Modular Tree Automata

Deriving Modular Recursion Schemes from Tree Automata

Patrick Bahr

University of Copenhagen,
Department of Computer Science
paba@diku.dk

11th International Conference on
Mathematics of Program Construction
Madrid, Spain, June 25 - 27, 2012

Goals

Syntax-directed computations on ASTs

- program analysis
- complex program transformations
- compiler construction in general

Goals

Syntax-directed computations on ASTs

- program analysis
- complex program transformations
- compiler construction in general

Desired properties

- extensibility
- modularity
- reusability
- build complex programs by combining simple ones

Goals

Syntax-directed computations on ASTs

- program analysis
- complex program transformations
- compiler construction in general

Desired properties

- extensibility
- modularity
- reusability
- build complex programs by combining simple ones

Embed the solution into Haskell.

How do we achieve these goals?

How do we achieve these goals?

Locality

simple syntax-directed functions are local in nature

How do we achieve these goals?

Locality

simple syntax-directed functions are local in nature

Compositionality

syntax-directed functions can be combined and composed

How do we achieve these goals?

Locality

simple syntax-directed functions are local in nature

Compositionality

syntax-directed functions can be combined and composed

Contextuality

syntax-directed functions may depend on (the result of) others

How do we achieve these goals?

Locality

simple syntax-directed functions are local in nature

Compositionality

syntax-directed functions can be combined and composed

Contextuality

syntax-directed functions may depend on (the result of) others

- NB: This breaks locality and has to be carefully restricted!
- But it is convenient/necessary for
  - compositionality
  - expressivity

Locality

Tree automata

- Computation according to a set of rules.
- Applicability of rules depend only on "local" information.
- The effect of a rule application is locally restricted.

Locality

Tree automata

- Computation according to a set of rules.
- Applicability of rules depend only on “local” information.
- The effect of a rule application is locally restricted.

$$f(x_1, x_2, \ldots, x_n) \longrightarrow t[x_1, x_2, \ldots, x_n]$$

Locality

Tree automata

- Computation according to a set of rules.
- Applicability of rules depend only on "local" information.
- The effect of a rule application is locally restricted.

$$f(q_1(x_1), q_2(x_2), \ldots, q_n(x_n)) \longrightarrow q(t[x_1, x_2, \ldots, x_n])$$

Compositionality

We shall compose tree automata along 3 different dimensions.

Compositionality

We shall compose tree automata along 3 different dimensions.

sequential composition: a.k.a. deforestation

$$\mu\mathcal{F}_1 \xrightarrow{[\![\mathcal{A}_1]\!]} \mu\mathcal{F}_2 \xrightarrow{[\![\mathcal{A}_2]\!]} \mu\mathcal{F}_3$$

Compositionality

We shall compose tree automata along 3 different dimensions.

sequential composition: a.k.a. deforestation

$$
\begin{tikzcd}
\mu\mathcal{F}_1 \arrow[r, "[\![\mathcal{A}_1]\!]"] \arrow[rr, bend right, "[\![\mathcal{A}_1 \circ \mathcal{A}_2]\!]"] & \mu\mathcal{F}_2 \arrow[r, "[\![\mathcal{A}_2]\!]"] & \mu\mathcal{F}_3
\end{tikzcd}
$$

Compositionality

We shall compose tree automata along 3 different dimensions.

sequential composition: a.k.a. deforestation

$$\begin{tikzcd} \mu\mathcal{F}_1 \arrow[r, "[\![\mathcal{A}_1]\!]"] \arrow[rr, bend right, "[\![\mathcal{A}_1 \circ \mathcal{A}_2]\!]"'] & \mu\mathcal{F}_2 \arrow[r, "[\![\mathcal{A}_2]\!]"] & \mu\mathcal{F}_3 \end{tikzcd}$$

input signature: the type of the AST

$[\![\mathcal{A}_1]\!] : \mu\mathcal{F} \to R$

$[\![\mathcal{A}_2]\!] : \mu\mathcal{G} \to R$

Compositionality

We shall compose tree automata along 3 different dimensions.

sequential composition: a.k.a. deforestation

$$
\begin{tikzcd}
\mu\mathcal{F}_1 \arrow[r, "[\![\mathcal{A}_1]\!]"] \arrow[rr, bend right, "[\![\mathcal{A}_1 \circ \mathcal{A}_2]\!]"'] & \mu\mathcal{F}_2 \arrow[r, "[\![\mathcal{A}_2]\!]"] & \mu\mathcal{F}_3
\end{tikzcd}
$$

input signature: the type of the AST

$$
\begin{array}{l} [\![\mathcal{A}_1]\!] : \mu\mathcal{F} \to R \\ [\![\mathcal{A}_2]\!] : \mu\mathcal{G} \to R \end{array} \quad \Longrightarrow \quad [\![\mathcal{A}_1 + \mathcal{A}_2]\!] : \mu(\mathcal{F} + \mathcal{G}) \to R
$$

Compositionality

We shall compose tree automata along 3 different dimensions.

sequential composition: a.k.a. deforestation

$$\begin{tikzcd} \mu\mathcal{F}_1 \arrow[r, "[\![\mathcal{A}_1]\!]"] \arrow[rr, bend right, "[\![\mathcal{A}_1 \circ \mathcal{A}_2]\!]"'] & \mu\mathcal{F}_2 \arrow[r, "[\![\mathcal{A}_2]\!]"] & \mu\mathcal{F}_3 \end{tikzcd}$$

input signature: the type of the AST

$$\begin{array}{l} [\![\mathcal{A}_1]\!] : \mu\mathcal{F} \to R \\ [\![\mathcal{A}_2]\!] : \mu\mathcal{G} \to R \end{array} \quad \Longrightarrow \quad [\![\mathcal{A}_1 + \mathcal{A}_2]\!] : \mu(\mathcal{F} + \mathcal{G}) \to R$$

output type: tupling / product automaton construction

$$[\![\mathcal{A}_1]\!] : \mu\mathcal{F} \to R$$
$$[\![\mathcal{A}_2]\!] : \mu\mathcal{F} \to S$$

Compositionality

We shall compose tree automata along 3 different dimensions.

sequential composition: a.k.a. deforestation

$$\begin{tikzcd} \mu\mathcal{F}_1 \arrow[r, "[\![\mathcal{A}_1]\!]"] \arrow[rr, bend right, "[\![\mathcal{A}_1 \circ \mathcal{A}_2]\!]"'] & \mu\mathcal{F}_2 \arrow[r, "[\![\mathcal{A}_2]\!]"] & \mu\mathcal{F}_3 \end{tikzcd}$$

input signature: the type of the AST

$$\begin{array}{l} [\![\mathcal{A}_1]\!] : \mu\mathcal{F} \to R \\ [\![\mathcal{A}_2]\!] : \mu\mathcal{G} \to R \end{array} \quad \Longrightarrow \quad [\![\mathcal{A}_1 + \mathcal{A}_2]\!] : \mu(\mathcal{F} + \mathcal{G}) \to R$$

output type: tupling / product automaton construction

$$\begin{array}{l} [\![\mathcal{A}_1]\!] : \mu\mathcal{F} \to R \\ [\![\mathcal{A}_2]\!] : \mu\mathcal{F} \to S \end{array} \quad \Longrightarrow \quad [\![\mathcal{A}_1 \times \mathcal{A}_2]\!] : \mu\mathcal{F} \to R \times S$$

Contextuality

tupling / product automaton construction

$$\begin{array}{l} [\![\mathcal{A}_1]\!] : \mu\mathcal{F} \to R \\ [\![\mathcal{A}_2]\!] : \mu\mathcal{F} \to S \end{array} \quad \Longrightarrow \quad [\![\mathcal{A}_1 \times \mathcal{A}_2]\!] : \mu(\mathcal{F}) \to R \times S$$

Contextuality

tupling / product automaton construction

$$\begin{aligned} \mathcal{A}_1 &: \mathcal{F} \to R \\ \mathcal{A}_2 &: \mathcal{F} \to S \end{aligned} \quad \Longrightarrow \quad \mathcal{A}_1 \times \mathcal{A}_2 : \mathcal{F} \to R \times S$$

Contextuality

tupling / product automaton construction

$$\begin{aligned} \mathcal{A}_1 &: \mathcal{F} \to R \\ \mathcal{A}_2 &: \mathcal{F} \to S \end{aligned} \quad \Longrightarrow \quad \mathcal{A}_1 \times \mathcal{A}_2 : \mathcal{F} \to R \times S$$

mutumorphisms / dependent product automata

$$\begin{aligned} \mathcal{A}_1 &: \mathcal{F} \to R \\ \mathcal{A}_2 &: R \Rightarrow \mathcal{F} \to S \end{aligned}$$

Contextuality

tupling / product automaton construction

$$\begin{array}{lcl} \mathcal{A}_1 & : & \mathcal{F} \to R \\ \mathcal{A}_2 & : & \mathcal{F} \to S \end{array} \quad \Longrightarrow \quad \mathcal{A}_1 \times \mathcal{A}_2 \; : \; \mathcal{F} \to R \times S$$

mutumorphisms / dependent product automata

$$\begin{array}{l} \mathcal{A}_1 : \qquad\quad \mathcal{F} \to R \\ \mathcal{A}_2 : R \Rightarrow \mathcal{F} \to S \end{array} \quad \Longrightarrow \quad \mathcal{A}_1 \times \mathcal{A}_2 : \mathcal{F} \to R \times S$$

Contextuality

tupling / product automaton construction

$$\begin{aligned} \mathcal{A}_1 &: \mathcal{F} \to R \\ \mathcal{A}_2 &: \mathcal{F} \to S \end{aligned} \quad \Longrightarrow \quad \mathcal{A}_1 \times \mathcal{A}_2 : \mathcal{F} \to R \times S$$

mutumorphisms / dependent product automata

$$\begin{aligned} \mathcal{A}_1 &: S \Rightarrow \mathcal{F} \to R \\ \mathcal{A}_2 &: R \Rightarrow \mathcal{F} \to S \end{aligned} \quad \Longrightarrow \quad \mathcal{A}_1 \times \mathcal{A}_2 : \mathcal{F} \to R \times S$$

Outline

Terms in Haskell

Data types as fixed points of functors

$\textbf{data}\ \mathit{Term}\ f = \mathit{In}\ (f\ (\mathit{Term}\ f))$

Terms in Haskell

Data types as fixed points of functors

data $\mathit{Term}\ f = \mathit{In}\ (f\ (\mathit{Term}\ f))$

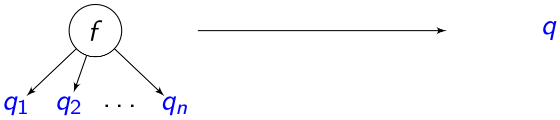
Functors

class $\mathit{Functor}\ f$ **where**
  $\mathit{fmap} :: (a \to b) \to f\ a \to f\ b$

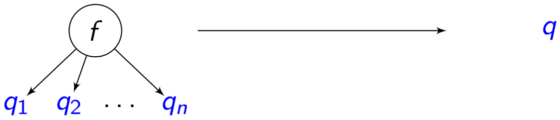
Bottom-Up State Transitions in Haskell

Bottom-Up State Transitions in Haskell

Bottom-Up State Transitions in Haskell



Bottom-Up State Transitions in Haskell



Bottom-up state transition rules as algebras

type $\mathit{UpState}\ f\ q = f\ q \to q$

Bottom-Up State Transitions in Haskell

Bottom-up state transition rules as algebras

type $UpState\ f\ q = f\ q \to q$

$$runUpState :: Functor\ f \Rightarrow UpState\ f\ q \to Term\ f \to q$$
$$runUpState\ \phi\ (In\ t) = \phi\ (fmap\ (runUpState\ \phi)\ t)$$

Bottom-Up State Transitions in Haskell

Bottom-up s a.k.a. catamorphism / fold bras

type $UpState\ f\ q = f\ q \rightarrow q$

$runUpState :: Functor\ f \Rightarrow UpState\ f\ q \rightarrow Term\ f \rightarrow q$
$runUpState\ \phi\ (In\ t) = \phi\ (fmap\ (runUpState\ \phi)\ t)$

Composing State Spaces – Motivating Example

A simple expression language

$\textbf{data}\ \mathit{Sig}\ e = \mathit{Val}\ \mathit{Int} \mid \mathit{Plus}\ e\ e$

Composing State Spaces – Motivating Example

A simple expression language

data $\mathit{Sig}\ e = \mathit{Val}\ \mathit{Int} \mid \mathit{Plus}\ e\ e$

Task: writing a code generator

type $\mathit{Addr} = \mathit{Int}$
data $\mathit{Instr} = \mathit{Acc}\ \mathit{Int} \mid \mathit{Load}\ \mathit{Addr} \mid \mathit{Store}\ \mathit{Addr} \mid \mathit{Add}\ \mathit{Addr}$
type $\mathit{Code} = [\mathit{Instr}]$

Composing State Spaces – Motivating Example

A simple expression language

data $\mathit{Sig}\ e = \mathit{Val}\ \mathit{Int} \mid \mathit{Plus}\ e\ e$

Task: writing a code generator

type $\mathit{Addr} = \mathit{Int}$
data $\mathit{Instr} = \mathit{Acc}\ \mathit{Int} \mid \mathit{Load}\ \mathit{Addr} \mid \mathit{Store}\ \mathit{Addr} \mid \mathit{Add}\ \mathit{Addr}$
type $\mathit{Code} = [\mathit{Instr}]$

The problem

$\mathit{codeSt} :: \mathit{UpState}\ \mathit{Sig}\ \mathit{Code}$
$\mathit{codeSt}\ (\mathit{Val}\ i) = [\mathit{Acc}\ i]$
$\mathit{codeSt}\ (\mathit{Plus}\ x\ y) = x +\!\!+ [\mathit{Store}\ a] +\!\!+ y +\!\!+ [\mathit{Add}\ a]$
  where $a = \ldots$

Composing State Spaces – Motivating Example

A simple expression language

data $Sig\ e = Val\ Int \mid Plus\ e\ e$

Task: writing a code generator

type $Addr = Int$
data $Instr = Acc\ Int \mid Load\ Addr \mid Store\ Addr \mid Add\ Addr$
type $Code = [Instr]$

$Sig\ Code \to Code$

The problem

$codeSt :: UpState\ Sig\ Code$
$codeSt\ (Val\ i) = [Acc\ i]$
$codeSt\ (Plus\ x\ y) = x +\!\!+ [Store\ a] +\!\!+ y +\!\!+ [Add\ a]$
  where $a = \ldots$

Tupling

Tuple the code with an address counter

$$
\begin{aligned}
&\mathit{codeAddrSt} :: \mathit{UpState}\ \mathit{Sig}\ (\mathit{Code}, \mathit{Addr})\\
&\mathit{codeAddrSt}\ (\mathit{Val}\ i) &&= ([\mathit{Acc}\ i], 0)\\
&\mathit{codeAddrSt}\ (\mathit{Plus}\ (x, a')\ (y, a)) &&= (x +\!\!+ [\mathit{Store}\ a] +\!\!+ y +\!\!+ [\mathit{Add}\ a],\\
& && \qquad 1 + \mathit{max}\ a\ a')
\end{aligned}
$$

Tupling

Tuple the code with an address counter

$$
\begin{aligned}
&\mathit{codeAddrSt} :: \mathit{UpState}\ \mathit{Sig}\ (\mathit{Code}, \mathit{Addr})\\
&\mathit{codeAddrSt}\ (\mathit{Val}\ i) = ([\mathit{Acc}\ i], 0)\\
&\mathit{codeAddrSt}\ (\mathit{Plus}\ (x, a')\ (y, a)) = (x \mathbin{+\!\!+} [\mathit{Store}\ a] \mathbin{+\!\!+} y \mathbin{+\!\!+} [\mathit{Add}\ a],\\
&\qquad 1 + \mathit{max}\ a\ a')
\end{aligned}
$$

Run the automaton

$$
\begin{aligned}
&\mathit{code} :: \mathit{Term}\ \mathit{Sig} \to (\mathit{Code}, \mathit{Addr})\\
&\mathit{code} = \mathit{runUpState}\ \mathit{codeAddrSt}
\end{aligned}
$$

Tupling

Tuple the code with an address counter

$$
\begin{aligned}
&\mathit{codeAddrSt} :: \mathit{UpState}\ \mathit{Sig}\ (\mathit{Code}, \mathit{Addr}) \\
&\mathit{codeAddrSt}\ (\mathit{Val}\ i) = ([\mathit{Acc}\ i], 0) \\
&\mathit{codeAddrSt}\ (\mathit{Plus}\ (x, a')\ (y, a)) = (x +\!\!+ [\mathit{Store}\ a] +\!\!+ y +\!\!+ [\mathit{Add}\ a], \\
&\qquad 1 + \mathit{max}\ a\ a')
\end{aligned}
$$

Run the automaton

$$
\begin{aligned}
&\mathit{code} :: \mathit{Term}\ \mathit{Sig} \to (\mathit{Code}, \mathit{Addr}) \\
&\mathit{code} = \mathit{fst}\ .\ \mathit{runUpState}\ \mathit{codeAddrSt}
\end{aligned}
$$

Tupling

Tuple the code with an address counter

$$
\begin{aligned}
&\mathit{codeAddrSt} :: \mathit{UpState}\ \mathit{Sig}\ (\mathit{Code}, \mathit{Addr}) \\
&\mathit{codeAddrSt}\ (\mathit{Val}\ i) = ([\mathit{Acc}\ i], 0) \\
&\mathit{codeAddrSt}\ (\mathit{Plus}\ (x, a')\ (y, a)) = (x +\!\!+ [\mathit{Store}\ a] +\!\!+ y +\!\!+ [\mathit{Add}\ a], \\
&\qquad 1 + \mathit{max}\ a\ a')
\end{aligned}
$$

Run the automaton

$$
\begin{aligned}
&\mathit{code} :: \mathit{Term}\ \mathit{Sig} \to \mathit{Code} \\
&\mathit{code} = \mathit{fst}\ .\ \mathit{runUpState}\ \mathit{codeAddrSt}
\end{aligned}
$$

Product Automata

Deriving projections

class $a \in b$ **where**

$pr :: b \to a$

Product Automata

Deriving projections

class $a \in b$ **where**
$pr :: b \to a$

$a \in b$ iff

- b is of the form $(b_1, (b_2, \ldots b_n))$ and
- $a = b_i$ for some i

Product Automata

Deriving projections

class $a \in b$ **where**
  $pr :: b \to a$

$a \in b$ iff

- b is of the form $(b_1, (b_2, \ldots b_n))$ and
- $a = b_i$ for some i

For example: $Addr \in (Code, Addr)$

Product Automata

Deriving projections

class $a \in b$ **where**
$pr :: b \to a$

$a \in b$ iff

- b is of the form $(b_1, (b_2, \dots b_n))$ and
- $a = b_i$ for some i

For example: $Addr \in (Code, Addr)$

Dependent state transition functions

type $UpState \quad f \quad q = \qquad f\ q \to q$

Product Automata

Deriving projections

class $a \in b$ **where**
  $pr :: b \to a$

$a \in b$ iff

- b is of the form $(b_1, (b_2, \dots b_n))$ and
- $a = b_i$ for some i

For example: $Addr \in (Code, Addr)$

Dependent state transition functions

type $UpState\ f\ \ q = f\ q \to q$
type $DUpState\ f\ p\ q = (q \in p) \Rightarrow f\ p \to q$

Product Automata

Deriving projections

$$\textbf{class } a \in b \textbf{ where} \quad pr :: b \to a$$

$a \in b$ iff

- b is of the form $(b_1, (b_2, \ldots b_n))$ and
- $a = b_i$ for some i

For example: $Addr \in (Code, Addr)$

Dependent state transition functions

$$\textbf{type } UpState\ f\ q = f\ q \to q$$
$$\textbf{type } DUpState\ f\ p\ q = (q \in p) \Rightarrow f\ p \to q$$

Product Automata

Deriving projections

class $a \in b$ **where**
  $pr :: b \to a$

$a \in b$ iff

- b is of the form $(b_1, (b_2, \ldots b_n))$ and
- $a = b_i$ for some i

For example: $Addr \in (Code, Addr)$

Dependent state transition functions

type $UpState\ f\ q = f\ q \to q$
type $DUpState\ f\ p\ q = (q \in p) \Rightarrow f\ p \to q$

Product Automata

Deriving projections

class $a \in b$ **where**
$\quad pr :: b \to a$

$a \in b$ iff

- b is of the form $(b_1, (b_2, \ldots b_n))$ and
- $a = b_i$ for some i

For example: $Addr \in (Code, Addr)$

Dependent state transition functions

$$
\begin{aligned}
&\textbf{type } UpState\ f\ \ q = f\ q \to q \\
&\textbf{type } DUpState\ f\ p\ q = (q \in p) \Rightarrow f\ p \to q
\end{aligned}
$$

Product state transition

$$
\begin{aligned}
(\otimes) :: (p \in c, q \in c) &\Rightarrow DUpState\ f\ c\ p \to DUpState\ f\ c\ q \\
&\to DUpState\ f\ c\ (p, q)
\end{aligned}
$$

$$(sp \otimes sq)\ t = (sp\ t, sq\ t)$$

Running Dependent State Transition Functions

The types

$$\textbf{type}\ \mathit{UpState}\ f\ q = f\ q \rightarrow q$$
$$\textbf{type}\ \mathit{DUpState}\ f\ p\ q = (q \in p) \Rightarrow f\ p \rightarrow q$$

Running Dependent State Transition Functions

The types

$$\textbf{type}\ \mathit{UpState}\ \ f\ q\ \ = \qquad\qquad\quad f\ q \to q$$
$$\textbf{type}\ \mathit{DUpState}\ f\ p\ q = (q \in p) \Rightarrow f\ p \to q$$

Running dependent state transitions

$$\mathit{runDUpState} :: \mathit{Functor}\ f \Rightarrow \mathit{DUpState}\ f\ q\ q \to \mathit{Term}\ f \to q$$
$$\mathit{runDUpState}\ f = \mathit{runUpState}\ f$$

Running Dependent State Transition Functions

The types

$$\textbf{type}\ \mathit{UpState}\ \ f\ q\ \ = \qquad\qquad f\ q \to q$$
$$\textbf{type}\ \mathit{DUpState}\ f\ p\ q = (q \in p) \Rightarrow f\ p \to q$$

Running dependent state transitions

$$\mathit{runDUpState} :: \mathit{Functor}\ f \Rightarrow \mathit{DUpState}\ f\ q\ q \to \mathit{Term}\ f \to q$$
$$\mathit{runDUpState}\ f = \mathit{runUpState}\ f$$

From state transition to dependent state transition

$$\mathit{dUpState} :: \mathit{Functor}\ f \Rightarrow \mathit{UpState}\ f\ q \to \mathit{DUpState}\ f\ p\ q$$
$$\mathit{dUpState}\ st = st\ .\ \mathit{fmap}\ pr$$

The Code Generator Example

The code generator

$$
\begin{aligned}
&\mathit{codeSt} :: (\mathit{Int} \in q) \Rightarrow \mathit{DUpState}\ \mathit{Sig}\ q\ \mathit{Code}\\
&\mathit{codeSt}\ (\mathit{Val}\ i) \qquad = [\mathit{Acc}\ i]\\
&\mathit{codeSt}\ (\mathit{Plus}\ x\ y) = \mathit{pr}\ x +\!\!+ [\mathit{Store}\ a] +\!\!+ \mathit{pr}\ y +\!\!+ [\mathit{Add}\ a]\\
&\quad \textbf{where}\ a = \mathit{pr}\ y
\end{aligned}
$$

The Code Generator Example

The code generator

$$\begin{aligned}
&\mathit{codeSt} :: (\mathit{Int} \in q) \Rightarrow \mathit{DUpState}\ \mathit{Sig}\ q\ \mathit{Code}\\
&\mathit{codeSt}\ (\mathit{Val}\ i) \qquad = [\mathit{Acc}\ i]\\
&\mathit{codeSt}\ (\mathit{Plus}\ x\ y) = \mathit{pr}\ x +\!\!+ [\mathit{Store}\ a] +\!\!+ \mathit{pr}\ y +\!\!+ [\mathit{Add}\ a]\\
&\quad \mathbf{where}\ a = \mathit{pr}\ y
\end{aligned}$$

Generating fresh addresses

$$\begin{aligned}
&\mathit{heightSt} :: \mathit{UpState}\ \mathit{Sig}\ \mathit{Int}\\
&\mathit{heightSt}\ (\mathit{Val}\ _) \qquad = 0\\
&\mathit{heightSt}\ (\mathit{Plus}\ x\ y) = 1 + \mathit{max}\ x\ y
\end{aligned}$$

The Code Generator Example

The code generator

$$
\begin{aligned}
&\mathit{codeSt} :: (\mathit{Int} \in q) \Rightarrow \mathit{DUpState}\ \mathit{Sig}\ q\ \mathit{Code} \\
&\mathit{codeSt}\ (\mathit{Val}\ i) \quad\ \ = [\mathit{Acc}\ i] \\
&\mathit{codeSt}\ (\mathit{Plus}\ x\ y) = \mathit{pr}\ x +\!\!+ [\mathit{Store}\ a] +\!\!+ \mathit{pr}\ y +\!\!+ [\mathit{Add}\ a] \\
&\quad \mathbf{where}\ a = \mathit{pr}\ y
\end{aligned}
$$

Generating fresh addresses

$$
\begin{aligned}
&\mathit{heightSt} :: \mathit{UpState}\ \mathit{Sig}\ \mathit{Int} \\
&\mathit{heightSt}\ (\mathit{Val}\ \_) \quad\ \ = 0 \\
&\mathit{heightSt}\ (\mathit{Plus}\ x\ y) = 1 + \mathit{max}\ x\ y
\end{aligned}
$$

Combining the components

$$
\begin{aligned}
&\mathit{code} :: \mathit{Term}\ \mathit{Sig} \rightarrow \mathit{Code} \\
&\mathit{code} = \mathit{fst}\ .\ \mathit{runUpState}\ (\mathit{codeSt} \otimes \mathit{dUpState}\ \mathit{heightSt})
\end{aligned}
$$

The Code Generator Example

The code generator

$$\begin{aligned}
&\mathit{codeSt} :: (\mathit{Int} \in q) \Rightarrow \mathit{DUpState}\ \mathit{Sig}\ q\ \mathit{Code}\\
&\mathit{codeSt}\ (\mathit{Val}\ i) \qquad = [\mathit{Acc}\ i]\\
&\mathit{codeSt}\ (\mathit{Plus}\ x\ y) = \mathit{pr}\ x +\!\!+ [\mathit{Store}\ a] +\!\!+ \mathit{pr}\ y +\!\!+ [\mathit{Add}\ a]\\
&\quad \textbf{where}\ a = \mathit{pr}\ y
\end{aligned}$$

Generating fresh addresses

$$\begin{aligned}
&\mathit{heightSt} :: \mathit{UpState}\ \mathit{Sig}\ \mathit{Int}\\
&\mathit{heightSt}\ (\mathit{Val}\ _) \qquad = 0\\
&\mathit{heightSt}\ (\mathit{Plu}
\end{aligned}$$

Combining the components

$$\begin{aligned}
&\mathit{code} :: \mathit{Term}\ \mathit{Sig} \rightarrow \mathit{Code}\\
&\mathit{code} = \mathit{fst}\ .\ \mathit{runUpState}\ \overbrace{(\mathit{codeSt} \otimes \mathit{dUpState}\ \mathit{heightSt})}^{(\mathit{Code} \in q,\, \mathit{Int} \in q) \Rightarrow \mathit{DUpState}\ \mathit{Sig}\ q\ (\mathit{Code},\, \mathit{Int})}
\end{aligned}$$

Outline

Combining Signatures

Signatures & automata may be combined in the style of “Data types à la carte” [Swierstra 2008].

Combining Signatures

Signatures & automata may be combined in the style of "Data types à la carte" [Swierstra 2008].

Coproduct of signatures

$$\mathbf{data}\ (f \oplus g)\ e = \mathit{Inl}\ (f\ \ e) \mid \mathit{Inr}\ (g\ \ e)$$

Combining Signatures

Signatures & automata may be combined in the style of "Data types à la carte" [Swierstra 2008].

Coproduct of signatures

$$\textbf{data}\ (f \oplus g)\ e = \mathit{Inl}\ (f\ \ e) \mid \mathit{Inr}\ (g\ \ e)$$

Example

$$\begin{aligned}&\textbf{data}\ \mathit{Inc}\ e = \mathit{Inc}\ e\\ &\textbf{type}\ \mathit{Sig}' \quad = \mathit{Inc} \oplus \mathit{Sig}\end{aligned}$$

Combining Signatures

Signatures & automata may be combined in the style of "Data types à la carte" [Swierstra 2008].

Coproduct of signatures

$$\textbf{data}\ (f \oplus g)\ e = \mathit{Inl}\ (f\ \ e) \mid \mathit{Inr}\ (g\ \ e)$$

Example

$$\begin{aligned}&\textbf{data}\ \mathit{Inc}\ e = \mathit{Inc}\ e\\ &\textbf{type}\ \mathit{Sig}' \ \ = \mathit{Inc} \oplus \mathit{Sig}\end{aligned}$$

Subsignature type class

$$\begin{aligned}&\textbf{class}\ f \preceq g\ \textbf{where}\\ &\quad \mathit{inj} :: f\ \ a \to g\ \ a\end{aligned}$$

Combining Signatures

Signatures & automata may be combined in the style of "Data types à la carte" [Swierstra 2008].

Coproduct of signatures

data $(f \oplus g)\ e = Inl\ (f\ \ e) \mid Inr\ (g\ \ e)$

Example

data $Inc\ e = Inc\ e$
type $Sig' \quad = Inc \oplus Sig$

Subsignature type class

class $f \preceq g$ **where**
$\quad inj :: f\ \ a \rightarrow g\ \ a$

$f \preceq g$ iff

- $g = g_1 \oplus g_2 \oplus \ldots \oplus g_n$ and
- $f = g_i, \quad 0 < i \leq n$

Combining Signatures

Signatures & automata may be combined in the style of "Data types à la carte" [Swierstra 2008].

Coproduct of signatures

data $(f \oplus g)\ e = \mathit{Inl}\ (f\ \ e) \mid \mathit{Inr}\ (g\ \ e)$

Example

data $\mathit{Inc}\ e = \mathit{Inc}\ e$
type $\mathit{Sig}' \ \ = \mathit{Inc} \oplus \mathit{Sig}$

Subsignature type class

class $f \preceq g$ **where**
$\quad \mathit{inj} :: f\ \ a \rightarrow g\ \ a$

For example: $\mathit{Inc} \preceq \mathit{Sig}'$

$f \preceq g$ iff

- $g = g_1 \oplus g_2 \oplus \ldots \oplus g_n$ and
- $f = g_i, \quad 0 < i \leq n$

Combining Automata

Making the height compositional

class *HeightSt f* **where**
  $heightSt :: DUpState\ f\ q\ Int$

instance $(HeightSt\ f, HeightSt\ g) \Rightarrow HeightSt\ (f \oplus g)$ **where**
  $heightSt\ (Inl\ x) = heightSt\ x$
  $heightSt\ (Inr\ x) = heightSt\ x$

Combining Automata

Making the height compositional

$$\textbf{class}\ \mathit{HeightSt}\ f\ \textbf{where}$$
$$\quad \mathit{heightSt} :: \mathit{DUpState}\ f\ q\ \mathit{Int}$$

$$\textbf{instance}\ (\mathit{HeightSt}\ f, \mathit{HeightSt}\ g) \Rightarrow \mathit{HeightSt}\ (f \oplus g)\ \textbf{where}$$
$$\quad \mathit{heightSt}\ (\mathit{Inl}\ x) = \mathit{heightSt}\ x$$
$$\quad \mathit{heightSt}\ (\mathit{Inr}\ x) = \mathit{heightSt}\ x$$

Defining the height on Sig

$$\textbf{instance}\ \mathit{HeightSt}\ \mathit{Sig}\ \textbf{where}$$
$$\quad \mathit{heightSt}\ (\mathit{Val}\ \_) \qquad = 0$$
$$\quad \mathit{heightSt}\ (\mathit{Plus}\ x\ y) = 1 + \mathit{max}\ x\ y$$

Combining Automata

Making the height compositional

$$
\begin{array}{l}
\textbf{class } \mathit{HeightSt}\ f\ \textbf{where} \\
\quad \mathit{heightSt} :: \mathit{DUpState}\ f\ q\ \mathit{Int}
\end{array}
$$

$$
\begin{array}{l}
\textbf{instance } (\mathit{HeightSt}\ f, \mathit{HeightSt}\ g) \Rightarrow \mathit{HeightSt}\ (f \oplus g)\ \textbf{where} \\
\quad \mathit{heightSt}\ (\mathit{Inl}\ x) = \mathit{heightSt}\ x \\
\quad \mathit{heightSt}\ (\mathit{Inr}\ x) = \mathit{heightSt}\ x
\end{array}
$$

Defining the height on Sig

$$
\begin{array}{l}
\textbf{instance } \mathit{HeightSt}\ \mathit{Sig}\ \textbf{where} \\
\quad \mathit{heightSt}\ (\mathit{Val}\ \_) \quad\ \ = 0 \\
\quad \mathit{heightSt}\ (\mathit{Plus}\ x\ y) = 1 + \mathit{max}\ x\ y
\end{array}
$$

Defining the height on Inc

$$
\begin{array}{l}
\textbf{instance } \mathit{HeightSt}\ \mathit{Inc}\ \textbf{where} \\
\quad \mathit{heightSt}\ (\mathit{Inc}\ x) = 1 + x
\end{array}
$$

Outline

Bottom-Up Tree Transducers

Bottom-Up Tree Transducers

Bottom-Up Tree Transducers

From terms to contexts

data $\mathit{Term}\ f = \mathit{In}\ (f\ (\mathit{Term}\ f\ ))$

data $\mathit{Context}\ f\ a = \mathit{In}\ (f\ (\mathit{Context}\ f\ a))\ |\ \mathit{Hole}\ a$

Bottom-Up Tree Transducers

type *Term f* = *Context f Empty*

From terms to contexts

data *Term f* = *In* (*f* (*Term f*))

data *Context f a* = *In* (*f* (*Context f a*)) | *Hole a*

Bottom-Up Tree Transducers

From terms to contexts

data $Term \; f \quad = In\,(f\,(Term \quad f \;))$

data $Context\; f\; a = In\,(f\,(Context\; f\; a)) \mid Hole\; a$

Representing transduction rules, [Hasuo et al. 2007]

type $UpTrans\; f\; q\; g = \forall\, a.f\,(q,a) \to (q,\, Context\; g\; a)$

Bottom-Up Tree Transducers

From terms to contexts

data $Term\ f = In\ (f\ (Term\ f))$

data $Context\ f\ a = In\ (f\ (Context\ f\ a))\ |\ Hole\ a$

Representing transduction rules, [Hasuo et al. 2007]

type $UpTrans\ f\ q\ g = \forall\ a.f\ (q,a) \rightarrow (q,\ Context\ g\ a)$

Tree Homomorphisms

type $UpTrans\ f\ q\ g = \forall\, a\,.\, f\ (q, a) \rightarrow (q, Context\ g\ a)$

Tree Homomorphisms

$$\textbf{type}\ \mathit{UpTrans}\ f\quad g = \forall\, a\,.\, f\qquad a\ \ \rightarrow\qquad \mathit{Context}\ g\ a$$

Tree Homomorphisms

$$\textbf{type}\ Hom\ f\ g = \forall a\,.\, f\ a \to Context\ g\ a$$

Tree Homomorphisms

$$\textbf{type } Hom \quad f \quad g = \forall\, a \,.\, f \quad a \;\rightarrow \quad Context\ g\ a$$

Example (Desugaring)

class *DesugHom f g* **where**
  desugHom :: *Hom f g*

$desugar :: (Functor\ f, Functor\ g, DesugHom\ f\ g) \Rightarrow Term\ f \rightarrow Term\ g$
$desugar = runHom\ desugHom$

Tree Homomorphisms

type $\mathit{Hom}\ f\ g = \forall\, a\,.\, f\ a \to \mathit{Context}\ g\ a$

Example (Desugaring)

class $\mathit{DesugHom}\ f\ g$ **where**
$\quad \mathit{desugHom} :: \mathit{Hom}\ f\ g$

$\mathit{desugar} :: (\mathit{Functor}\ f, \mathit{Functor}\ g, \mathit{DesugHom}\ f\ g) \Rightarrow \mathit{Term}\ f \to \mathit{Term}\ g$
$\mathit{desugar} = \mathit{runHom}\ \mathit{desugHom}$

instance $(\mathit{Sig} \preceq g) \Rightarrow \mathit{DesugHom}\ \mathit{Inc}\ g$ **where**
$\quad \mathit{desugHom}\ (\mathit{Inc}\ x) = \mathit{Hole}\ x\ \text{`}\mathit{plus}\text{`}\ \mathit{val}\ 1$

instance $(\mathit{Functor}\ g, f \preceq g) \Rightarrow \mathit{DesugHom}\ f\ g$ **where**
$\quad \mathit{desugHom} = \mathit{simpCxt}\ .\ \mathit{inj}$

Tree Homomorphisms

type $Hom\ f\ g = \forall\, a\,.\, f\ a \rightarrow Context\ g\ a$

Example (Desugaring)

class $DesugHom\ f\ g$ **where**
$desugHom :: Hom\ f\ g$

$desugar :: (Functor\ f, Functor\ g, DesugHom\ f\ g) \Rightarrow Term\ f \rightarrow Term\ g$
$desugar = runHom\ desugHom$

instance $(Sig \preceq g) \Rightarrow$
$desugHom\ (Inc\ x)$

instance $(Functor\ g, f \preceq g) \Rightarrow DesugHom\ f\ g$ **where**
$desugHom = simpCxt\ .\ inj$

$simpCxt :: Functor\ g \Rightarrow g\ a \rightarrow Context\ g\ a$
$simpCxt\ t = In\ (fmap\ Hole\ t)$

Stateful Tree Homomorphisms

Decomposing tree transducers

type $\mathit{Hom}\ f\ g = \forall\, a\,.\, f\ a \rightarrow \mathit{Context}\ g\ a$
type $\mathit{UpState}\ f\ q = f\ q \rightarrow q$
type $\mathit{UpTrans}\ f\ q\ g = \forall\, a\,.\, f\ (q, a) \rightarrow (q, \mathit{Context}\ g\ a)$

Stateful Tree Homomorphisms

Decomposing tree transducers

$$\textbf{type}\ \mathit{Hom}\ f\ g = \forall\, a\,.\, f\ a \rightarrow \mathit{Context}\ g\ a$$
$$\textbf{type}\ \mathit{UpState}\ f\ q = f\ q \rightarrow q$$
$$\textbf{type}\ \mathit{UpTrans}\ f\ q\ g = \forall\, a\,.\, f\ (q, a) \rightarrow (q, \mathit{Context}\ g\ a)$$

Making homomorphisms dependent on a state

$$\textbf{type}\ \mathit{QHom}\ f\ q\ g = \forall\, a.\ f\ a \rightarrow \mathit{Context}\ g\ a$$

Stateful Tree Homomorphisms

Decomposing tree transducers

type $\mathit{Hom}\ f\ g = \forall\, a\, .\, f\ a \to \mathit{Context}\ g\ a$
type $\mathit{UpState}\ f\ q = f\ q \to q$
type $\mathit{UpTrans}\ f\ q\ g = \forall\, a\, .\, f\ (q, a) \to (q, \mathit{Context}\ g\ a)$

Making homomorphisms dependent on a state

type $\mathit{QHom}\ f\ q\ g = \forall\, a.\ f(q,\ a) \to \mathit{Context}\ g\ a$

Stateful Tree Homomorphisms

Decomposing tree transducers

$$\begin{aligned}
&\textbf{type } \mathit{Hom}\ f\ g = \forall\, a\,.\, f\ a \to \mathit{Context}\ g\ a \\
&\textbf{type } \mathit{UpState}\ f\ q = f\ q \to q \\
&\textbf{type } \mathit{UpTrans}\ f\ q\ g = \forall\, a\,.\, f\ (q, a) \to (q, \mathit{Context}\ g\ a)
\end{aligned}$$

Making homomorphisms dependent on a state

$$\textbf{type } \mathit{QHom}\ f\ q\ g = \forall\, a.\ (a \to q) \to f\ a \to \mathit{Context}\ g\ a$$

Stateful Tree Homomorphisms

Decomposing tree transducers

$$\begin{array}{l}
\textbf{type}\ \mathit{Hom}\ f\ g = \forall\, a\,.\, f\ a \rightarrow \mathit{Context}\ g\ a \\
\textbf{type}\ \mathit{UpState}\ f\ q = f\ q \rightarrow q \\
\textbf{type}\ \mathit{UpTrans}\ f\ q\ g = \forall\, a\,.\, f\ (q, a) \rightarrow (q, \mathit{Context}\ g\ a)
\end{array}$$

Making homomorphisms dependent on a state

$$\textbf{type}\ \mathit{QHom}\ f\ q\ g = \forall\, a.\ (a \rightarrow q) \rightarrow f\ a \rightarrow \mathit{Context}\ g\ a$$

From stateful homomorphisms to tree transducers

$$\begin{array}{l}
\mathit{upTrans} :: (\mathit{Functor}\ f, \mathit{Functor}\ g) \Rightarrow \\
\qquad \mathit{UpState}\ f\ q \rightarrow \mathit{QHom}\ f\ q\ g \rightarrow \mathit{UpTrans}\ f\ q\ g \\
\mathit{upTrans}\ \mathit{st}\ \mathit{hom}\ t = (q, c)\ \textbf{where} \\
\quad q = \mathit{st}\ (\mathit{fmap}\ \mathit{fst}\ t) \\
\quad c = \mathit{fmap}\ \mathit{snd}\ (\mathit{hom}\ \mathit{fst}\ t)
\end{array}$$

An Example

Extending the signature with let bindings

$$\begin{aligned}
&\textbf{type}\ \mathit{Name} &&= \mathit{String}\\
&\textbf{data}\ \mathit{Let}\ e &&= \mathit{LetIn}\ \mathit{Name}\ e\ e \mid \mathit{Var}\ \mathit{Name}\\
&\textbf{type}\ \mathit{LetSig} &&= \mathit{Let} \oplus \mathit{Sig}
\end{aligned}$$

An Example

Extending the signature with let bindings

$$
\begin{aligned}
&\textbf{type}\ \mathit{Name} &&= \mathit{String}\\
&\textbf{data}\ \mathit{Let}\ e &&= \mathit{LetIn}\ \mathit{Name}\ e\ e \mid \mathit{Var}\ \mathit{Name}\\
&\textbf{type}\ \mathit{LetSig} &&= \mathit{Let} \oplus \mathit{Sig}
\end{aligned}
$$

$\textbf{type}\ \mathit{Vars} = \mathit{Set}\ \mathit{Name}$

$\textbf{class}\ \mathit{FreeVarsSt}\ f\ \textbf{where}$
$\quad \mathit{freeVarsSt} :: \mathit{UpState}\ f\ \mathit{Vars}$

An Example

Extending the signature with let bindings

$$
\begin{aligned}
&\textbf{type}\ \mathit{Name} &&= \mathit{String}\\
&\textbf{data}\ \mathit{Let}\ e &&= \mathit{LetIn}\ \mathit{Name}\ e\ e \mid \mathit{Var}\ \mathit{Name}\\
&\textbf{type}\ \mathit{LetSig} &&= \mathit{Let} \oplus \mathit{Sig}
\end{aligned}
$$

$$
\begin{aligned}
&\textbf{type}\ \mathit{Vars} = \mathit{Set}\ \mathit{Name}\\
&\textbf{class}\ \mathit{FreeVarsSt}\ f\ \textbf{where}\\
&\quad \mathit{freeVarsSt} :: \mathit{UpState}\ f\ \mathit{Vars}\\
&\textbf{instance}\ \mathit{FreeVarsSt}\ \mathit{Sig}\ \textbf{where}\\
&\quad \mathit{freeVarsSt}\ (\mathit{Plus}\ x\ y) = x\ `\mathit{union}`\ y\\
&\quad \mathit{freeVarsSt}\ (\mathit{Val}\ \_) \quad = \mathit{empty}\\
&\textbf{instance}\ \mathit{FreeVarsSt}\ \mathit{Let}\ \textbf{where}\\
&\quad \mathit{freeVarsSt}\ (\mathit{Var}\ v) \qquad = \mathit{singleton}\ v\\
&\quad \mathit{freeVarsSt}\ (\mathit{LetIn}\ v\ e\ s) = \textbf{if}\ v\ `\mathit{member}`\ s\ \textbf{then}\ e\ `\mathit{union}`\ \mathit{delete}\ v\ s\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{else}\ \ s
\end{aligned}
$$

An Example (Cont'd)

$$\begin{array}{l}
\textbf{class}\ \mathit{RemLetHom}\ f\ \ q\ g\ \textbf{where} \\
\quad \mathit{remLetHom} :: \mathit{QHom}\ f\ \ q\ g \\
\textbf{instance}\ (\mathit{Vars} \in q, \mathit{Let} \preceq g, \mathit{Functor}\ g) \Rightarrow \mathit{RemLetHom}\ \mathit{Let}\ q\ g\ \textbf{where} \\
\quad \mathit{remLetHom}\ \mathit{qOf}\ \ (\mathit{LetIn}\ v\ _\ s) \mid \neg\ (v\ \text{`}\mathit{member}\text{`}\ \mathit{qOf}\ \ s) = \mathit{Hole}\ s \\
\quad \mathit{remLetHom}\ _\ \qquad t \qquad\qquad\qquad\qquad\qquad\qquad\qquad = \mathit{simpCxt}\ (\mathit{inj}\ \ t) \\
\textbf{instance}\ (\mathit{Functor}\ f, \mathit{Functor}\ g, f \preceq g) \Rightarrow \mathit{RemLetHom}\ f\ \ q\ g\ \textbf{where} \\
\quad \mathit{remLetHom}\ _ = \ \mathit{simpCxt} \ .\ \mathit{inj}
\end{array}$$

An Example (Cont'd)

$$
\begin{aligned}
&\textbf{class } \mathit{RemLetHom}\ f\ q\ g\ \textbf{where} \\
&\quad \mathit{remLetHom} :: \mathit{QHom}\ f\ q\ g \\
&\textbf{instance } (\mathit{Vars} \in q, \mathit{Let} \preceq g, \mathit{Functor}\ g) \Rightarrow \mathit{RemLetHom}\ \mathit{Let}\ q\ g\ \textbf{where} \\
&\quad \mathit{remLetHom}\ \mathit{qOf}\ (\mathit{LetIn}\ v\ \_\ s) \mid \neg\, (v\ \text{`}\mathit{member}\text{`}\ \mathit{qOf}\ s) = \mathit{Hole}\ s \\
&\quad \mathit{remLetHom}\ \_\ \ t = \mathit{simpCxt}\ (\mathit{inj}\ t) \\
&\textbf{instance } (\mathit{Functor}\ f, \mathit{Functor}\ g, f \preceq g) \Rightarrow \mathit{RemLetHom}\ f\ q\ g\ \textbf{where} \\
&\quad \mathit{remLetHom}\ \_ = \mathit{simpCxt} . \mathit{inj}
\end{aligned}
$$

Combining state transition and homomorphism

$$
\begin{aligned}
&\mathit{remLet} :: (\mathit{Functor}\ f, \mathit{FreeVarsSt}\ f, \mathit{RemLetHom}\ f\ \mathit{Vars}\ f) \\
&\qquad \Rightarrow \mathit{Term}\ f \to (\mathit{Vars}, \mathit{Term}\ f) \\
&\mathit{remLet} = \mathit{runUpHom}\ \ \mathit{freeVarsSt}\ \mathit{remLetHom}
\end{aligned}
$$

An Example (Cont'd)

```
class RemLetHom f q g where
  remLetHom :: QHom f q g

instance (Vars ∈ q, Let ≼ g, Functor g) ⇒ RemLetHom Let q g where
  remLetHom qOf (LetIn v _ s) | ¬ (v `member` qOf s) = Hole s
  remLetHom _    t                                   = simpCxt (inj t)

instance (Functor f, Functor g, f ≼ g) ⇒ RemLetHom f q g where
  remLetHom _
```

```
runUpHom :: UpState f q → QHom f q g
         → Term f → Term g
runUpHom st hom = runUpTrans (upTrans st hom)
```

Combining state

```
remLet :: (Func
       ⇒ Term f → (Vars, Term f)
remLet = runUpHom freeVarsSt remLetHom
```

An Example (Cont'd)

$$
\begin{array}{l}
\textbf{class } \mathit{RemLetHom}\ f\ q\ g \textbf{ where} \\
\quad \mathit{remLetHom} :: \mathit{QHom}\ f\ q\ g
\end{array}
$$

$$
\begin{array}{l}
\textbf{instance } (\mathit{Vars} \in q, \mathit{Let} \preceq g, \mathit{Functor}\ g) \Rightarrow \mathit{RemLetHom}\ \mathit{Let}\ q\ g \textbf{ where} \\
\quad \mathit{remLetHom}\ \mathit{qOf}\ (\mathit{LetIn}\ v\ \_\ s) \mid \neg\,(v\ \text{`}\mathit{member}\text{`}\ \mathit{qOf}\ s) = \mathit{Hole}\ s \\
\quad \mathit{remLetHom}\ \_\ \quad t \qquad\qquad\qquad\qquad\qquad\qquad\quad = \mathit{simpCxt}\ (\mathit{inj}\ t)
\end{array}
$$

$$
\begin{array}{l}
\textbf{instance } (\mathit{Functor}\ f, \mathit{Functor}\ g, f \preceq g) \Rightarrow \mathit{RemLetHom}\ f\ q\ g \textbf{ where} \\
\quad \mathit{remLetHom}\ \_ = \mathit{simpCxt}\ .\ \mathit{inj}
\end{array}
$$

Combining state transition and homomorphism

$$
\begin{array}{l}
\mathit{remLet} :: (\mathit{Functor}\ f, \mathit{FreeVarsSt}\ f, \mathit{RemLetHom}\ f\ \mathit{Vars}\ f) \\
\qquad\quad \Rightarrow \mathit{Term}\ f \rightarrow (\mathit{Vars}, \mathit{Term}\ f)
\end{array}
$$

$$
\mathit{remLet} = \mathit{runUpHom}\ \ \mathit{freeVarsSt}\ \mathit{remLetHom}
$$

$$
\begin{array}{l}
\mathit{remLet} :: \mathit{Term}\ \mathit{LetSig} \qquad\qquad\quad \rightarrow \mathit{Term}\ \mathit{LetSig} \\
\mathit{remLet} :: \mathit{Term}\ (\mathit{Inc} \oplus \mathit{LetSig}) \rightarrow \mathit{Term}\ (\mathit{Inc} \oplus \mathit{LetSig})
\end{array}
$$

Beyond Bottom-Up Tree Automata

What have we seen?

- Bottom-up tree acceptors (a.k.a. folds)
- Bottom-up tree transducers
- "dependent" versions thereof

Beyond Bottom-Up Tree Automata

What have we seen?

- Bottom-up tree acceptors (a.k.a. folds)
- Bottom-up tree transducers
- "dependent" versions thereof

Other Tree recursion schemes

- Top-down tree acceptors
- Top-down tree transducers
- "dependent" versions thereof

Beyond Bottom-Up Tree Automata

What have we seen?

- Bottom-up tree acceptors (a.k.a. folds)
- Bottom-up tree transducers
- “dependent” versions thereof

Other Tree recursion schemes

- Top-down tree acceptors
- Top-down tree transducers
- “dependent” versions thereof
- automata with bidirectional state propagation
- (restricted versions of macro tree transducers)

What have we gained?

Modularity & Reusability

- modularity along three dimensions (signature, sequential composition, state space)
- decoupling of state propagation and tree transformation
- operations on automata (beyond product & sum) allow us to construct new automata from old ones

What have we gained?

Modularity & Reusability

- modularity along three dimensions (signature, sequential composition, state space)
- decoupling of state propagation and tree transformation
- operations on automata (beyond product & sum) allow us to construct new automata from old ones

Interface between tree automata

- dependencies between automata by constraints on the state space
- modularity allows us to replace individual components

Try It Out!

This is part of the compositional data types Haskell library `compdata`:

```
> cabal install compdata
```

http://hackage.haskell.org/package/compdata