Strictification of Circular Programs

J.P. Fernandes[1] J. Saraiva[1]

D. Seidel[2] J. Voigtländer[2]

[1]University of Minho

[2]University of Bonn

IFIP WG 2.1 meeting #66

Multi-Traversal Programs

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

treemin :: Tree Int → Int
treemin (Leaf n)    = n
treemin (Fork l r)  = min (treemin l) (treemin r)

replace :: Tree Int → Int → Tree Int
replace (Leaf n)   m = Leaf m
replace (Fork l r) m = Fork (replace l m)
                            (replace r m)

run :: Tree Int → Tree Int
run t = replace t (treemin t)
```

Circular Programs [Bird 1984]

The previous can be transformed into:

```
repmin :: Tree Int → Int → (Tree Int, Int)
repmin (Leaf n)   m = (Leaf m, n)
repmin (Fork l r) m = (Fork l′ r′, min m1 m2)
  where (l′, m1) = repmin l m
        (r′, m2) = repmin r m

run :: Tree Int → Tree Int
run t = let (nt, m) = repmin t m in nt
```

Only one traversal!

Circular Programs

Other uses/appearances of circular programs:

- as attribute grammar realization
  [Johnsson 1987, Kuiper & Swierstra 1987]

Circular Programs

Other uses/appearances of circular programs:

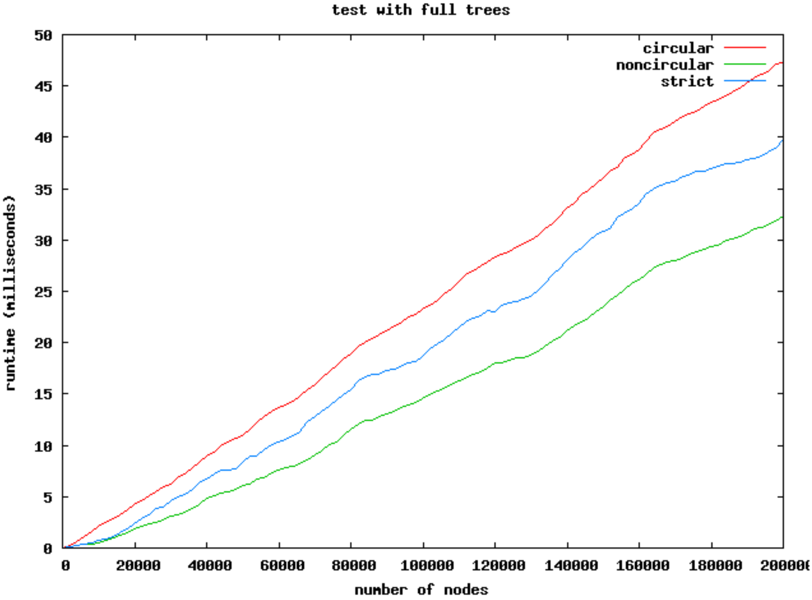
- as attribute grammar realization
  [Johnsson 1987, Kuiper & Swierstra 1987]
- as algorithmic tool
  [Jones & Gibbons 1993, Okasaki 2000]

Circular Programs

Other uses/appearances of circular programs:

- as attribute grammar realization
  [Johnsson 1987, Kuiper & Swierstra 1987]
- as algorithmic tool
  [Jones & Gibbons 1993, Okasaki 2000]
- as target for deforestation/fusion techniques
  [V. 2004, Fernandes et al. 2007]
- . . .

But:


test with full trees
circular
noncircular
strict
runtime (milliseconds)
50
45
40
35
30
25
20
15
10
5
0
0
20000
40000
60000
80000
100000
120000
140000
160000
180000
20000
number of nodes

The Aim

```
repmin (Leaf n)   m = (Leaf m, n)
repmin (Fork l r) m = (Fork l' r', min m1 m2)
   where (l', m1) = repmin l m
         (r', m2) = repmin r m
```

```
run t = let (nt, m) = repmin t m in nt
```

$$\Big\Downarrow \; ?$$

```
treemin (Leaf n)    = n
treemin (Fork l r)  = min (treemin l) (treemin r)
```

```
replace (Leaf n)   m = Leaf m
replace (Fork l r) m = Fork (replace l m) (replace r m)
```

```
run t = replace t (treemin t)
```

Getting Started

Let us have a look at:

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

and try to learn something about it.

Getting Started

Let us have a look at:

$$
\begin{array}{l}
\texttt{repmin} :: \mathsf{Tree\ Int} \rightarrow \mathsf{Int} \rightarrow (\mathsf{Tree\ Int}, \mathsf{Int}) \\
\texttt{repmin}\ (\mathsf{Leaf}\ n)\quad m = (\mathsf{Leaf}\ m, n) \\
\texttt{repmin}\ (\mathsf{Fork}\ l\ r)\ m = (\mathsf{Fork}\ l'\ r', \texttt{min}\ m_1\ m_2) \\
\quad \textbf{where}\ (l', m_1)\ = \texttt{repmin}\ l\ m \\
\qquad\qquad\quad (r', m_2) = \texttt{repmin}\ r\ m
\end{array}
$$

and try to learn something about it.

What better way to learn something about a function than looking at its inferred type?

Dependency Analysis for Free

It turns out that (from the given equations):

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

Dependency Analysis for Free

It turns out that (from the given equations):

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

Very interesting: the second output cannot possibly depend on the second input!

Dependency Analysis for Free

It turns out that (from the given equations):

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

Very interesting: the second output cannot possibly depend on the second input!

Hence, for every $t :: \mathsf{Tree\ Int}$ and m_1, m_2:

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

Dependency Analysis for Free

It turns out that (from the given equations):

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

Very interesting: the second output cannot possibly depend on the second input!

Hence, for every t :: Tree Int and m_1, m_2:

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

Indeed, for every t :: Tree Int and m:

$$\texttt{snd}\ (\texttt{repmin}\ t\ m) \equiv \texttt{snd}\ (\texttt{repmin}\ t\ \bot)$$

Achieving Noncircularity

$$\texttt{run}\ t = \textbf{let}\ (nt, m) = \texttt{repmin}\ t\ m\ \textbf{in}\ nt$$

Achieving Noncircularity

$$\texttt{run}\ t = \textbf{let}\ (nt, m) = \texttt{repmin}\ t\ m\ \textbf{in}\ nt$$

$$\Downarrow\ \text{by referential transparency}$$

$$\begin{aligned}\texttt{run}\ t = \textbf{let}\ & (nt, \_) = \texttt{repmin}\ t\ m \\ & (\_, m) = \texttt{repmin}\ t\ m \\ \textbf{in}\ & nt\end{aligned}$$

Achieving Noncircularity

$$\texttt{run}\ t = \mathbf{let}\ (nt, m) = \texttt{repmin}\ t\ m\ \mathbf{in}\ nt$$

$$\Downarrow \quad \text{by referential transparency}$$

$$\begin{array}{l}\texttt{run}\ t = \mathbf{let}\ (nt, \_) = \texttt{repmin}\ t\ m \\ \qquad\qquad\quad\ \ (\_, m) = \texttt{repmin}\ t\ m \\ \qquad\quad\ \ \mathbf{in}\ nt\end{array}$$

$$\Downarrow \quad \text{by } \texttt{snd}\ (\texttt{repmin}\ t\ m)\ \equiv\ \texttt{snd}\ (\texttt{repmin}\ t\ \bot)$$

$$\begin{array}{l}\texttt{run}\ t = \mathbf{let}\ (nt, \_) = \texttt{repmin}\ t\ m \\ \qquad\qquad\quad\ \ (\_, m) = \texttt{repmin}\ t\ \bot \\ \qquad\quad\ \ \mathbf{in}\ nt\end{array}$$

Towards Efficiency

Instead of having:

$$(\_, m) = \texttt{repmin}\ t\ \bot$$

Towards Efficiency

Instead of having:

$$(\_, m) = \texttt{repmin}\ t\ \bot$$

let us define a specialized function:

$$\texttt{repmin}_{\texttt{snd}} :: \text{Tree Int} \to \text{Int}$$
$$\texttt{repmin}_{\texttt{snd}}\ t = \texttt{snd}\ (\texttt{repmin}\ t\ \bot)$$

Towards Efficiency

Instead of having:

$$(\_, m) = \texttt{repmin}\ t\ \bot$$

let us define a specialized function:

$\texttt{repmin}_{\texttt{snd}} :: \mathsf{Tree\ Int} \rightarrow \mathsf{Int}$
$\texttt{repmin}_{\texttt{snd}}\ t = \texttt{snd}\ (\texttt{repmin}\ t\ \bot)$

which then lets us replace the above binding with:

$$m = \texttt{repmin}_{\texttt{snd}}\ t$$

Towards Efficiency

Instead of having:

$$(\_, m) = \texttt{repmin}\ t\ \bot$$

let us define a specialized function:

$\texttt{repmin}_{\texttt{snd}} :: \mathsf{Tree\ Int} \to \mathsf{Int}$
$\texttt{repmin}_{\texttt{snd}}\ t = \texttt{snd}\ (\texttt{repmin}\ t\ \bot)$

which then lets us replace the above binding with:

$$m = \texttt{repmin}_{\texttt{snd}}\ t$$

Using fold/unfold-transformations, it is easy to derive a direct definition for $\texttt{repmin}_{\texttt{snd}}$!

Towards Efficiency

Resulting definition:

$$
\begin{aligned}
&\mathtt{repmin}_{\mathtt{snd}} :: \mathsf{Tree\ Int} \rightarrow \mathsf{Int}\\
&\mathtt{repmin}_{\mathtt{snd}}\ (\mathsf{Leaf}\ n) \quad = n\\
&\mathtt{repmin}_{\mathtt{snd}}\ (\mathsf{Fork}\ l\ r) = \mathtt{min}\ (\mathtt{repmin}_{\mathtt{snd}}\ l)\\
&\qquad\qquad\qquad\qquad\qquad\quad (\mathtt{repmin}_{\mathtt{snd}}\ r)
\end{aligned}
$$

Towards Efficiency

Resulting definition:

$$
\begin{aligned}
&\texttt{repmin}_{\texttt{snd}} :: \text{Tree Int} \rightarrow \text{Int} \\
&\texttt{repmin}_{\texttt{snd}}\ (\text{Leaf}\ n) \quad = n \\
&\texttt{repmin}_{\texttt{snd}}\ (\text{Fork}\ l\ r) = \texttt{min}\ (\texttt{repmin}_{\texttt{snd}}\ l) \\
&\qquad\qquad\qquad\qquad\qquad\quad (\texttt{repmin}_{\texttt{snd}}\ r)
\end{aligned}
$$

Similarly, for $(nt, \_) = \texttt{repmin}\ t\ m$:

$$
\begin{aligned}
&\texttt{repmin}_{\texttt{fst}} :: \text{Tree Int} \rightarrow b \rightarrow \text{Tree}\ b \\
&\texttt{repmin}_{\texttt{fst}}\ (\text{Leaf}\ n) \quad m = \text{Leaf}\ m \\
&\texttt{repmin}_{\texttt{fst}}\ (\text{Fork}\ l\ r)\ m = \text{Fork}\ (\texttt{repmin}_{\texttt{fst}}\ l\ m) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad (\texttt{repmin}_{\texttt{fst}}\ r\ m)
\end{aligned}
$$

Final Program

$$\begin{array}{l}
\texttt{run} :: \text{Tree Int} \to \text{Tree Int} \\
\texttt{run}\ t = \textbf{let}\ (nt, \_) = \texttt{repmin}\ t\ m \\
\qquad\qquad\quad\ (\_, m) = \texttt{repmin}\ t\ \bot \\
\qquad\quad\ \ \textbf{in}\ nt
\end{array}$$

$$\Bigg\Downarrow \quad \begin{array}{l}
\text{by } \texttt{fst}\ (\texttt{repmin}\ t\ m) \equiv \texttt{repmin}_{\texttt{fst}}\ t\ m, \\
\qquad \texttt{snd}\ (\texttt{repmin}\ t\ \bot) \equiv \texttt{repmin}_{\texttt{snd}}\ t
\end{array}$$

$$\begin{array}{l}
\texttt{run} :: \text{Tree Int} \to \text{Tree Int} \\
\texttt{run}\ t = \texttt{repmin}_{\texttt{fst}}\ t\ (\texttt{repmin}_{\texttt{snd}}\ t)
\end{array}$$

A General Strategy

1. Detect dependencies of outputs of a circular call on its inputs. Preferrably, do this light-weight. As far as possible, type-based [Kobayashi 2001].

A General Strategy

1. Detect dependencies of outputs of a circular call on its inputs. Preferrably, do this light-weight. As far as possible, type-based [Kobayashi 2001].

2. Naively split the circular call into several ones, each computing only one of the outputs. Exploit information from above to decouple these calls.

A General Strategy

1. Detect dependencies of outputs of a circular call on its inputs. Preferrably, do this light-weight. As far as possible, type-based [Kobayashi 2001].

2. Naively split the circular call into several ones, each computing only one of the outputs. Exploit information from above to decouple these calls.

3. Specialize the different calls (using partial evaluation, slicing, . . .) to work only with those pieces of input and output that are relevant.

A More Challenging Example: Breadth-First Numbering [Okasaki 2000]

```
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
```

Let us Try the General Strategy

```
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′
```

Inferred type of `bfn` is still

```
Tree a → [Int] → (Tree Int, [Int])
```

Let us Try the General Strategy

```
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′
```

Inferred type of `bfn` is still

$$\text{Tree } a \to [\text{Int}] \to (\text{Tree Int}, [\text{Int}])$$

Precise dependency of output list on input list too intricate for type system to figure out!

A Little Help

Note that second output of `bfn` always built from second input by (potentially repeatedly) incrementing list elements.

A Little Help

Note that second output of `bfn` always built from second input by (potentially repeatedly) incrementing list elements.

So let us derive a variant with:

$$\begin{aligned} \texttt{bfn}\ t\ ks\ &\equiv\ \textbf{let}\ (nt, ds) = \texttt{bfn}_{\texttt{Off}}\ t\ ks \\ &\ \ \textbf{in}\ (nt, \texttt{zipPlus}\ ks\ ds) \end{aligned}$$

where:

$$\begin{array}{llll} \texttt{zipPlus} :: [\mathsf{Int}] \to [\mathsf{Int}] \to [\mathsf{Int}] \\ \texttt{zipPlus}\ [\,] & ds & = ds \\ \texttt{zipPlus}\ ks & [\,] & = ks \\ \texttt{zipPlus}\ (k : ks) & (d : ds) & = (k + d) : (\texttt{zipPlus}\ ks\ ds) \end{array}$$

A Little Help

The pretty straightforward derivation result:

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

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

Applying our General Strategy

$$\begin{aligned}
\mathtt{run}\ t = \mathbf{let}\ (nt, ds) &= \mathtt{bfn}_{\mathtt{Off}}\ t\ (1 : ks) \\
ks &= \mathtt{zipPlus}\ (1 : ks)\ ds \\
\mathbf{in}\ nt &
\end{aligned}$$

$$\Big\Downarrow \quad \text{by splitting calls}$$

$$\begin{aligned}
\mathtt{run}\ t = \mathbf{let}\ (nt, \_) &= \mathtt{bfn}_{\mathtt{Off}}\ t\ (1 : ks) \\
(\_, ds) &= \mathtt{bfn}_{\mathtt{Off}}\ t\ (1 : ks) \\
ks &= \mathtt{zipPlus}\ (1 : ks)\ ds \\
\mathbf{in}\ nt &
\end{aligned}$$

Removing One of the Two Circularities

From

$$(\_, ds) = \mathtt{bfn}_{\mathtt{Off}}\ t\ (1 : ks)$$

to

$$(\_, ds) = \mathtt{bfn}_{\mathtt{Off}}\ t\ \bot$$

where:

$$\begin{aligned}
&\mathtt{bfn}_{\mathtt{Off}} :: \mathsf{Tree}\ a \to b \to (c, [\mathsf{Int}])\\
&\mathtt{bfn}_{\mathtt{Off}}\ \mathsf{Empty}\qquad ks \qquad\quad = (\bot, [\,])\\
&\mathtt{bfn}_{\mathtt{Off}}\ (\mathsf{Fork}\ \_\ l\ r)\ \tilde{}(k : ks) = (\bot,\\
&\qquad\qquad\qquad\qquad\qquad\qquad 1 : (\mathtt{zipPlus}\ ds\ ds'))\\
&\quad \mathbf{where}\ (l', ds)\ \ = \mathtt{bfn}_{\mathtt{Off}}\ l\ \bot\\
&\qquad\qquad\ (r', ds') = \mathtt{bfn}_{\mathtt{Off}}\ r\ \bot
\end{aligned}$$

Specializing . . .

. . . leads to:

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

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

Looking at $ks = \texttt{zipPlus}\ (1 : ks)\ ds$

$$
\begin{aligned}
& [\,k_0, k_1, \ldots\,] \\
\equiv\;& \texttt{zipPlus}\ [\,1, k_0, k_1, \ldots\,]\ [\,d_0, d_1, \ldots, d_n\,]
\end{aligned}
$$

Looking at $ks = \texttt{zipPlus}\ (1 : ks)\ ds$

$$
\begin{aligned}
& [k_0, k_1, \ldots] \\
\equiv\ & \texttt{zipPlus}\ [1, k_0, k_1, \ldots]\ [d_0, d_1, \ldots, d_n] \\
\equiv\ & (1 + d_0) : (\texttt{zipPlus}\ [k_0, k_1, \ldots]\ [d_1, \ldots, d_n])
\end{aligned}
$$

Looking at $ks = \texttt{zipPlus}\ (1 : ks)\ ds$

$$
\begin{aligned}
& [k_0, k_1, \ldots] \\
\equiv\ & \texttt{zipPlus}\ [1, k_0, k_1, \ldots]\ [d_0, d_1, \ldots, d_n] \\
\equiv\ & (1 + d_0) : (\texttt{zipPlus}\ [k_0, k_1, \ldots]\ [d_1, \ldots, d_n]) \\
\equiv\ & (1 + d_0) : ((1 + d_0) + d_1) : \\
& (\texttt{zipPlus}\ [k_1, \ldots]\ [d_2, \ldots, d_n])
\end{aligned}
$$

Looking at $ks = \texttt{zipPlus}\ (1 : ks)\ ds$

$$
\begin{aligned}
& [k_0, k_1, \ldots] \\
\equiv\ & \texttt{zipPlus}\ [1, k_0, k_1, \ldots]\ [d_0, d_1, \ldots, d_n] \\
\equiv\ & (1 + d_0) : (\texttt{zipPlus}\ [k_0, k_1, \ldots]\ [d_1, \ldots, d_n]) \\
\equiv\ & (1 + d_0) : ((1 + d_0) + d_1) : (\texttt{zipPlus} \ldots) \\
\equiv\ & (1 + d_0) : ((1 + d_0) + d_1) : (((1 + d_0) + d_1) + d_2) : \\
& (\texttt{zipPlus}\ [k_2, \ldots]\ [d_3, \ldots, d_n])
\end{aligned}
$$

Looking at $ks = \texttt{zipPlus}\ (1 : ks)\ ds$

$$
\begin{aligned}
& [k_0, k_1, \ldots] \\
\equiv\ & \texttt{zipPlus}\ [1, k_0, k_1, \ldots]\ [d_0, d_1, \ldots, d_n] \\
\equiv\ & (1 + d_0) : (\texttt{zipPlus}\ [k_0, k_1, \ldots]\ [d_1, \ldots, d_n]) \\
\equiv\ & (1 + d_0) : ((1 + d_0) + d_1) : (\texttt{zipPlus}\ \ldots) \\
\equiv\ & (1 + d_0) : ((1 + d_0) + d_1) : (((1 + d_0) + d_1) + d_2) : \\
& (\texttt{zipPlus}\ [k_2, \ldots]\ [d_3, \ldots, d_n]) \\
\equiv\ & \ldots
\end{aligned}
$$

Looking at $ks = \texttt{zipPlus}\ (1 : ks)\ ds$

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

Looking at $ks = \texttt{zipPlus}\ (1 : ks)\ ds$

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

Looking at $ks = \texttt{zipPlus}\ (1 : ks)\ ds$

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

(Almost) Finally:

$$\begin{array}{l} \texttt{run} :: \mathsf{Tree}\ a \rightarrow \mathsf{Tree\ Int} \\ \texttt{run}\ t = \mathbf{let}\ nt = \texttt{fst}\ (\texttt{bfn}_{\texttt{Off}}\ t\ (1 : ks)) \\ \qquad\qquad\quad ds = \texttt{bfn}_{\texttt{Off},\texttt{snd}}\ t \\ \qquad\qquad\quad ks = (\texttt{tail}\ (\texttt{scanl}\ (+)\ 1\ ds)) +\!\!\!+ \\ \qquad\qquad\qquad\quad (\texttt{repeat}\ (\texttt{last}\ (\texttt{scanl}\ (+)\ 1\ ds))) \\ \qquad\quad \mathbf{in}\ nt \end{array}$$

Can be directly transliterated to OCaml!

(Almost) Finally:

$$
\begin{aligned}
&\texttt{run} :: \mathsf{Tree}\ \mathit{a} \rightarrow \mathsf{Tree\ Int}\\
&\texttt{run}\ \mathit{t} = \textbf{let}\ \mathit{nt} = \texttt{fst}\ (\texttt{bfn}_{\texttt{Off}}\ \mathit{t}\ (1 : \mathit{ks}))\\
&\qquad\qquad\quad\ \mathit{ds} = \texttt{bfn}_{\texttt{Off},\texttt{snd}}\ \mathit{t}\\
&\qquad\qquad\quad\ \mathit{ks} = (\texttt{tail}\ (\texttt{scanl}\ (+)\ 1\ \mathit{ds})) +\!\!+\\
&\qquad\qquad\qquad\qquad (\texttt{repeat}\ (\texttt{last}\ (\texttt{scanl}\ (+)\ 1\ \mathit{ds})))\\
&\qquad\quad\ \textbf{in}\ \mathit{nt}
\end{aligned}
$$

Can be directly transliterated to OCaml!

And/or optimized a bit:

$$
\begin{aligned}
&\texttt{run} :: \mathsf{Tree}\ \mathit{a} \rightarrow \mathsf{Tree\ Int}\\
&\texttt{run}\ \mathit{t} = \textbf{let}\ \mathit{ds} = \texttt{bfn}_{\texttt{Off},\texttt{snd}}\ \mathit{t}\\
&\qquad\quad\ \textbf{in}\ \texttt{fst}\ (\texttt{bfn}_{\texttt{Off}}\ \mathit{t}\ (\texttt{scanl}\ (+)\ 1\ \mathit{ds}))
\end{aligned}
$$

Inefficiency Lurking

$$
\begin{aligned}
&\mathtt{bfn}_{\mathtt{Off,snd}}\ \mathsf{Empty} && = [\,] \\
&\mathtt{bfn}_{\mathtt{Off,snd}}\ (\mathsf{Fork}\ \_\ l\ r) && = 1 : (\mathtt{zipPlus}\ (\mathtt{bfn}_{\mathtt{Off,snd}}\ l) \\
& && \qquad\qquad\qquad\ (\mathtt{bfn}_{\mathtt{Off,snd}}\ r))
\end{aligned}
$$

$$
\begin{aligned}
&\mathtt{bfn}_{\mathtt{Off}}\ \mathsf{Empty} && ks && = (\mathsf{Empty}, [\,]) \\
&\mathtt{bfn}_{\mathtt{Off}}\ (\mathsf{Fork}\ \_\ l\ r) && \tilde{}(k : ks) && = (\mathsf{Fork}\ k\ l'\ r', \\
& && && \qquad 1 : (\mathtt{zipPlus}\ ds\ ds')) \\
&\quad \mathbf{where}\ (l', ds) && = \mathtt{bfn}_{\mathtt{Off}}\ l\ ks \\
&\qquad\qquad\ \ (r', ds') && = \mathtt{bfn}_{\mathtt{Off}}\ r\ (\mathtt{zipPlus}\ ks\ ds)
\end{aligned}
$$

$$
\begin{aligned}
&\mathtt{run}\ t = \mathbf{let}\ ds = \mathtt{bfn}_{\mathtt{Off,snd}}\ t \\
&\qquad\quad\ \ \mathbf{in}\ \mathtt{fst}\ (\mathtt{bfn}_{\mathtt{Off}}\ t\ (\mathtt{scanl}\ (+)\ 1\ ds))
\end{aligned}
$$

One Alternative

Exploit

$$\begin{aligned}\texttt{bfn}\ t\ ks \;\equiv\; & \textbf{let}\ (nt, ds) = \texttt{bfn}_{\texttt{Off}}\ t\ ks\\ & \textbf{in}\ (nt, \texttt{zipPlus}\ ks\ ds)\end{aligned}$$

One Alternative

Exploit

$$\begin{aligned} \mathtt{bfn}\ t\ ks \ &\equiv\ \textbf{let}\ (nt, ds) = \mathtt{bfn}_{\mathtt{Off}}\ t\ ks \\ &\ \ \textbf{in}\ (nt, \mathtt{zipPlus}\ ks\ ds) \end{aligned}$$

to get:

$$\begin{aligned} &\mathtt{bfn}\ \mathrm{Empty}\qquad\quad ks \qquad\quad\;\; = (\mathrm{Empty}, ks) \\ &\mathtt{bfn}\ (\mathrm{Fork}\ \_\ l\ r)\ \tilde{}(k : ks) = (\mathrm{Fork}\ k\ l'\ r', (k+1) : ks'') \\ &\quad \textbf{where}\ (l', ks')\ \ = \mathtt{bfn}\ l\ ks \\ &\qquad\qquad\ (r', ks'') = \mathtt{bfn}\ r\ ks' \end{aligned}$$

$$\begin{aligned} \mathtt{run}\ t = {} & \textbf{let}\ ds = \mathtt{bfn}_{\mathtt{Off},\mathtt{snd}}\ t \\ & \textbf{in}\ \mathtt{fst}\ (\mathtt{bfn}\ t\ (\mathtt{scanl}\ (+)\ 1\ ds)) \end{aligned}$$

Taking Stock

We now have an essentially two-phase solution:

1. First phase to compute (in *ds*) the widths of levels:

$$\begin{aligned}\mathtt{bfn}_{\mathtt{Off},\mathtt{snd}}\ \mathsf{Empty} &= [\,] \\ \mathtt{bfn}_{\mathtt{Off},\mathtt{snd}}\ (\mathsf{Fork}\ \_\ l\ r) &= 1 : (\mathtt{zipPlus}\ (\mathtt{bfn}_{\mathtt{Off},\mathtt{snd}}\ l) \\ &\qquad\qquad\qquad (\mathtt{bfn}_{\mathtt{Off},\mathtt{snd}}\ r))\end{aligned}$$

Taking Stock

We now have an essentially two-phase solution:

1. First phase to compute (in *ds*) the widths of levels:

   $$\begin{aligned}\texttt{bfn}_{\texttt{Off,snd}}\ \text{Empty} &= [\,]\\ \texttt{bfn}_{\texttt{Off,snd}}\ (\text{Fork}\ \_\ l\ r) &= 1 : (\texttt{zipPlus}\ (\texttt{bfn}_{\texttt{Off,snd}}\ l)\\ &\qquad\qquad\qquad\qquad (\texttt{bfn}_{\texttt{Off,snd}}\ r))\end{aligned}$$

2. An intermediate step ($\texttt{scanl}\ (+)\ 1\ ds$) to compute level beginnings.

Taking Stock

We now have an essentially two-phase solution:

1. First phase to compute (in *ds*) the widths of levels:

$$\begin{aligned}\texttt{bfn}_{\texttt{Off},\texttt{snd}}\ \mathsf{Empty} &= [\,]\\ \texttt{bfn}_{\texttt{Off},\texttt{snd}}\ (\mathsf{Fork}\ \_\ l\ r) &= 1 : (\texttt{zipPlus}\ (\texttt{bfn}_{\texttt{Off},\texttt{snd}}\ l)\\ &\qquad\qquad\qquad\quad (\texttt{bfn}_{\texttt{Off},\texttt{snd}}\ r))\end{aligned}$$

2. An intermediate step ($\texttt{scanl}\ (+)\ 1\ ds$) to compute level beginnings.

3. The second phase doing the actual numbering, using the original `bfn`-function (but without circular dependency).

References I

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

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

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 Press, 2007.

References II

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.
IFIP Working Group 2.1 working paper 705 WIN-2.

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

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

References III

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 Press, 2000.

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