S.L. Peyton Jones, and E. Van Wyk. Aspect-oriented compilers. In *Generative and Component-Based Software Engineering 1999, Revised Papers*, LNCS 1799:121–133. Springer, 2000.

J.P. Fernandes and J. Saraiva. Tools and libraries to model and manipulate circular programs. In *Partial Evaluation and Semantics-Based Program Manipulation, Proceedings*, pages 102–111. ACM, 2007.

J.P. Fernandes, A. Pardo, and J. Saraiva. A shortcut fusion rule for circular program calculation. In *Haskell Workshop, Proceedings*, pages 95–106. ACM, 2007.

T. Johnsson. Attribute grammars as a functional programming paradigm. In *Functional Programming Languages and Computer Architecture, Proceedings*, LNCS 274:154–173. Springer, 1987.

G. Jones and J. Gibbons. Linear-time breadth-first tree algorithms: An exercise in the arithmetic of folds and zips. Technical Report 71, Department of Computer Science, University of Auckland, 1993.

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

U. Kastens, A.M. Sloane, and W.M. Waite. *Generating Software from Specifications*. Jones & Bartlett Publishers, 2007.

N. Kobayashi. Type-based useless-variable elimination. *Higher-Order and Symbolic Computation*, 14(2–3):221–260, 2001.

M.F. Kuiper and S.D. Swierstra. Using attribute grammars to derive efficient functional programs. In *Computing Science in the Netherlands, Proceedings*, pages 39–52. SION, 1987.

C. Okasaki. Breadth-first numbering: lessons from a small exercise in algorithm design. In *International Conference on Functional Programming, Proceedings*, pages 131–136. ACM, 2000.

A. Pardo, J.P. Fernandes, and J. Saraiva. Shortcut fusion rules for the derivation of circular and higher-order monadic programs. In *Partial Evaluation and Program Manipulation, Proceedings*, pages 81–90. ACM, 2009.

A. Pettorossi and M. Proietti. Importing and exporting information in program development. In *Partial Evaluation and Mixed Computation 1987, Proceedings*, pages 405–425. North-Holland, 1988.

A. Pettorossi and M. Proietti. Rules and strategies for transforming functional and logic programs. *ACM Computing Surveys*, 28(2):360–414, 1996.

A. Pettorossi and A. Skowron. Higher order generalization in program derivation. In *Theory and Practice of Software Development, Proceedings*, LNCS 250:182–196. Springer, 1987.

J. Saraiva. *Purely Functional Implementation of Attribute Grammars*. PhD thesis, Utrecht University, Department of Computer Science, 1999.

J. Voigtländer. Using circular programs to deforest in accumulating parameters. *Higher-Order and Symbolic Computation*, 17(1–2):129–163, 2004.