ASIA-PEPM, September 14, 2002

Using Circular Programs to Deforest in Accumulating Parameters

Janis Voigtländer

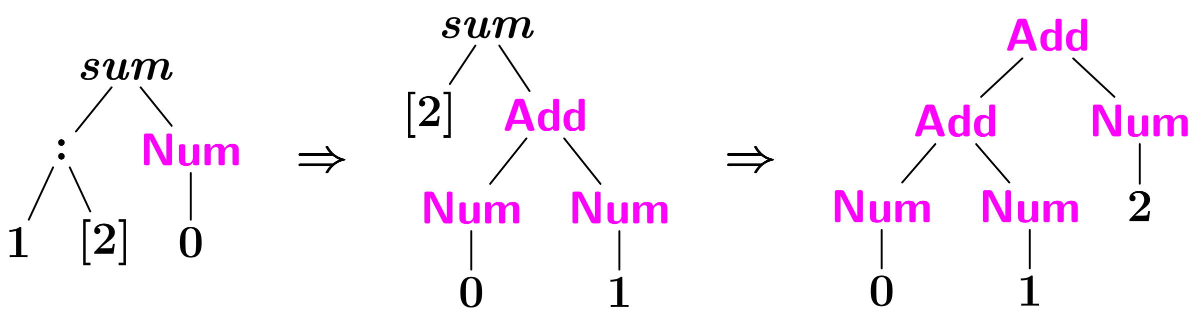
Dresden University of Technology

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

Supported by the "Deutsche Forschungsgemeinschaft" under grant KU 1290/2-1 and by the "Gesellschaft von Freunden und Förderern der TU Dresden" with a travel grant.

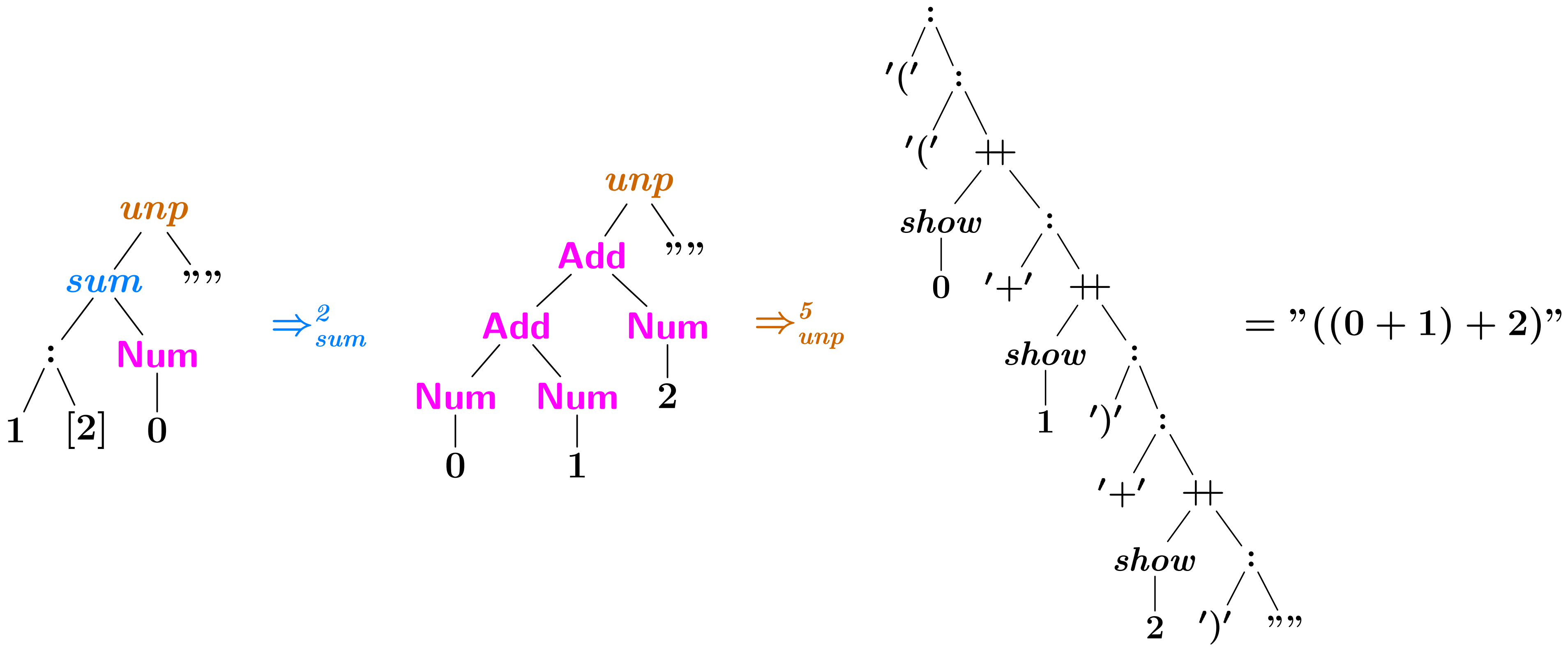
Functions with Accumulating Parameters

$$\begin{array}{l}
\text{data } \textsf{Term} \; = \; \textsf{Num Int} \mid \textsf{Add Term Term} \\
\\
sum :: [\textsf{Int}] \; \longrightarrow \; \textsf{Term} \; \longrightarrow \; \textsf{Term} \\
sum \quad [x] \quad y \; = \; \textsf{Add}\; y\; (\textsf{Num}\; x) \\
sum\; (x : xs)\; y \; = \; sum\; xs\; (\textsf{Add}\; y\; (\textsf{Num}\; x))
\end{array}$$

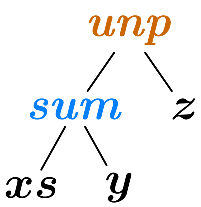


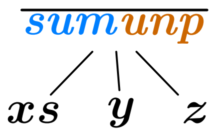
$$\begin{array}{l}
unp :: \textsf{Term} \; \longrightarrow \; \textsf{String} \; \longrightarrow \; \textsf{String} \\
unp \quad (\textsf{Num}\; x) \quad z \; = \; show\; x \mathbin{+\!\!+} z \\
unp\; (\textsf{Add}\; x_1\; x_2)\; z \; = \; '(' : unp\; x_1\; ('+' : unp\; x_2\; (')' : z))
\end{array}$$

Intermediate Results



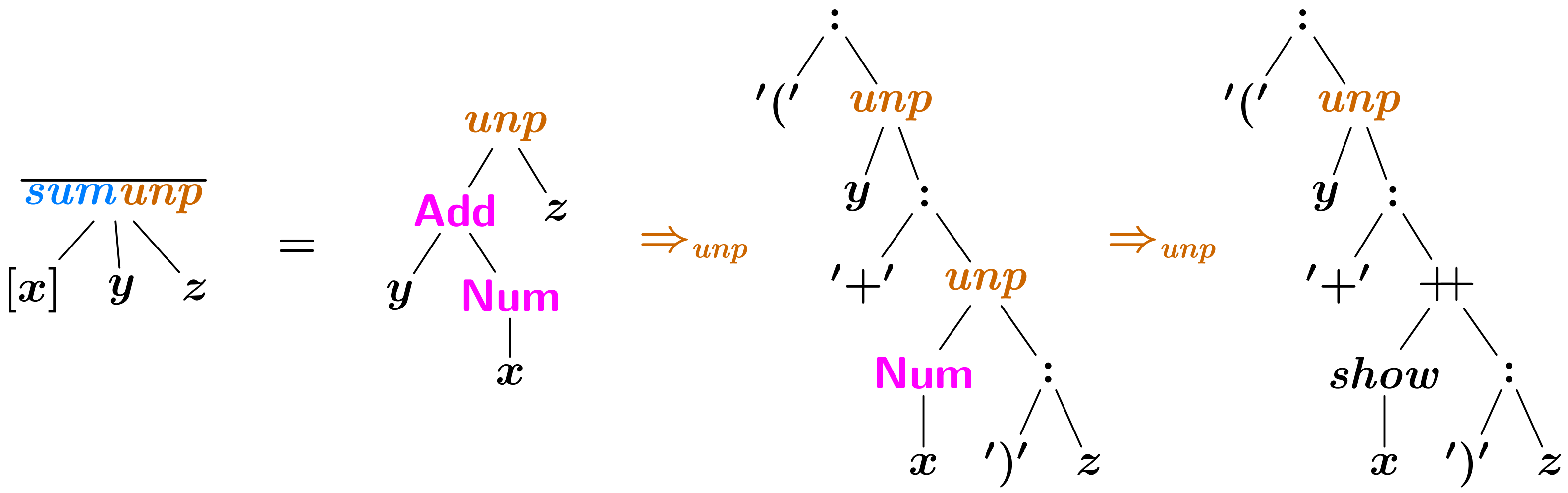
Deforestation [Wad90, HJ92]

Key-ideas: folding 

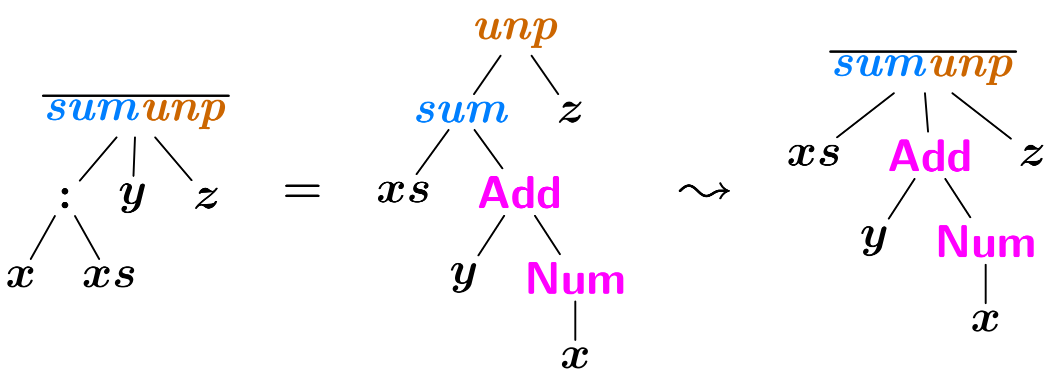
to 

, and "translating" right-hand sides of *sum* **with rules of** *unp***:**

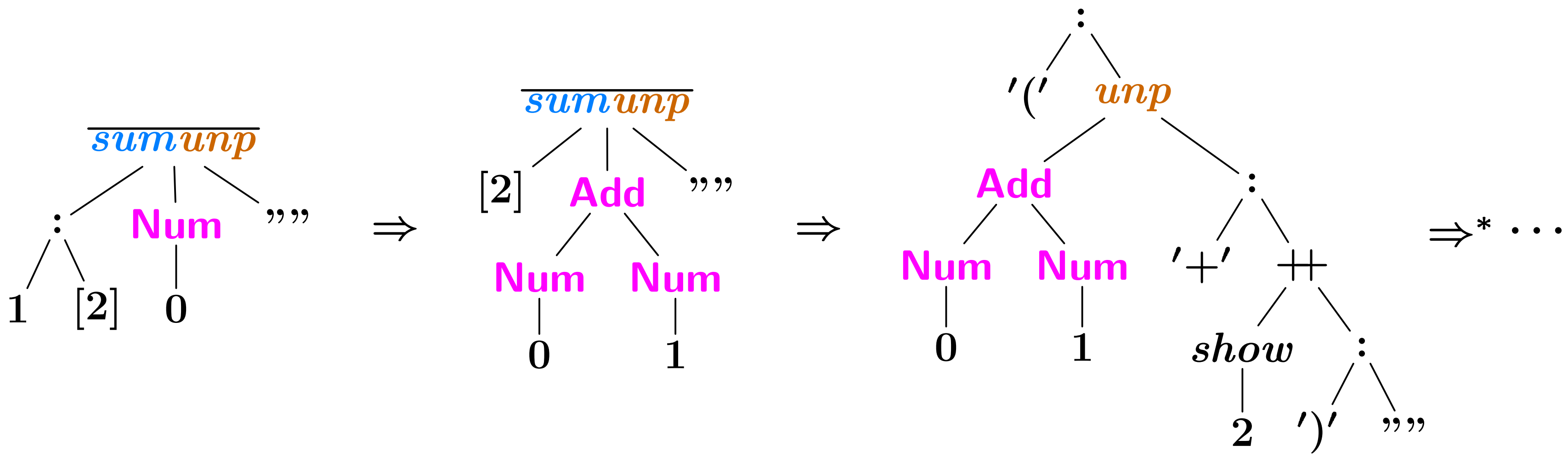
1.



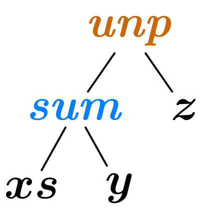
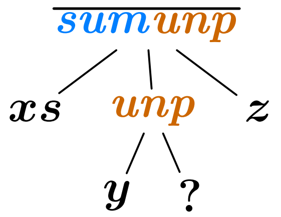
2.



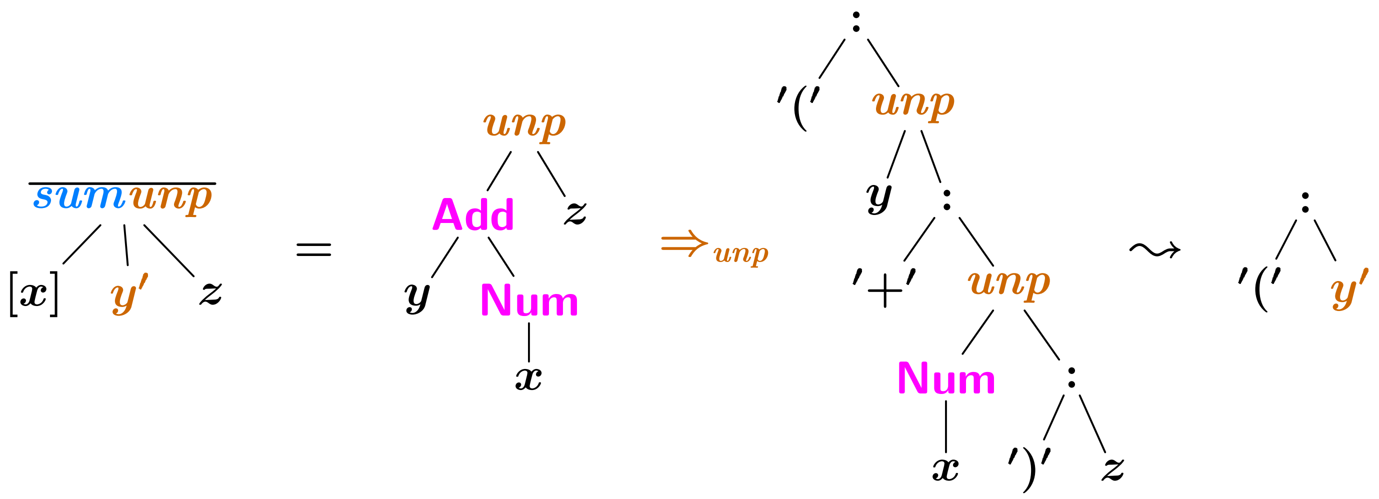
Deforestation eliminated only part of the intermediate result:



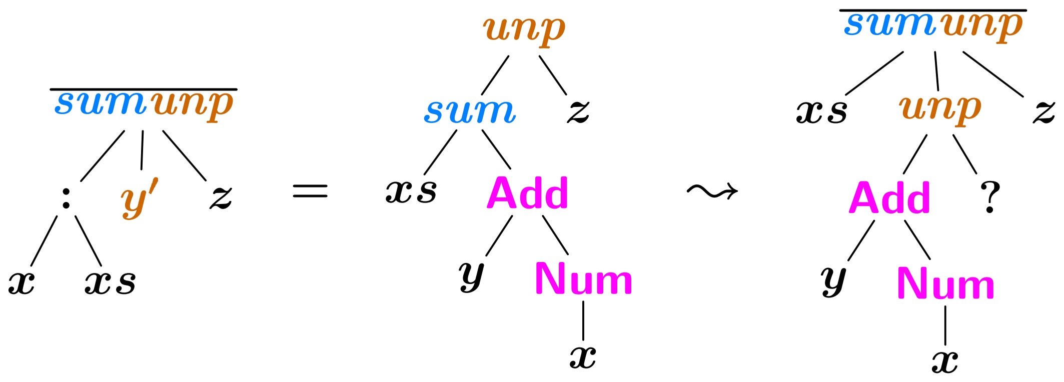
How to Deforest in Accumulating Parameters ?

Approach: replace  by , and hence assume that $\overline{sum\,unp}$ has as second argument the correct translation of sum's accumulating parameter with unp:

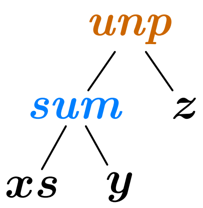
1.

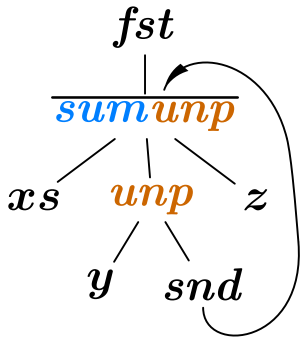


2.



Idea: let $\overline{sum\,unp}$ return a tuple, consisting of the composition of sum and unp (as before) and additionally the parameter value with which unp "arrives" at the accumulating parameter of sum,

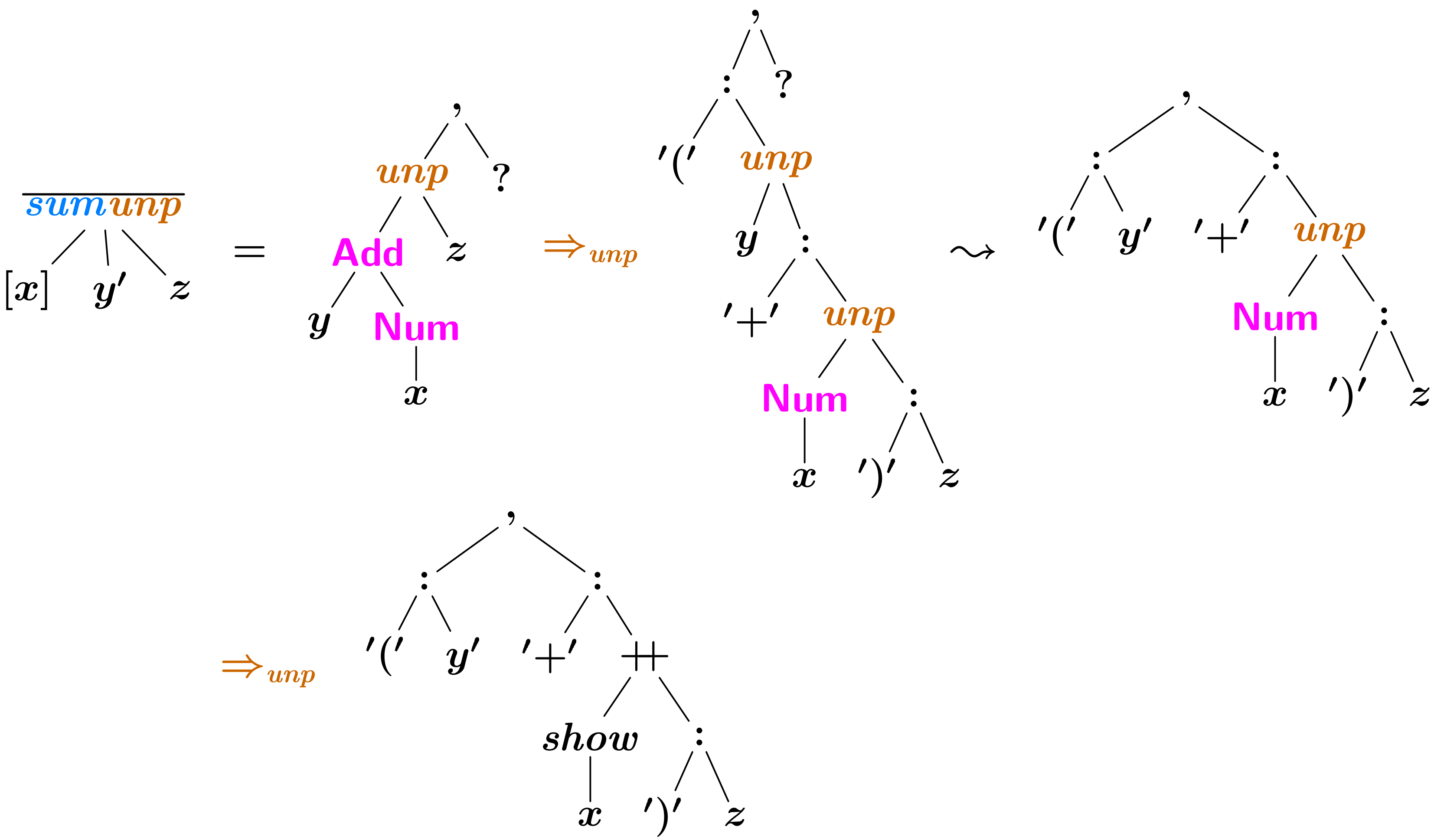
i.e., 

will then be replaced by 

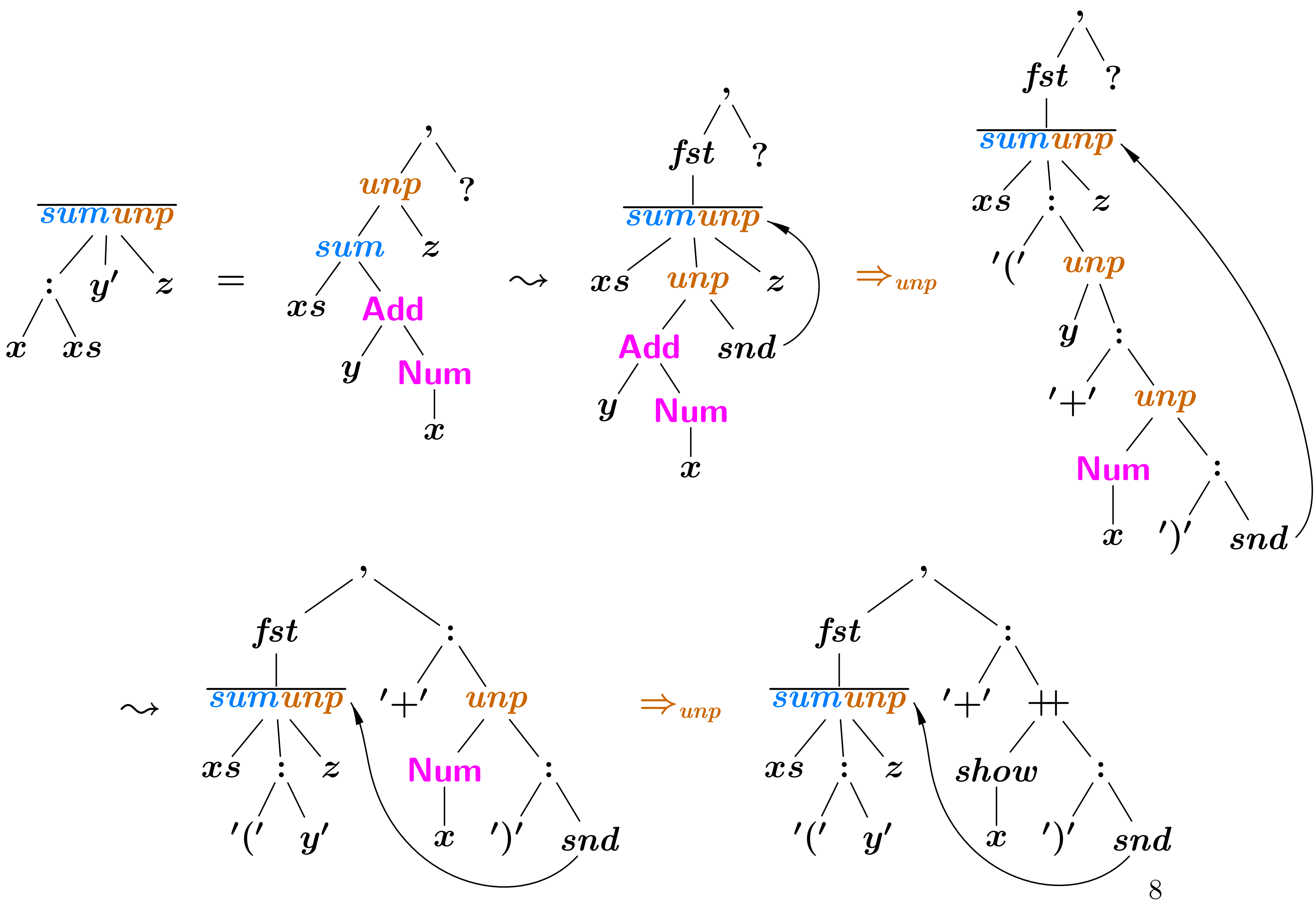
.

Lazy Composition

1.



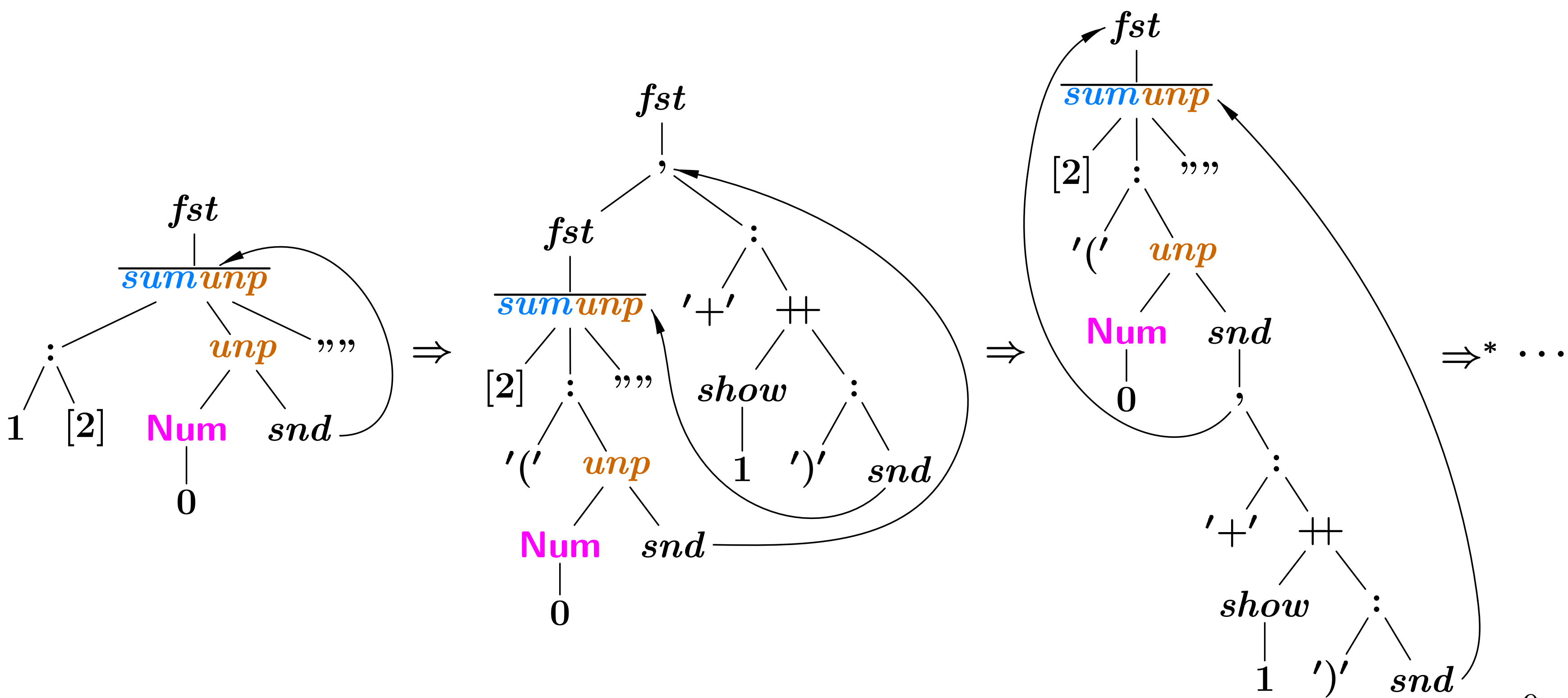
2.



$\overline{sum\,unp}$
:
y'
z
x
xs
=
,
unp
?
sum
z
xs
Add
y
Num
x
$\leadsto$
,
fst
?
$\overline{sum\,unp}$
xs
unp
z
Add
snd
y
Num
x
$\Rightarrow_{unp}$
,
fst
?
$\overline{sum\,unp}$
xs
:
z
$'('$
unp
y
:
$'+'$
unp
Num
:
x
$')'$
snd
$\leadsto$
,
fst
:
$\overline{sum\,unp}$
$'+'$
unp
xs
:
z
Num
:
$'('$
y'
x
$')'$
snd
$\Rightarrow_{unp}$
,
fst
:
$\overline{sum\,unp}$
$'+'$
$++$
xs
:
z
$show$
:
$'('$
y'
x
$')'$
snd

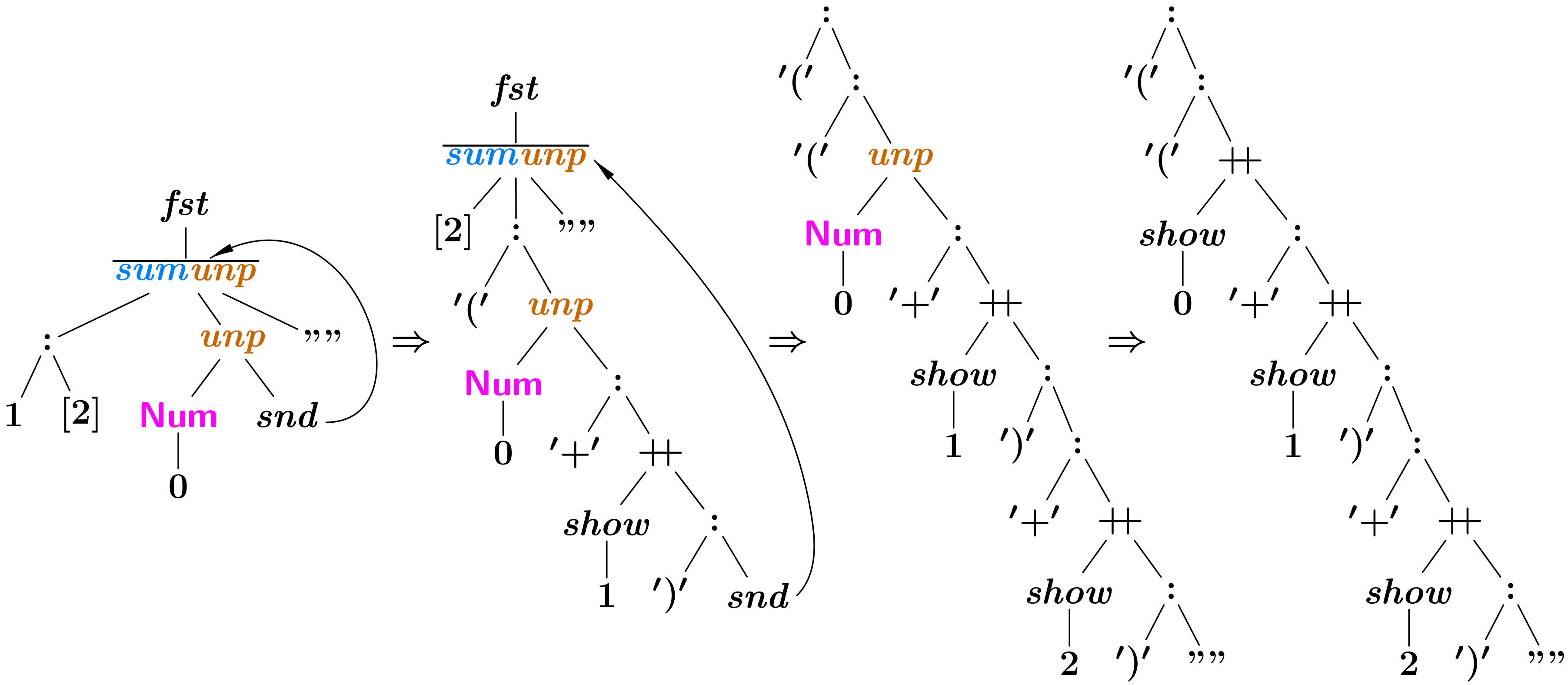
Evaluation of Transformed Program

$$
\begin{array}{llll}
\overline{sum\,unp} :: [\mathsf{Int}] \to \mathsf{String} \to \mathsf{String} \to (\mathsf{String}, \mathsf{String}) \\
\overline{sum\,unp}\quad [x] \quad y'\ z & = ('(' : y'\,,\ '+' : show\ x +\!\!+ ')' : z) \\
\overline{sum\,unp}\ (x : xs)\ y'\ z & = (fst\ v\,,\ '+' : show\ x +\!\!+ ')' : snd\ v) \\
 & \quad \text{where } v = \overline{sum\,unp}\ xs\ ('(' : y')\ z
\end{array}
$$



Using Optimized Tuple Updates [Gro99]

$$
\begin{aligned}
\overline{sum\,unp}\ [x]\ y'\ z &= ('(' : y' \,,\, '+' : show\ x \mathbin{+\!\!+} ')' : z) \\
\overline{sum\,unp}\ (x : xs)\ y'\ z &= (fst\ v \,,\, '+' : show\ x \mathbin{+\!\!+} ')' : snd\ v) \\
&\quad \text{where } v = \overline{sum\,unp}\ xs\ ('(' : y')\ z
\end{aligned}
$$



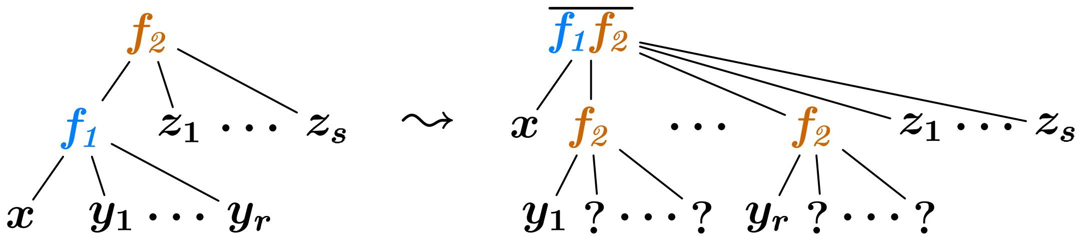
Applicability of Lazy Composition (1)

MTT-functions (cf. macro tree transducers [EV85]):

- first-order
- defined by structural recursion on one principal argument
- pattern-matching only possible on this recursion argument
- calls to external functions allowed in consumer (e.g. unp), but not in producer (e.g. sum)
- no mutual recursion (yet)

Applicability of Lazy Composition (2)

In order to ensure that the ?-values in



are always uniquely determined, and (as a consequence) that the resulting circular program terminates, the producer f_1 must be linear in its accumulating parameters and the consumer f_2 must be linear in its recursion variables.

Possible Extensions

- **mutual recursion**
- **relaxing linearity restrictions (a bit)**
- **handle external calls also in the producer (using *laws*)**
- **conditional expressions**
- ***zip*-like functions as producers**
- **. . . ?**

References

[EV85] J. Engelfriet and H. Vogler. Macro tree transducers. *J. Comput. Syst. Sci.*, 31:71–145, 1985.

[Gro99] J. Groningen. Optimising recursive functions yielding multiple results in tuples in a lazy functional language. In *Implementation of Functional Languages, Lochem, The Netherlands, Proceedings*, volume 1868 of *LNCS*, pages 59–76. Springer-Verlag, 1999.

[HJ92] G. Hamilton and S. Jones. Extending deforestation for first order functional programs. In *1991 Glasgow Workshop on Functional Programming, Portree, Scotland, Proceedings*, Series of Workshops in Computing, pages 134–145. Springer-Verlag, 1992.

[VK01] J. Voigtländer and A. Kühnemann. Composition of functions with accumulating parameters. Technical Report TUD-FI01-08, Dresden University of Technology, 2001.

[Wad90] P. Wadler. Deforestation: Transforming programs to eliminate trees. *Theoret. Comput. Sci.*, 73:231–248, 1990.