Striktifizierung zirkulärer Programme

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

D. Seidel[2] J. Voigtländer[2]

[1]University of Minho

[2]Universität Bonn

2. Mai 2011

Multi-Traversal Programme

```
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)
```

Zirkuläre Programme [Bird 1984]

Transformation des vorherigen Programms in:

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

$$
\begin{aligned}
&\texttt{run} :: \text{Tree Int} \to \text{Tree Int}\\
&\texttt{run}\ t = \textbf{let}\ (nt, m) = \texttt{repmin}\ t\ m\ \textbf{in}\ nt
\end{aligned}
$$

Nur ein Durchlauf!

Zirkuläre Programme

Andere Verwendungen zirkulärer Programme:

- als Realisierung von Attributgrammatiken [Johnsson 1987, Kuiper & Swierstra 1987]

Zirkuläre Programme

Andere Verwendungen zirkulärer Programme:

- als Realisierung von Attributgrammatiken
  [Johnsson 1987, Kuiper & Swierstra 1987]

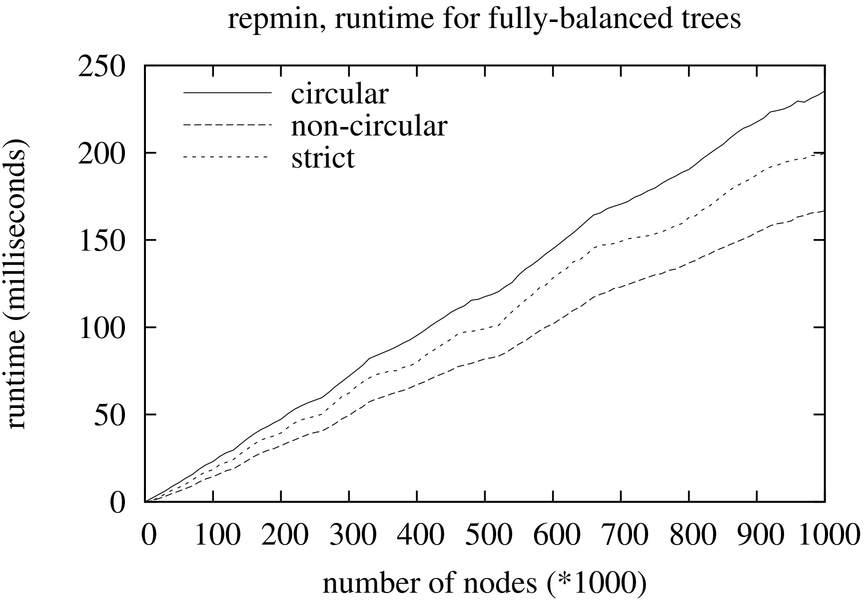
- als algorithmisches Werkzeug
  [Jones & Gibbons 1993, Okasaki 2000]

Zirkuläre Programme

Andere Verwendungen zirkulärer Programme:

- als Realisierung von Attributgrammatiken [Johnsson 1987, Kuiper & Swierstra 1987]
- als algorithmisches Werkzeug [Jones & Gibbons 1993, Okasaki 2000]
- als Ziel für Deforestationstechniken [V. 2004, Fernandes et al. 2007]
- ...

Aber:



repmin, runtime for fully-balanced trees
circular
non-circular
strict
runtime (milliseconds)
number of nodes (*1000)
0
50
100
150
200
250
0
100
200
300
400
500
600
700
800
900
1000

Unser Ziel

$$\begin{aligned}
&\texttt{repmin}\ (\mathsf{Leaf}\ n)\quad m = (\mathsf{Leaf}\ m, n)\\
&\texttt{repmin}\ (\mathsf{Fork}\ l\ r)\ m = (\mathsf{Fork}\ l'\ r', \texttt{min}\ m_1\ m_2)\\
&\quad \textbf{where}\ (l', m_1) = \texttt{repmin}\ l\ m\\
&\qquad\qquad\ (r', m_2) = \texttt{repmin}\ r\ m
\end{aligned}$$

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

$$\Big\Downarrow ?$$

$$\begin{aligned}
&\texttt{treemin}\ (\mathsf{Leaf}\ n)\quad = n\\
&\texttt{treemin}\ (\mathsf{Fork}\ l\ r) = \texttt{min}\ (\texttt{treemin}\ l)\ (\texttt{treemin}\ r)
\end{aligned}$$

$$\begin{aligned}
&\texttt{replace}\ (\mathsf{Leaf}\ n)\quad m = \mathsf{Leaf}\ m\\
&\texttt{replace}\ (\mathsf{Fork}\ l\ r)\ m = \mathsf{Fork}\ (\texttt{replace}\ l\ m)\ (\texttt{replace}\ r\ m)
\end{aligned}$$

$$\texttt{run}\ t = \texttt{replace}\ t\ (\texttt{treemin}\ t)$$

Ein Anfang

Beginnen wir mit:

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

und versuchen, das Programm zu analysieren.

Ein Anfang

Beginnen wir mit:

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

und versuchen, das Programm zu analysieren.

Was können wir über die Funktion an Hand ihres inferierten Typs erfahren?

Abhängigkeitsanalyse „for Free“

Es ergibt sich:

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

Abhängigkeitsanalyse „for Free“

Es ergibt sich:

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

Sehr interessant: die zweite Ausgabe kann unmöglich
von der zweiten Eingabe abhängen!

Abhängigkeitsanalyse „for Free“

Es ergibt sich:

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

Sehr interessant: die zweite Ausgabe kann unmöglich von der zweiten Eingabe abhängen!

Also, für alle $t :: \mathsf{Tree\ Int}$ und m_1, m_2:

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

Abhängigkeitsanalyse „for Free“

Es ergibt sich:

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

Sehr interessant: die zweite Ausgabe kann unmöglich von der zweiten Eingabe abhängen!

Also, für alle $t :: \text{Tree Int}$ und m_1, m_2:

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

Äquivalent, für alle $t :: \text{Tree Int}$ und m:

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

Aufbrechen von Zirkularität

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

Aufbrechen von Zirkularität

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

$$\Big\Downarrow\ \text{referentielle Transparenz}$$

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

Aufbrechen von Zirkularität

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

$$\Big\Downarrow \quad \text{referentielle Transparenz}$$

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

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

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

Hin zu mehr Effizienz

Statt weiter zu benutzen:

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

Hin zu mehr Effizienz

Statt weiter zu benutzen:

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

führen wir eine spezialisierte Funktion ein:

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

Hin zu mehr Effizienz

Statt weiter zu benutzen:

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

führen wir eine spezialisierte Funktion ein:

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

und können dann obige Bindung ersetzen durch:

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

Hin zu mehr Effizienz

Statt weiter zu benutzen:

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

führen wir eine spezialisierte Funktion ein:

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

und können dann obige Bindung ersetzen durch:

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

Mittels unfold/fold-Transformationen lässt sich eine direkte Definition für $\texttt{repmin}_{\texttt{snd}}$ herleiten!

Hin zu mehr Effizienz

Resultierende Definition:

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

Hin zu mehr Effizienz

Resultierende Definition:

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

Auf gleiche Weise, für $(nt, \_) = \texttt{repmin}\ t\ m$:

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

Finales Programm

$$\mathtt{run} :: \mathsf{Tree\ Int} \rightarrow \mathsf{Tree\ Int}$$

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

$$\Big\Downarrow \quad \begin{aligned} \mathtt{fst}\ (\mathtt{repmin}\ t\ m) &\equiv \mathtt{repmin}_{\mathtt{fst}}\ t\ m, \\ \mathtt{snd}\ (\mathtt{repmin}\ t\ \bot) &\equiv \mathtt{repmin}_{\mathtt{snd}}\ t \end{aligned}$$

$$\mathtt{run} :: \mathsf{Tree\ Int} \rightarrow \mathsf{Tree\ Int}$$

$$\mathtt{run}\ t = \mathtt{repmin}_{\mathtt{fst}}\ t\ (\mathtt{repmin}_{\mathtt{snd}}\ t)$$

Eine allgemeine Strategie

1. Abhängigkeiten der Ausgaben zirkulärer Aufrufe von den Eingaben erkennen.

Eine allgemeine Strategie

1. Abhängigkeiten der Ausgaben zirkulärer Aufrufe von den Eingaben erkennen.
   Soweit möglich, typ-basiert [Kobayashi 2001].

Eine allgemeine Strategie

1. Abhängigkeiten der Ausgaben zirkulärer Aufrufe von den Eingaben erkennen.
   Soweit möglich, typ-basiert [Kobayashi 2001].

2. Jeden zirkulären Aufruf in mehrere teilen, jeder nur eine der Ausgaben berechnend. Obiges Wissen nutzen, um Aufrufe zu entkoppeln.

Eine allgemeine Strategie

1. Abhängigkeiten der Ausgaben zirkulärer Aufrufe von den Eingaben erkennen.
   Soweit möglich, typ-basiert [Kobayashi 2001].

2. Jeden zirkulären Aufruf in mehrere teilen, jeder nur eine der Ausgaben berechnend. Obiges Wissen nutzen, um Aufrufe zu entkoppeln.

3. Die verschiedenen Aufrufe dahingehend spezialisieren, nur noch mit jeweils relevanten Teilen der Ein- und Ausgabe zu arbeiten.

Ein stärker herausforderndes Beispiel: Breadth-First Nummerierung [Okasaki 2000]

$$\textbf{data}\ \mathsf{Tree}\ a = \mathsf{Empty} \mid \mathsf{Fork}\ a\ (\mathsf{Tree}\ a)\ (\mathsf{Tree}\ a)$$

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

$$\begin{array}{l}
\texttt{run} :: \mathsf{Tree}\ a \to \mathsf{Tree\ Int} \\
\texttt{run}\ t = \textbf{let}\ (nt, ks) = \texttt{bfn}\ t\ (1 : ks)\ \textbf{in}\ nt
\end{array}$$

Probieren wir die allgemeine Strategie

$$\textbf{data}\ \mathsf{Tree}\ a = \mathsf{Empty} \mid \mathsf{Fork}\ a\ (\mathsf{Tree}\ a)\ (\mathsf{Tree}\ a)$$

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

Inferierter Typ von `bfn` bleibt bei

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

Probieren wir die allgemeine Strategie

$$\textbf{data}\ \text{Tree}\ a = \text{Empty} \mid \text{Fork}\ a\ (\text{Tree}\ a)\ (\text{Tree}\ a)$$

$$\begin{array}{l}
\texttt{bfn} :: \text{Tree}\ a \to [\text{Int}] \to (\text{Tree Int}, [\text{Int}]) \\
\texttt{bfn}\ \text{Empty} \qquad\quad ks \qquad\quad\ = (\text{Empty}, ks) \\
\texttt{bfn}\ (\text{Fork}\ _\ l\ r)\ \tilde{}(k : ks) = (\text{Fork}\ k\ l'\ r', (k+1) : ks'') \\
\quad \textbf{where}\ (l', ks')\ \ = \texttt{bfn}\ l\ ks \\
\qquad\qquad\quad (r', ks'') = \texttt{bfn}\ r\ ks'
\end{array}$$

Inferierter Typ von `bfn` bleibt bei

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

Zu komplexe Form der Abängigkeit der Ausgabeliste von der Eingabeliste!

Etwas Hilfe

Beachte: zweite Ausgabe von `bfn` wird stets aus der zweiten Eingabe durch (gegebenenfalls wiederholtes) Inkrementieren von Listenelementen gebildet.

Etwas Hilfe

Beachte: zweite Ausgabe von `bfn` wird stets aus der zweiten Eingabe durch (gegebenenfalls wiederholtes) Inkrementieren von Listenelementen gebildet.

Leiten wir also eine Variante mit

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

her, wobei:

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

Etwas Hilfe

Ergebnis ziemlich direkter Herleitung:

$$
\begin{array}{l}
\mathtt{bfn_{Off}} :: \mathrm{Tree}\ a \to [\mathrm{Int}] \to (\mathrm{Tree\ Int}, [\mathrm{Int}]) \\
\mathtt{bfn_{Off}}\ \mathrm{Empty}\qquad ks \qquad\quad = (\mathrm{Empty}, [\,]) \\
\mathtt{bfn_{Off}}\ (\mathrm{Fork}\ \_\ l\ r)\ \tilde{}(k : ks) = (\mathrm{Fork}\ k\ l'\ r', \\
\qquad\qquad\qquad\qquad\qquad\qquad 1 : (\mathtt{zipPlus}\ ds\ ds')) \\
\quad \textbf{where}\ (l', ds)\ \ = \mathtt{bfn_{Off}}\ l\ ks \\
\qquad\qquad\ (r', ds') = \mathtt{bfn_{Off}}\ r\ (\mathtt{zipPlus}\ ks\ ds)
\end{array}
$$

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

Anwendung unserer allgemeinen Strategie

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

$$\Big\Downarrow \quad \text{Teilung eines Aufrufs}$$

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

Entfernen einer der beiden Zirkularitäten

Von

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

zu

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

wobei:

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

Spezialisieren ...

... führt zu:

$$
\begin{aligned}
&\texttt{bfn}_{\texttt{Off},\texttt{snd}} :: \mathsf{Tree}\ a \rightarrow [\mathsf{Int}]\\
&\texttt{bfn}_{\texttt{Off},\texttt{snd}}\ \mathsf{Empty} \qquad\quad = [\,]\\
&\texttt{bfn}_{\texttt{Off},\texttt{snd}}\ (\mathsf{Fork}\ \_\ l\ r) = 1 : (\texttt{zipPlus}\ ds\ ds')\\
&\quad \textbf{where}\ ds\ \ = \texttt{bfn}_{\texttt{Off},\texttt{snd}}\ l\\
&\qquad\qquad\quad ds' = \texttt{bfn}_{\texttt{Off},\texttt{snd}}\ r
\end{aligned}
$$

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

Betrachtung von $ks = \texttt{zipPlus}\ (1 : ks)\ ds$

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

Betrachtung von $ks = \texttt{zipPlus}\ (1 : ks)\ ds$

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

Betrachtung von $ks = \texttt{zipPlus}\ (1 : ks)\ ds$

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

Betrachtung von $ks = \texttt{zipPlus}\ (1 : ks)\ ds$

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

Betrachtung von $ks = \texttt{zipPlus}\ (1 : ks)\ ds$

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

Betrachtung von $ks = \texttt{zipPlus}\ (1 : ks)\ ds$

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

Betrachtung von $ks = \texttt{zipPlus}\ (1 : ks)\ ds$

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

Betrachtung von $ks = \texttt{zipPlus}\ (1 : ks)\ ds$

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

(Fast) am Ziel:

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

Kann direkt in OCaml ausgedrückt werden!

(Fast) am Ziel:

$$
\begin{aligned}
&\mathtt{run} :: \mathsf{Tree}\ a \to \mathsf{Tree\ Int}\\
&\mathtt{run}\ t = \mathbf{let}\ nt = \mathtt{fst}\ (\mathtt{bfn}_{\mathtt{Off}}\ t\ (1 : ks))\\
&\phantom{\mathtt{run}\ t = \mathbf{let}\ } ds = \mathtt{bfn}_{\mathtt{Off},\mathtt{snd}}\ t\\
&\phantom{\mathtt{run}\ t = \mathbf{let}\ } ks = (\mathtt{tail}\ (\mathtt{scanl}\ (+)\ 1\ ds)) +\!\!+\\
&\phantom{\mathtt{run}\ t = \mathbf{let}\ ks = } (\mathtt{repeat}\ (\mathtt{last}\ (\mathtt{scanl}\ (+)\ 1\ ds)))\\
&\phantom{\mathtt{run}\ t = {}} \mathbf{in}\ nt
\end{aligned}
$$

Kann direkt in OCaml ausgedrückt werden!

Oder noch etwas optimiert:

$$
\begin{aligned}
&\mathtt{run} :: \mathsf{Tree}\ a \to \mathsf{Tree\ Int}\\
&\mathtt{run}\ t = \mathbf{let}\ ds = \mathtt{bfn}_{\mathtt{Off},\mathtt{snd}}\ t\\
&\phantom{\mathtt{run}\ t = {}} \mathbf{in}\ \mathtt{fst}\ (\mathtt{bfn}_{\mathtt{Off}}\ t\ (\mathtt{scanl}\ (+)\ 1\ ds))
\end{aligned}
$$

Noch versteckte Ineffizienz

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

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

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

Eine Alternative

Ausnutzen von

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

Eine Alternative

Ausnutzen von

$$\mathtt{bfn}\ t\ \mathit{ks} \;\equiv\; \textbf{let}\ (\mathit{nt}, \mathit{ds}) = \mathtt{bfn}_{\mathtt{Off}}\ t\ \mathit{ks}\ \textbf{in}\ (\mathit{nt}, \mathtt{zipPlus}\ \mathit{ks}\ \mathit{ds})$$

um zu erhalten:

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

$$\begin{array}{l}
\mathtt{run}\ t = \textbf{let}\ \mathit{ds} = \mathtt{bfn}_{\mathtt{Off,snd}}\ t \\
\qquad\quad\ \textbf{in}\ \mathtt{fst}\ (\mathtt{bfn}\ t\ (\mathtt{scanl}\ (+)\ 1\ \mathit{ds}))
\end{array}$$

Bestandsaufnahme

Wir haben nun im Grunde eine Zwei-Phasen-Lösung:

1. Erste Phase, um (in *ds*) die Breiten der einzelnen „Level" zu berechnen:

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

Bestandsaufnahme

Wir haben nun im Grunde eine Zwei-Phasen-Lösung:

1. Erste Phase, um (in *ds*) die Breiten der einzelnen „Level“ zu berechnen:

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

2. Ein Zwischenschritt ($\mathtt{scanl}\ (+)\ 1\ ds$), um die „Levelanfänge“ zu berechnen.

Bestandsaufnahme

Wir haben nun im Grunde eine Zwei-Phasen-Lösung:

1. Erste Phase, um (in *ds*) die Breiten der einzelnen „Level“ zu berechnen:

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

2. Ein Zwischenschritt ($\mathtt{scanl}\ (+)\ 1\ ds$), um die „Levelanfänge“ zu berechnen.

3. Zweite Phase führt eigentliche Nummerierung durch, unter Verwendung der ursprünglichen `bfn`-Funktion (aber ohne Zirkularität).

Rück- und Ausblick

Bisher:

- Kombination bekannter Analyse- und Transformationstechniken
- Verbindung zu klassischer Implementierung von Attributgrammatiken
- konkret für `bfn`, mehrere Varianten möglich und interessant

Rück- und Ausblick

Bisher:

- Kombination bekannter Analyse- und Transformationstechniken
- Verbindung zu klassischer Implementierung von Attributgrammatiken
- konkret für `bfn`, mehrere Varianten möglich und interessant

Weiter:

- Potential für „Entdeckung“ neuer Algorithmen
- Zusammenspiel mit anderen Techniken zur Programmtransformation

Referenzen 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.

Referenzen 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.

Referenzen 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.