Colloquium at TU München, November 18, 2003

Elimination of Intermediate Results in Functional Programs

Janis Voigtländer

Dresden University of Technology

http://wwwtcs.inf.tu-dresden.de/∼voigt

Supported by the "Deutsche Forschungsgemeinschaft" under grant KU 1290/2-4.

Outline

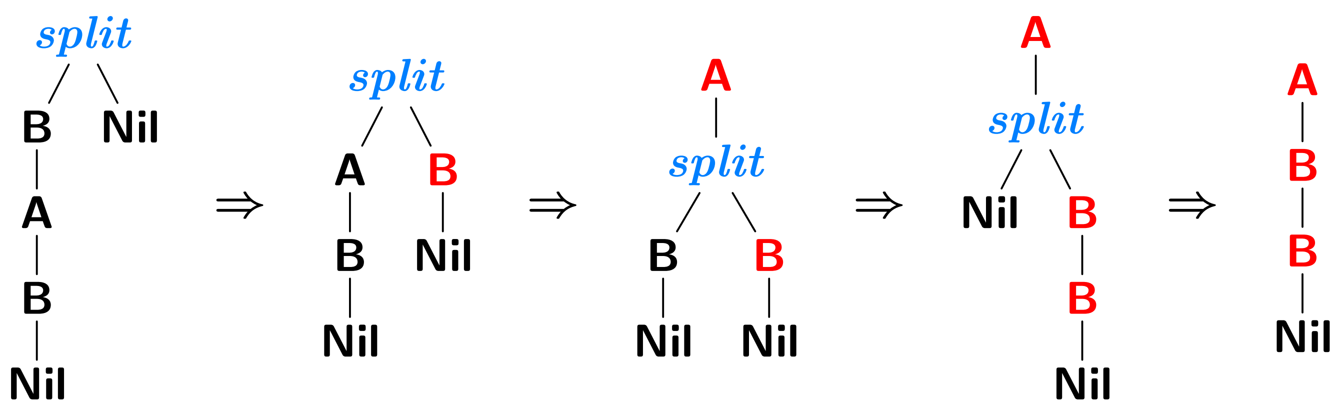
1. **Functional programs and intermediate results**
2. **Deforestation**
3. **Tree transducer composition**
4. **Surprise**

Why functional programming matters

- **declarative specifications, but executable**
- **no side effects $\Rightarrow$ referential transparency**
- **clear semantics $\Rightarrow$ equational reasoning**
- **encourages use of high-level programming constructs**
- **high potential for modularisation of programs**

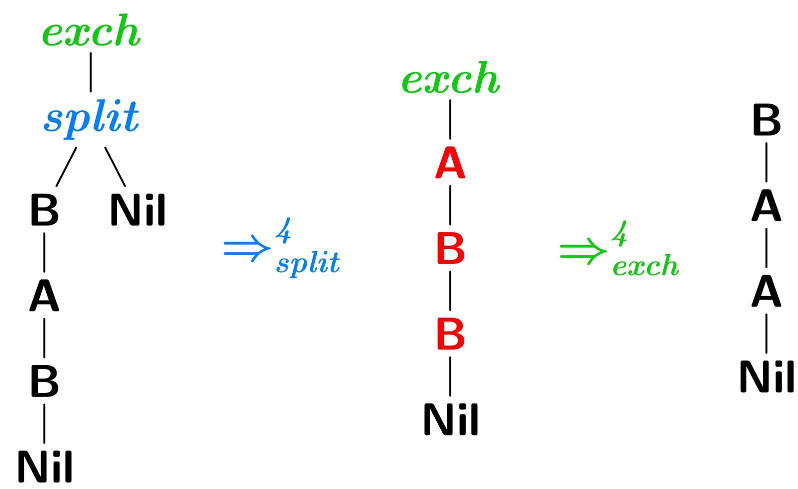
Function definition by structural recursion

$$
\begin{aligned}
&\text{data } \mathsf{List} \;=\; \mathsf{A}\ \mathsf{List} \mid \mathsf{B}\ \mathsf{List} \mid \mathsf{Nil}\\
&split :: \mathsf{List} \longrightarrow \mathsf{List} \longrightarrow \mathsf{List}\\
&split\ (\mathsf{A}\ u)\ y \;=\; \mathsf{A}\ (split\ u\ y)\\
&split\ (\mathsf{B}\ u)\ y \;=\; split\ u\ (\mathsf{B}\ y)\\
&split\ \ \mathsf{Nil}\ \ y \;=\; y
\end{aligned}
$$



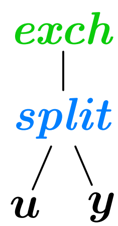
Modularity vs. efficiency

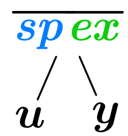
$$
\begin{aligned}
&\mathit{exch} :: \textsf{List} \longrightarrow \textsf{List}\\
&\mathit{exch}\ (\textsf{A}\ v) = \textsf{B}\ (\mathit{exch}\ v)\\
&\mathit{exch}\ (\textsf{B}\ v) = \textsf{A}\ (\mathit{exch}\ v)\\
&\mathit{exch}\ \ \textsf{Nil} = \textsf{Nil}\\
&\mathit{main}\ t = \mathit{exch}\ (\mathit{split}\ t\ \textsf{Nil})
\end{aligned}
$$



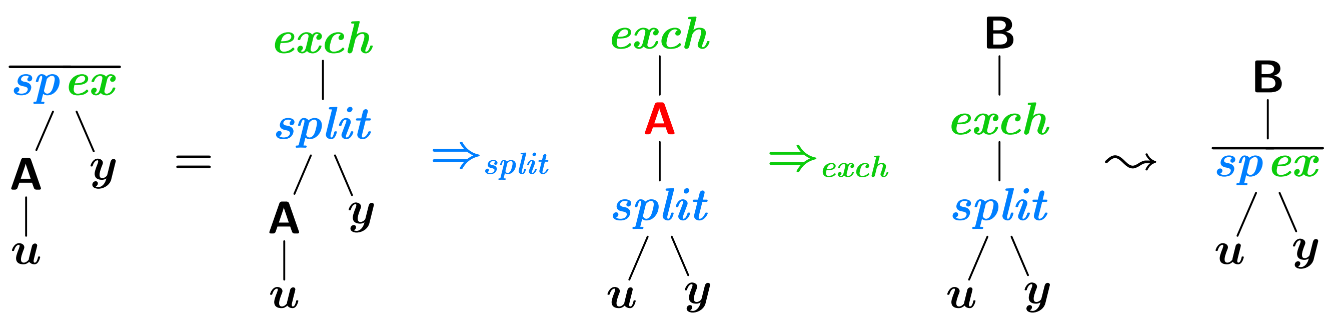
Intermediate results lead to inefficiencies!

Deforestation [Wadler, 1990]

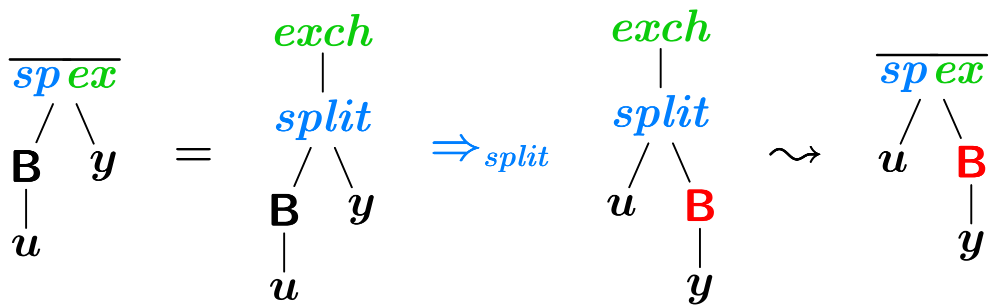
Key ideas: folding 

to 

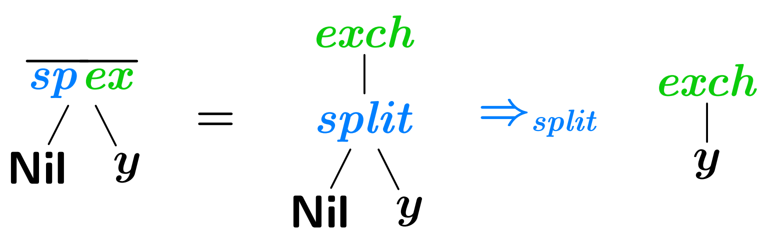
and "translating" right-hand sides of *split* with rules of *exch*:

1. 

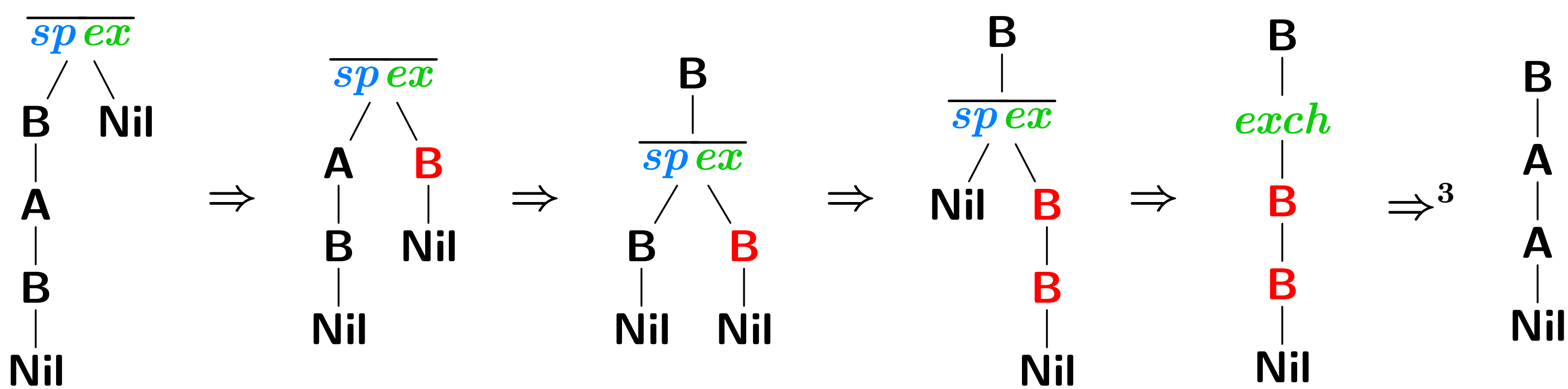
2.



3.



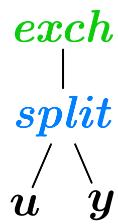
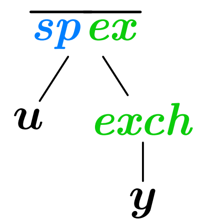
Deforestation eliminated only part of the intermediate result:

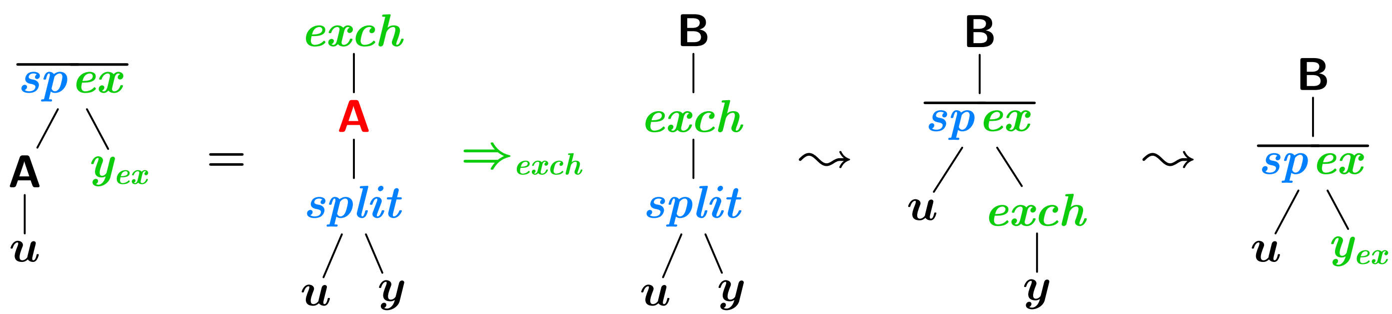


Tree transducer theory comes to the rescue

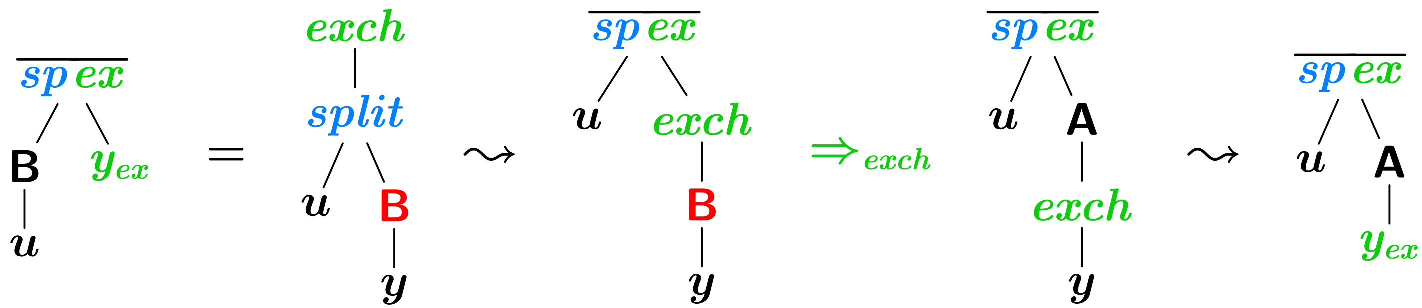
- **Tree transducers are:**
  - **finite devices computing tree translations (tree automata with output)**
  - **used as models for syntax-directed semantics**
  - **used as models for fragments of XML query languages**
  - **often, special functional programs**
- **Their theory studies:**
  - **complexity, decidability issues**
  - **expressive power of different classes**
  - **closure under composition**

Example: *MAC* ; *TOP* ⊆ *MAC* [Engelfriet & Vogler, 1985]

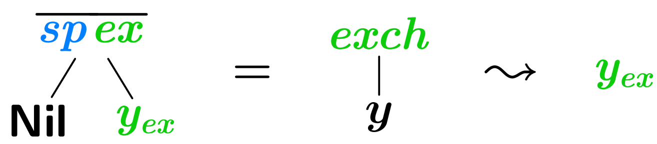
Approach: replace  by  and hence assume that $\overline{sp\,ex}$ has as second argument the translation of *split*'s accumulating parameter with *exch*:

1. 

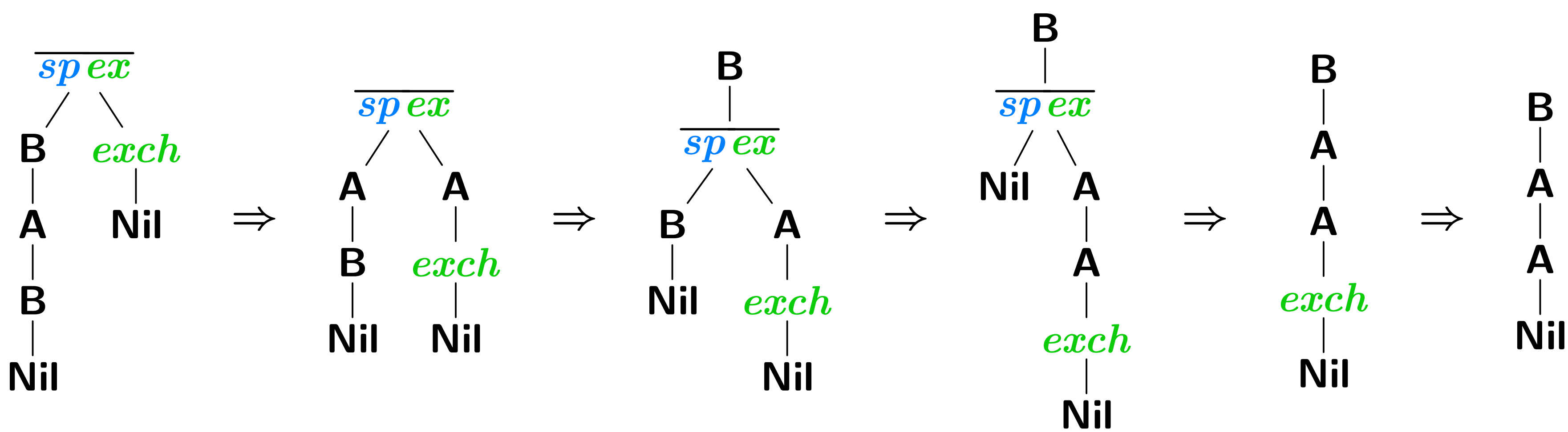
2.



3.



Production of intermediate result completely avoided:



How about more interesting cases?

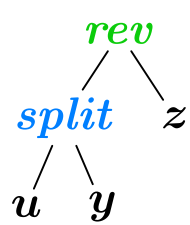
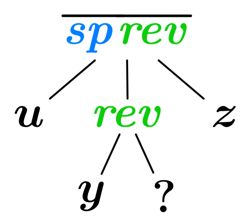
$rev :: \mathsf{List} \longrightarrow \mathsf{List} \longrightarrow \mathsf{List}$

$rev\ (\mathsf{A}\ v)\ z = rev\ v\ (\mathsf{A}\ z)$

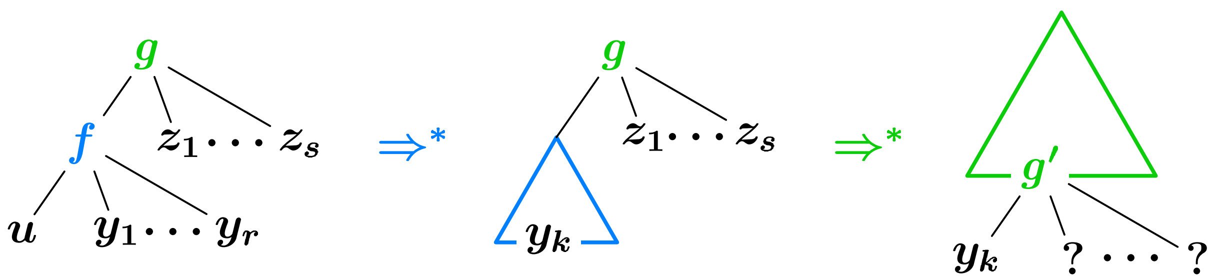
$rev\ (\mathsf{B}\ v)\ z = rev\ v\ (\mathsf{B}\ z)$

$rev\ \ \mathsf{Nil}\ \ z = z$

$main\ t = rev\ (split\ t\ \mathsf{Nil})\ \mathsf{Nil}$

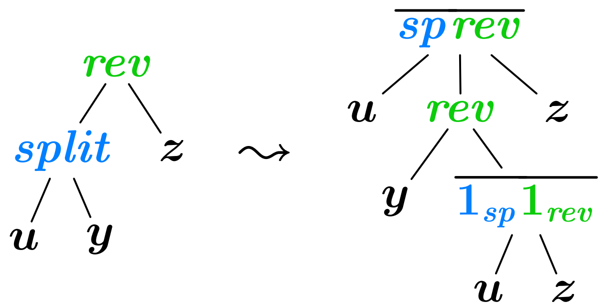
Have to replace  **by**  **, but how exactly?**

In general, what about the values in question in:



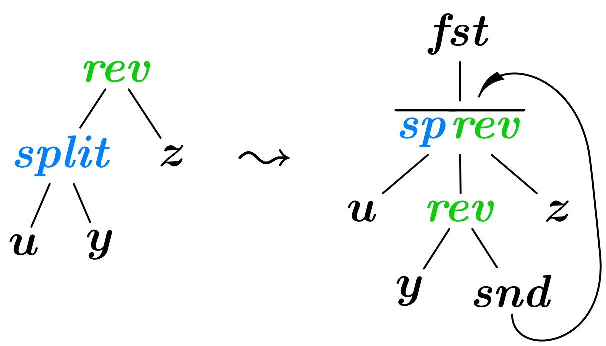
Two solutions

1. **Using auxiliary functions:**



[V. & Kühnemann] Composition of functions with accumulating parameters. ***J. Funct. Prog.***, to appear.

2. **Using tupling and circular bindings:**



[V.] Using circular programs to deforest in accumulating parameters. ***Higher-Order and Symb. Comp.***, to appear.

After post-processing (in both cases):

$$\overline{sp\,rev}' :: \mathsf{List} \longrightarrow \mathsf{List} \longrightarrow \mathsf{List}$$

$$\overline{sp\,rev}'\ (\mathsf{A}\ u)\ z = \overline{sp\,rev}'\ u\ (\mathsf{A}\ z)$$

$$\overline{sp\,rev}'\ (\mathsf{B}\ u)\ z = \mathsf{B}\ (\overline{sp\,rev}'\ u\ z)$$

$$\overline{sp\,rev}'\ \mathsf{Nil}\ z = z$$

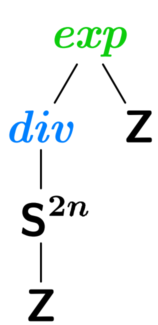
$$main'\ t = \overline{sp\,rev}'\ t\ \mathsf{Nil}$$

What about efficiency?

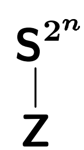
$$
\begin{array}{l}
\text{data } \mathsf{Nat} = \mathsf{S}\ \mathsf{Nat} \mid \mathsf{Z} \\
div, div' :: \mathsf{Nat} \to \mathsf{Nat} \\
div\ (\mathsf{S}\ u) = div'\ u \\
div\ \mathsf{Z} = \mathsf{Z} \\
div'\ (\mathsf{S}\ u) = \mathsf{S}\ (div\ u) \\
div'\ \mathsf{Z} = \mathsf{Z} \\
exp :: \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat} \\
exp\ (\mathsf{S}\ v)\ z = exp\ v\ (exp\ v\ z) \\
exp\ \mathsf{Z}\ z = \mathsf{S}\ z \\
main\ t = exp\ (div\ t)\ \mathsf{Z}
\end{array}
$$

$\rightsquigarrow$

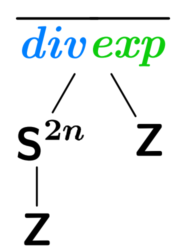
$$
\begin{array}{l}
\overline{div\,exp}, \overline{div'\,exp} :: \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat} \\
\overline{div\,exp}\ (\mathsf{S}\ u)\ z = \overline{div'\,exp}\ u\ z \\
\overline{div\,exp}\ \mathsf{Z}\ z = \mathsf{S}\ z \\
\overline{div'\,exp}\ (\mathsf{S}\ u)\ z = \overline{div\,exp}\ u\ (\overline{div\,exp}\ u\ z) \\
\overline{div'\,exp}\ \mathsf{Z}\ z = \mathsf{S}\ z \\
main'\ t = \overline{div\,exp}\ t\ \mathsf{Z}
\end{array}
$$



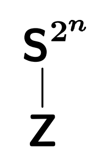
$\Rightarrow^{2 \cdot 2^n + 2n}$



, but



$\Rightarrow^{3 \cdot 2^n - 2}$



Formal efficiency analysis

- **Measure: number of *call-by-need* reduction steps**
- **Approach:**
  - **annotate original and composed programs to reflect performed reduction steps in the output**
  - **push annotation of composed program backwards through the composition construction**
  - **compare and manipulate resulting annotations of the original program to obtain sufficient criteria**

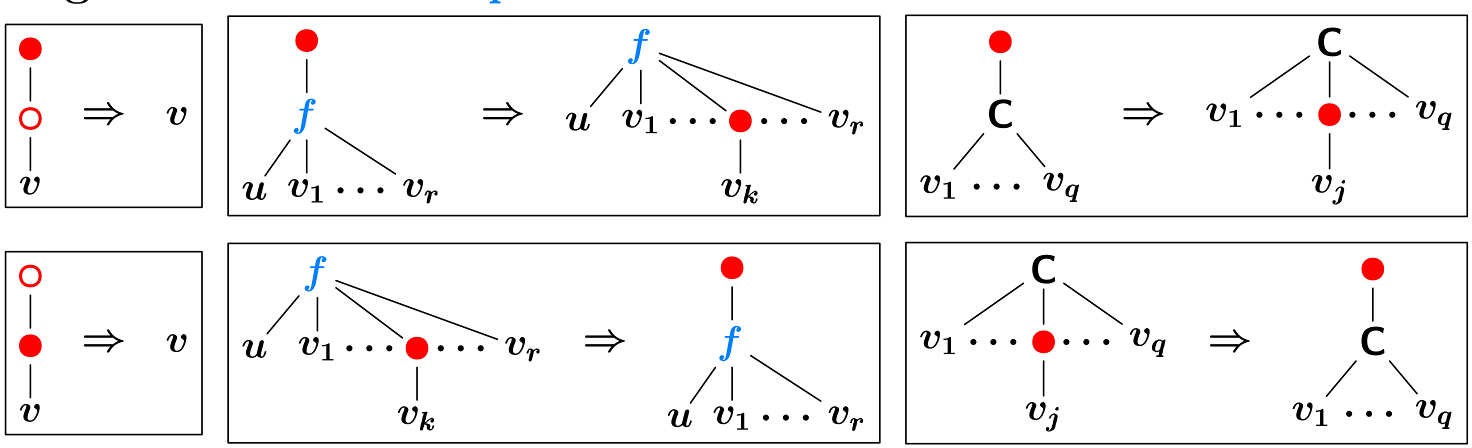
[V.] Conditions for efficiency improvement by tree transducer composition. *Proc. RTA'02*, LNCS 2378.

An Example Criterion at Work

Annotated program:

$$
\begin{array}{llll}
\mathit{split}\ (\mathsf{A}\ u)\ y & = & \bullet\ (\mathsf{A}\ (\mathit{split}\ u\ (\circ\ y))) & \qquad \mathit{rev}\ (\mathsf{A}\ v)\ z \ = \ \mathit{rev}\ v\ (\mathsf{A}\ z) \\
\mathit{split}\ (\mathsf{B}\ u)\ y & = & \mathit{split}\ u\ (\circ\ (\bullet\ (\mathsf{B}\ y))) & \qquad \mathit{rev}\ (\mathsf{B}\ v)\ z \ = \ \mathit{rev}\ v\ (\mathsf{B}\ z) \\
\mathit{split}\ \ \mathsf{Nil}\ \ y & = & y & \qquad \mathit{rev}\ \ \mathsf{Nil}\ \ z \ = \ z \\
 & & & \qquad \mathit{rev}\ (\circ\ v)\ z \ = \ \circ\ (\mathit{rev}\ v\ z) \\
\mathit{main}\ t & = & \mathit{rev}\ (\mathit{split}\ t\ (\circ\ (\bullet\ \mathsf{Nil})))\ \mathsf{Nil} & \qquad \mathit{rev}\ (\bullet\ v)\ z \ = \ \bullet\ (\mathit{rev}\ v\ z)
\end{array}
$$

Since *split* is *context-linear* and *-nondeleting*, and *rev* is *linear* and *nondeleting*, the following rules may be used with the aim of eliminating all ○-symbols in the right-hand sides of *split*:



Implementation

Haskell$^+$ system

- research tool
- annotated input programs:

$$
\begin{array}{l}
\text{beginmag } \mathit{Split}\ [\text{Mac}] \\
\quad \text{input } \mathsf{Data} \\
\quad \text{syn } \mathit{split} :: \mathsf{List} \longrightarrow \mathsf{List} \longrightarrow \mathsf{List} \\
\quad \mathit{split}\ (\mathsf{A}\ u)\ y \ = \ \mathsf{A}\ (\mathit{split}\ u\ y) \\
\quad \mathit{split}\ (\mathsf{B}\ u)\ y \ = \ \mathit{split}\ u\ (\mathsf{B}\ y) \\
\quad \mathit{split}\ \ \mathsf{Nil}\ \ y \ = \ y \\
\text{endmag}
\end{array}
$$

- requires user interaction

GHC compiler pass

- prototype implementation
- ordinary Haskell source, e.g.:

$$
\begin{array}{lcl}
\mathit{split}\ x\ y & = & \text{case } x \text{ of} \\
 & & \quad \mathsf{A}\ u \longrightarrow \mathsf{A}\ (\mathit{split}\ u\ y) \\
 & & \quad \mathsf{B}\ u \longrightarrow \mathit{split}\ u\ (\mathsf{B}\ y) \\
 & & \quad \mathsf{Nil} \ \longrightarrow y
\end{array}
$$

- integration as an optimization pass in compiler pipeline

Deaccumulation

Surprise: sometimes it is a good idea to transform an efficient program into an inefficient one.

$$\begin{aligned} split\ (\mathsf{A}\ u)\ y &= \mathsf{A}\ (split\ u\ y) \\ split\ (\mathsf{B}\ u)\ y &= split\ u\ (\mathsf{B}\ y) \\ split\ \ \mathsf{Nil}\ \ y &= y \\ main\ t &= split\ t\ \mathsf{Nil} \end{aligned}$$

$\leadsto$

$$\begin{aligned} split'\ (\mathsf{A}\ u) &= \mathsf{A}\ (split'\ u) \\ split'\ (\mathsf{B}\ u) &= app\ (split'\ u)\ (\mathsf{B}\ \mathsf{Nil}) \\ split'\ \ \mathsf{Nil} &= \mathsf{Nil} \\ app\ (\mathsf{A}\ u)\ y &= \mathsf{A}\ (app\ u\ y) \\ app\ (\mathsf{B}\ u)\ y &= \mathsf{B}\ (app\ u\ y) \\ app\ \ \mathsf{Nil}\ \ y &= y \\ main'\ t &= split'\ t \end{aligned}$$

linear runtime — quadratic runtime

Why?

Improving Provability

Proving idempotence of the original program, i.e.,

$$\mathit{split}\ (\mathit{split}\ t\ \textsf{Nil})\ \textsf{Nil}\ =\ \mathit{split}\ t\ \textsf{Nil}\,,$$

by induction on t requires a generalization that is difficult to find automatically.

In contrast, an inductive proof of

$$\mathit{split}'\ (\mathit{split}'\ t)\ =\ \mathit{split}'\ t$$

is much easier.

[Giesl, Kühnemann & V.] Deaccumulation — Improving Provability. *Proc. ASIAN'03*, LNCS, to appear.

References

[Engelfriet & Vogler, 1985] Macro tree transducers. *Journal of Computer and System Sciences*, **31**, 71–145.

[Giesl, Kühnemann & Voigtländer, 2003] Deaccumulation — Improving provability. *Proc. of: Asian Computing Science Conference.* LNCS, to appear. Springer-Verlag.

[Kühnemann, 1998] Benefits of tree transducers for optimizing functional programs. *Proc. of: Foundations of Software Technology & Theoretical Computer Science.* LNCS 1530. Springer-Verlag.

[Kühnemann, 1999] Comparison of deforestation techniques for functional programs and for tree transducers. *Proc. of: Functional and Logic Programming.* LNCS 1722. Springer-Verlag.

[Voigtländer & Kühnemann, 2004] Composition of functions with accumulating parameters. *Journal of Functional Programming,* to appear.

[Voigtländer, 2002a] Using circular programs to deforest in accumulating parameters. *Proc. of: Asian Symposium on Partial Evaluation and Semantics-Based Program Manipulation.* ACM Press. Extended version to appear in *Higher-Order and Symbolic Computation.*

[Voigtländer, 2002b] Conditions for efficiency improvement by tree transducer composition. *Proc. of: Rewriting Techniques and Applications.* LNCS 2378. Springer-Verlag.

[Wadler, 1990] Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, **73**, 231–248.