Deriving Modular Recursion Schemes from Tree Automata

Patrick Bahr

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

Computing Science Colloquium,
Utrecht University,
April 12th, 2012

Outline

1. Tree Automata
   - Bottom-Up Tree Acceptors
   - Bottom-Up Tree Transducers
2. Introducing Modularity
   - Composing State Spaces
   - Compositional Signatures
   - Decomposing Tree Transducers
3. Other Automata

Outline

1. Tree Automata
   - Bottom-Up Tree Acceptors
   - Bottom-Up Tree Transducers
2. Introducing Modularity
   - Composing State Spaces
   - Compositional Signatures
   - Decomposing Tree Transducers
3. Other Automata

Bottom-Up Tree Acceptors

Bottom-Up Tree Acceptors

$$f(q_1(x_1), q_2(x_2), \ldots, q_n(x_n)) \to q(f(x_1, x_2, \ldots, x_n))$$

An Example

The signature

$\mathcal{F} = \{\mathsf{and}/2, \mathsf{not}/1, \mathsf{tt}/0, \mathsf{ff}/0\}$

e.g.: $\mathsf{not}(\mathsf{and}(\mathsf{not}(\mathsf{ff}), \mathsf{and}(\mathsf{tt}, \mathsf{ff})))$

An Example

The signature

$\mathcal{F} = \{\mathsf{and}/2, \mathsf{not}/1, \mathsf{tt}/0, \mathsf{ff}/0\}$

e.g.: $\mathsf{not}(\mathsf{and}(\mathsf{not}(\mathsf{ff}), \mathsf{and}(\mathsf{tt}, \mathsf{ff})))$

The states

- set of states: $Q = \{q_0, q_1\}$
- $q_0 \rightsquigarrow$ false
- $q_1 \rightsquigarrow$ true
- accepting states: $Q_a = \{q_1\}$

An Example

The signature

$\mathcal{F} = \{\text{and}/2, \text{not}/1, \text{tt}/0, \text{ff}/0\}$

e.g.: $\text{not}(\text{and}(\text{not}(\text{ff}), \text{and}(\text{tt}, \text{ff})))$

The states

- set of states: $Q = \{q_0, q_1\}$
- $q_0 \rightsquigarrow$ false
- $q_1 \rightsquigarrow$ true
- accepting states: $Q_a = \{q_1\}$

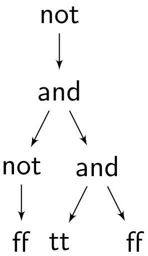
The rules of the automaton

$$\text{ff} \to q_0(\text{ff})$$
$$\text{tt} \to q_1(\text{tt})$$

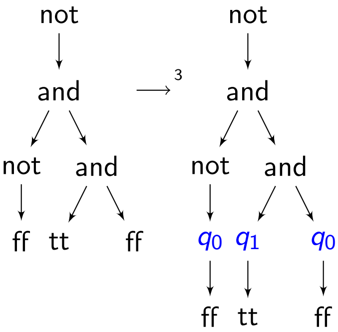
$$\text{not}(q_0(x)) \to q_1(\text{not}(x))$$
$$\text{not}(q_1(x)) \to q_0(\text{not}(x))$$

$$\text{and}(q_1(x), q_1(y)) \to q_1(\text{and}(x, y))$$
$$\text{and}(q_0(x), q_1(y)) \to q_0(\text{and}(x, y))$$
$$\text{and}(q_1(x), q_0(y)) \to q_0(\text{and}(x, y))$$
$$\text{and}(q_0(x), q_0(y)) \to q_0(\text{and}(x, y))$$

A Run of a Bottom-Up Tree Acceptor



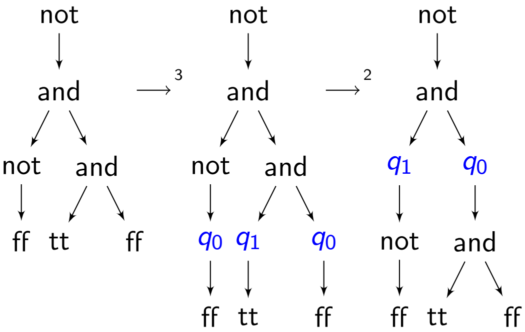
A Run of a Bottom-Up Tree Acceptor



$$\mathrm{ff} \to q_0(\mathrm{ff})$$

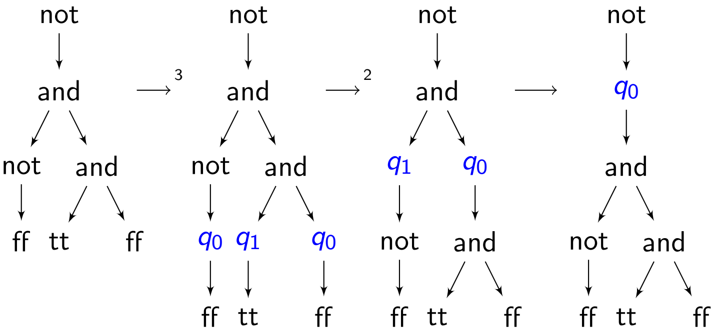
$$\mathrm{tt} \to q_1(\mathrm{tt})$$

A Run of a Bottom-Up Tree Acceptor



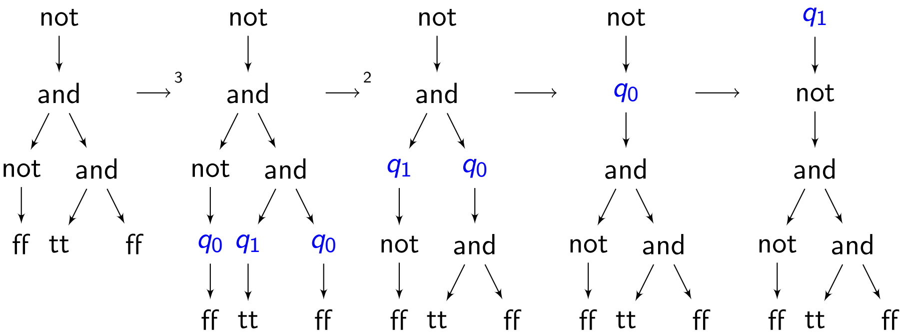
$$\text{not}(q_0(x)) \to q_1(\text{not}(x))$$
$$\text{and}(q_1(x), q_0(y)) \to q_0(\text{and}(x, y))$$

A Run of a Bottom-Up Tree Acceptor



$$\mathrm{not}(q_0(x)) \rightarrow q_1(\mathrm{not}(x))$$
$$\mathrm{and}(q_1(x), q_0(y)) \rightarrow q_0(\mathrm{and}(x, y))$$

A Run of a Bottom-Up Tree Acceptor



$$\mathrm{not}(q_0(x)) \to q_1(\mathrm{not}(x))$$

$$\mathrm{and}(q_1(x), q_0(y)) \to q_0(\mathrm{and}(x, y))$$

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

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

Functors

$\textbf{class}\ \mathit{Functor}\ f\ \textbf{where}$

$\quad \mathit{fmap} :: (a \to b) \to f\ a \to f\ b$

Terms in Haskell

Data types as fixed points of functors

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

Functors

$\textbf{class}\ \mathit{Functor}\ f\ \textbf{where}$

$\quad \mathit{fmap} :: (a \to b) \to f\ a \to f\ b$

Example

$\textbf{data}\ F\ a = \mathit{And}\ a\ a \mid \mathit{Not}\ a \mid \mathit{TT} \mid \mathit{FF}$

Terms in Haskell

Data types as fixed points of functors

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

Functors

class $Functor\ f$ **where**

$\quad fmap :: (a \to b) \to f\ a \to f\ b$

Example

data $F\ a = And\ a\ a \mid Not\ a \mid TT \mid FF$

instance $Functor\ F$ **where**

$$\begin{aligned} fmap\ f\ TT &= TT \\ fmap\ f\ (And\ x\ y) &= And\ (f\ x)\ (f\ y) \\ &\vdots \end{aligned}$$

Terms in Haskell

Data types as fixed points of functors

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

Functors

$$\begin{aligned}&\textbf{class}\ \mathit{Functor}\ f\ \textbf{where}\\ &\quad \mathit{fmap} :: (a \to b) \to f\ a \to f\ b\end{aligned}$$

Example

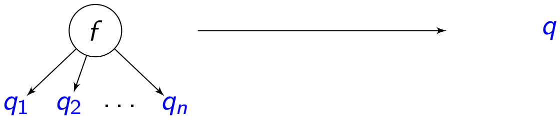
$$\textbf{data}\ F\ a = \mathit{And}\ a\ a \mid \mathit{Not}\ a \mid \mathit{TT} \mid \mathit{FF} \qquad \textbf{deriving}\ \mathit{Functor}$$

$$\begin{aligned}&\textbf{instance}\ \mathit{Functor}\ F\ \textbf{where}\\ &\quad \mathit{fmap}\ f\ \mathit{TT} &&= \mathit{TT}\\ &\quad \mathit{fmap}\ f\ (\mathit{And}\ x\ y) &&= \mathit{And}\ (f\ x)\ (f\ y)\\ &&&\vdots\end{aligned}$$

Bottom-Up State Transitions in Haskell

Bottom-Up State Transitions in Haskell

Bottom-Up State Transitions in Haskell

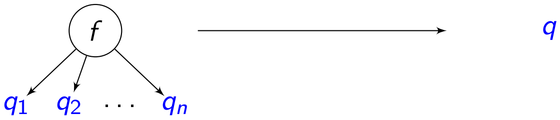


Bottom-Up State Transitions in Haskell

Bottom-up state transition rules as algebras

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

Bottom-Up State Transitions in Haskell

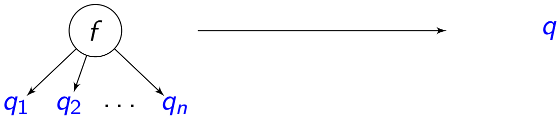


Bottom-up state transition rules as algebras

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

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

Bottom-Up State Transitions in Haskell



Bottom-up s a.k.a. catamorphism / fold **ras**

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

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

An Example

Signature

data $F\ a = And\ a\ a \mid Not\ a \mid TT \mid FF$

An Example

Signature

data $F\ a = And\ a\ a \mid Not\ a \mid TT \mid FF$

States

data $Q = Q0 \mid Q1$

Accepting states

$acc :: Q \to Bool$
$acc\ Q1 = True$
$acc\ Q0 = False$

An Example

Signature

$\textbf{data}\ F\ a = And\ a\ a \mid Not\ a \mid TT \mid FF$

States

$\textbf{data}\ Q = Q0 \mid Q1$

Accepting states

$$
\begin{aligned}
&acc :: Q \to Bool \\
&acc\ Q1 = True \\
&acc\ Q0 = False
\end{aligned}
$$

State transition function

$$
\begin{aligned}
&trans :: F\ Q \to Q \\
&trans\ FF && = Q0 \\
&trans\ TT && = Q1 \\
&trans\ (Not\ \ Q0) && = Q1 \\
&trans\ (Not\ \ Q1) && = Q0 \\
&trans\ (And\ Q1\ \ Q1) && = Q1 \\
&trans\ (And\ \_\ \ \ \_\ \ ) && = Q0
\end{aligned}
$$

Bottom-Up Tree Transducers

Bottom-Up Tree Transducers

Bottom-Up Tree Transducers

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

$$f \in \mathcal{F} \qquad t \in \mathcal{T}(\mathcal{G}, \mathcal{X}) \qquad \mathcal{X} = \{x_1, x_2, \ldots, x_n\}$$

An Example

The signature

$\mathcal{F} = \{\mathsf{and}/2, \mathsf{not}/1, \mathsf{ff}/0, \mathsf{tt}/0, \mathsf{b}/0\}$

An Example

The signature

$\mathcal{F} = \{\mathrm{and}/2, \mathrm{not}/1, \mathrm{ff}/0, \mathrm{tt}/0, \mathrm{b}/0\}$

The states

- $q_0 \rightsquigarrow$ false
- $q_1 \rightsquigarrow$ true
- $q_2 \rightsquigarrow$ don't know

An Example

The signature

$\mathcal{F} = \{\mathrm{and}/2, \mathrm{not}/1, \mathrm{ff}/0, \mathrm{tt}/0, \mathrm{b}/0\}$

The states

- $q_0 \rightsquigarrow$ false
- $q_1 \rightsquigarrow$ true
- $q_2 \rightsquigarrow$ don't know

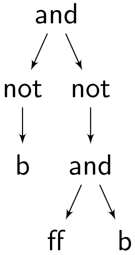
Transduction rules

$$\mathrm{tt} \to q_1(\mathrm{tt}) \qquad \mathrm{not}(q_0(x)) \to q_1(\mathrm{tt})$$
$$\mathrm{ff} \to q_0(\mathrm{ff}) \qquad \mathrm{not}(q_1(x)) \to q_0(\mathrm{ff})$$
$$\mathrm{b} \to q_2(\mathrm{b}) \qquad \mathrm{not}(q_2(x)) \to q_2(\mathrm{not}(x))$$

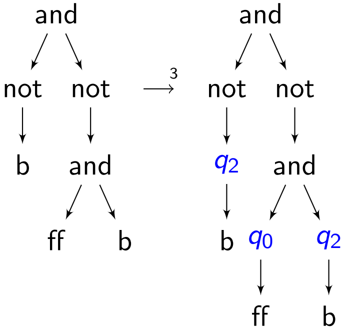
$$\mathrm{and}(q(x), p(y)) \to q_0(\mathrm{ff}) \quad \text{if } q_0 \in \{p, q\}$$

$$\mathrm{and}(q_1(x), q_1(y)) \to q_1(\mathrm{tt})$$
$$\mathrm{and}(q_1(x), q_2(y)) \to q_2(y)$$
$$\mathrm{and}(q_2(x), q_1(y)) \to q_2(x)$$
$$\mathrm{and}(q_2(x), q_2(y)) \to q_2(\mathrm{and}(x, y))$$

A Run of a Bottom-Up Transducer



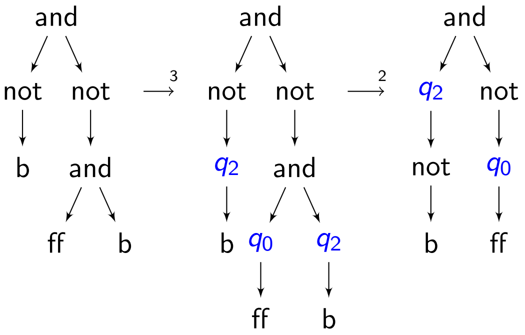
A Run of a Bottom-Up Transducer



$$\mathrm{ff} \to q_0(\mathrm{ff})$$

$$\mathrm{b} \to q_2(\mathrm{b})$$

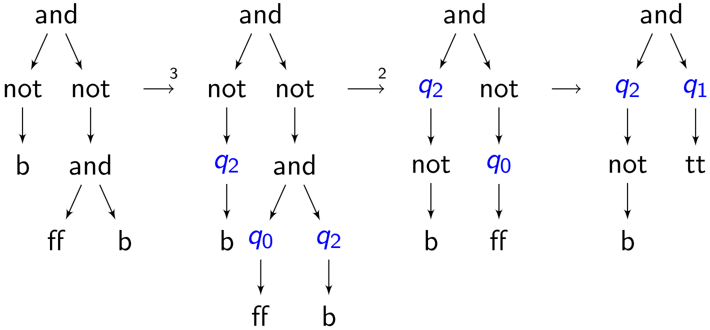
A Run of a Bottom-Up Transducer



$$\mathrm{not}(q_2(x)) \to q_2(\mathrm{not}(x))$$

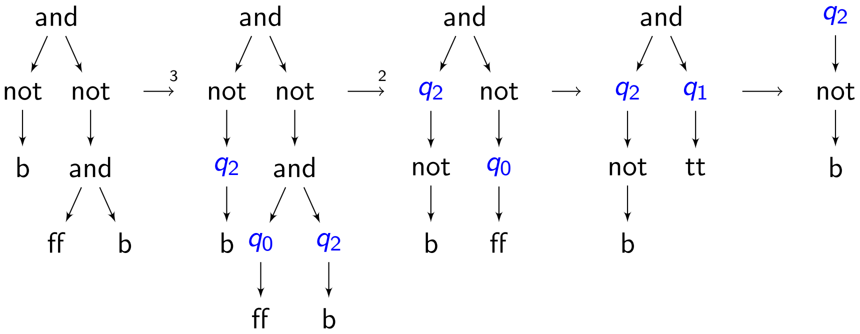
$$\mathrm{and}(q(x), p(y)) \to q_0(\mathrm{ff}) \quad \text{if } q_0 \in \{p, q\}$$

A Run of a Bottom-Up Transducer



$$\mathrm{not}(q_0(x)) \to q_1(\mathrm{tt})$$

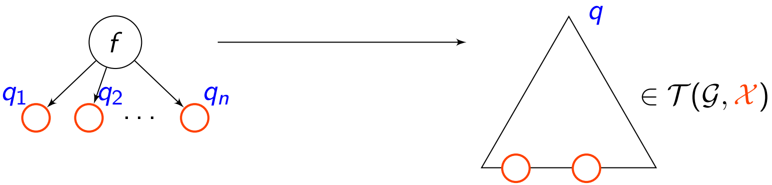
A Run of a Bottom-Up Transducer



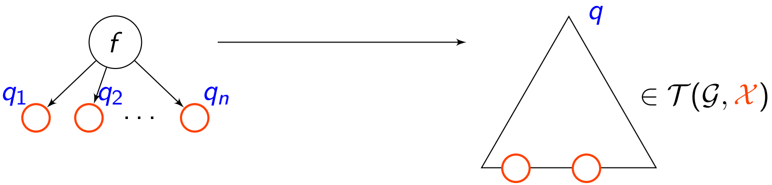
$$\mathsf{and}(q_2(x), q_1(y)) \to q_2(x)$$

Bottom-Up Tree Transducers

Bottom-Up Tree Transducers



Bottom-Up Tree Transducers

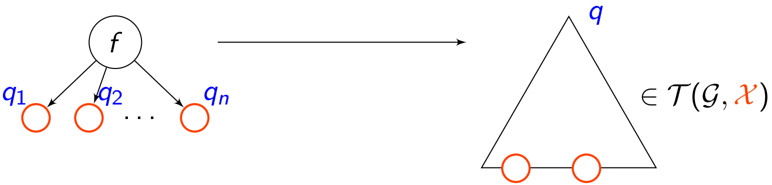


From terms to contexts

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

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

Bottom-Up Tree Transducers



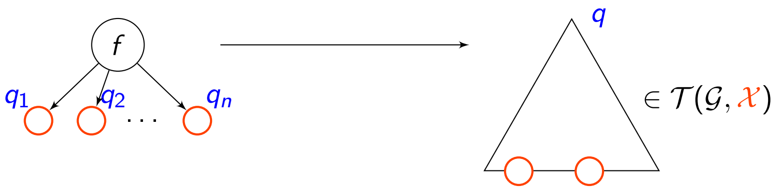
type *Term f* = *Context f Empty*

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

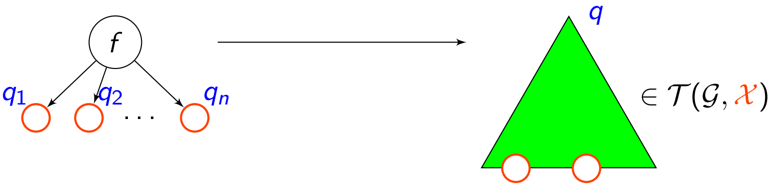
data $Term \; f \; = In \, (f \, (Term \; f \;))$

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

Representing transduction rules, [Hasuo et al. 2007]

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

Bottom-Up Tree Transducers



From terms to contexts

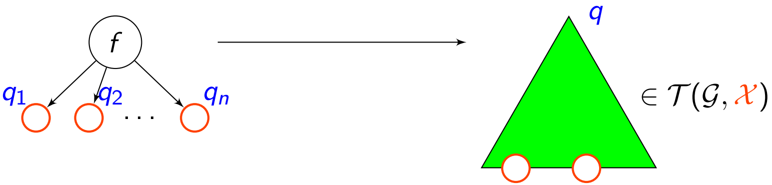
data $\mathit{Term}\ f = \mathit{In}\ (f\ (\mathit{Term}\ f))$

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

Representing transduction rules, [Hasuo et al. 2007]

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

Bottom-Up Tree Transducers



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

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

An Example

Signature And state

$$\textbf{data}\ F\ a = And\ a\ a \mid Not\ a \mid TT \mid FF \mid B$$
$$\textbf{data}\ Q\quad = Q0 \mid Q1 \mid Q2$$

An Example

Signature And state

$$
\begin{aligned}
&\textbf{data}\ F\ a = And\ a\ a \mid Not\ a \mid TT \mid FF \mid B\\
&\textbf{data}\ Q\ \ \ = Q0 \mid Q1 \mid Q2
\end{aligned}
$$

$$
\begin{aligned}
&\textbf{type}\ UpTrans\ f\ q\ g = \forall\ a.f\ (q,a) \rightarrow (q,\ Context\ g\ a)\\
&\textbf{data}\ Context\ f\ a = In\ (f\ (Context\ f\ a)) \mid Hole\ a
\end{aligned}
$$

The transduction function

$trans :: UpTrans\ F\ Q\ F$

An Example

Signature And state

data $F\ a = And\ a\ a \mid Not\ a \mid TT \mid FF \mid B$
data $Q \quad = Q0 \mid Q1 \mid Q2$

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

The transduction function

$trans :: UpTrans\ F\ Q\ F$

$\mathrm{tt} \rightarrow q_1(\mathrm{tt})$ $\qquad$ $trans\ TT = (Q1, tt\ )$

An Example

Signature And state

$$\textbf{data}\ F\ a = And\ a\ a \mid Not\ a \mid TT \mid FF \mid B$$
$$\textbf{data}\ Q \quad = Q0 \mid Q1 \mid Q2$$

$$\textbf{type}\ UpTrans\ f\ q\ g = \forall\ a.f\ (q,a) \rightarrow (q,\ Context\ g\ a)$$
$$\textbf{data}\ Context\ f\ a = In\ (f\ (Context\ f\ a)) \mid Hole\ a$$

The transduction function

$$tt :: Context\ F\ a$$
$$tt = In\ TT$$

$trans :: UpTrans\ F\ Q\ F$

$\mathrm{tt} \rightarrow q_1(\mathrm{tt})$ $\qquad$ $trans\ TT = (Q1, tt\ )$

An Example

Signature And state

$$\textbf{data}\ F\ a = And\ a\ a \mid Not\ a \mid TT \mid FF \mid B$$
$$\textbf{data}\ Q \quad = Q0 \mid Q1 \mid Q2$$

$$\textbf{type}\ UpTrans\ f\ q\ g = \forall\ a.f\ (q,a) \to (q,\ Context\ g\ a)$$
$$\textbf{data}\ Context\ f\ a = In\ (f\ (Context\ f\ a)) \mid Hole\ a$$

The transduction function

$$trans :: UpTrans\ F\ Q\ F$$

$$\mathrm{tt} \to q_1(\mathrm{tt}) \qquad trans\ TT = (Q1, tt\ )$$
$$\mathrm{not}(q_2(x)) \to q_2(\mathrm{not}(x)) \qquad trans\ (Not\ (Q2,\ x)) = (Q2, not\ \ (Hole\ x))$$

An Example

Signature And state

$$\textbf{data}\ F\ a = And\ a\ a \mid Not\ a \mid TT \mid FF \mid B$$
$$\textbf{data}\ Q \quad = Q0 \mid Q1 \mid Q2$$

$$\textbf{type}\ UpTrans\ f\ q\ g = \forall\ a.f\ (q,a) \rightarrow (q,\ Context\ g\ \ a)$$
$$\textbf{data}\ Context\ f\ \ a = In\ (f\ (Context\ f\ \ a)) \mid Hole\ a$$

The transduction functi

$$not :: Context\ F\ a \rightarrow Context\ F\ a$$
$$not = In\ .\ Not$$

$$trans :: UpTrans\ F\ Q\ F$$

$$\mathrm{tt} \rightarrow q_1(\mathrm{tt}) \qquad trans\ TT = (Q1, tt\ \ )$$
$$\mathrm{not}(q_2(x)) \rightarrow q_2(\mathrm{not}(x)) \qquad trans\ (Not\ (Q2,\ x)) = (Q2, not\ \ (Hole\ x))$$

An Example

Signature And state

data $F\ a = And\ a\ a \mid Not\ a \mid TT \mid FF \mid B$
data $Q \quad = Q0 \mid Q1 \mid Q2$

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

The transduction function

$trans :: UpTrans\ F\ \ Q\ \ F$

$$\mathrm{tt} \rightarrow q_1(\mathrm{tt}) \qquad trans\ TT = (Q1, tt\ )$$

$$\mathrm{not}(q_2(x)) \rightarrow q_2(\mathrm{not}(x)) \qquad trans\ (Not\ (Q2,\ x)) = (Q2, not\ \ (Hole\ x))$$

$$\mathrm{and}(q(x), p(y)) \rightarrow q_0(\mathrm{ff}) \quad \text{if } q_0 \in \{q, p\}$$

An Example

Signature And state

$$\textbf{data}\ F\ a = And\ a\ a \mid Not\ a \mid TT \mid FF \mid B$$
$$\textbf{data}\ Q\ \ = Q0 \mid Q1 \mid Q2$$

$$\textbf{type}\ UpTrans\ f\ q\ g = \forall\ a.f\ (q,a) \rightarrow (q,\ Context\ g\ \ a)$$
$$\textbf{data}\ Context\ f\ \ a = In\ (f\ (Context\ f\ \ a)) \mid Hole\ a$$

The transduction function

$trans :: UpTrans\ F\ Q\ F$

$$\mathrm{tt} \rightarrow q_1(\mathrm{tt}) \qquad trans\ TT = (Q1, tt\ )$$
$$\mathrm{not}(q_2(x)) \rightarrow q_2(\mathrm{not}(x)) \qquad trans\ (Not\ (Q2,\ x)) = (Q2, not\ \ (Hole\ x))$$
$$\mathrm{and}(q(x), p(y)) \rightarrow q_0(\mathrm{ff}) \quad \text{if } q_0 \in \{q, p\}$$
$$trans\ (And\ (q,\ x)\ (p,\ y)) \mid q \equiv Q0 \vee p \equiv Q0 = (Q0, ff)$$

Outline

1. Tree Automata
   - Bottom-Up Tree Acceptors
   - Bottom-Up Tree Transducers
2. Introducing Modularity
   - Composing State Spaces
   - Compositional Signatures
   - Decomposing Tree Transducers
3. Other Automata

Composing State Spaces – Motivating Example

A simple expression language

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

Composing State Spaces – Motivating Example

A simple expression language

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

Task: writing a code generator

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

Composing State Spaces – Motivating Example

A simple expression language

data *Sig e* = *Val Int* | *Plus e e*

Task: writing a code generator

type *Addr* = *Int*
data *Instr* = *Acc Int* | *Load Addr* | *Store Addr* | *Add Addr*
type *Code* = [*Instr*]

The problem

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

Composing State Spaces – Motivating Example

A simple expression language

data *Sig e* = *Val Int* | *Plus e e*

Task: writing a code generator

type *Addr* = *Int*
data *Instr* = *Acc Int* | *Load Addr* | *Store Addr* | *Add Addr*
type *Code* = [*Instr*]

Sig Code → *Code*

The problem

codeSt :: *UpState Sig Code*
codeSt (*Val i*) = [*Acc i*]
codeSt (*Plus x y*) = *x* ++ [*Store a*] ++ *y* ++ [*Add a*]
  where *a* = . . .

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

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

Run the automaton

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

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

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

Product Automata

Deriving projections

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

$pr :: b \rightarrow a$

Product Automata

Deriving projections

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

$a \in b$ iff

- b is of the form $(b_1, (b_2, ...))$ 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 $(b_1, (b_2, ...))$ 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 $(b_1, (b_2, ...))$ 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 $(b_1, (b_2, ...))$ 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**
  $pr :: b \to a$

$a \in b$ iff

- b is of the form $(b_1, (b_2, \ldots))$ 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**
$pr :: b \to a$

$a \in b$ iff

- b is of the form $(b_1, (b_2, \ldots))$ 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 $(b_1, (b_2, \ldots))$ 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 state transition

$$(\otimes) :: (p \in c, q \in c) \Rightarrow DUpState\ f\ c\ p \to DUpState\ f\ c\ q \to DUpState\ f\ c\ (p, q)$$
$$(sp \otimes sq)\ t = (sp\ t, sq\ t)$$

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

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

From state transition to dependent state transition

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

Running Dependent State Transition Functions

The types

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

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

From state transition to dependent state transition

$\mathit{dUpState} :: \mathit{Functor}\ f \Rightarrow \mathit{UpState}\ f\ q \to \mathit{DUpState}\ f\ p\ q$

$\mathit{dUpState}\ \mathit{st} = \mathit{st}\ .\ \mathit{fmap}\ \mathit{pr}$

Running dependent state transitions

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

$\mathit{runDUpState}\ f = \mathit{runUpState}\ f$

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

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

Generating fresh addresses

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

The Code Generator Example

The code generator

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

Generating fresh addresses

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

Combining the components

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

Outline

Combining Signatures

Coproduct of signatures

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

$f \oplus g$ is the sum of the signatures f and g

Combining Signatures

Coproduct of signatures

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

$f \oplus g$ is the sum of the signatures f and g

Example

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

Combining Automata

Making the height compositional

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

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

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**
  $heightSt\ (Inl\ x) = heightSt\ x$
  $heightSt\ (Inr\ x) = heightSt\ x$

Defining the height on Sig

instance *HeightSt Sig* **where**
  $heightSt\ (Val\ \_) \quad = 0$
  $heightSt\ (Plus\ x\ y) = 1 + max\ x\ y$

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

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

Defining the height on Inc

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

Subsignatures

Subsignature type class

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

Subsignatures

Subsignature type class

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

$f \preceq g$ iff

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

Subsignatures

Subsignature type class

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

For example: $Inc \preceq \underbrace{Inc \oplus Sig}_{Sig'}$

$f \preceq g$ iff

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

Subsignatures

Subsignature type class

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

For example: $Inc \preceq \underbrace{Inc \oplus Sig}_{Sig'}$

$f \preceq g$ iff

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

Injection and projection functions

$inject :: (g \preceq f) \Rightarrow g \; (Context \; f \; a) \rightarrow Context \; f \; a$
$inject = In \, . \, inj$

Outline

Tree Homomorphisms

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

Tree Homomorphisms

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

Tree Homomorphisms

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

Tree Homomorphisms

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

Example (Desugaring)

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

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

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

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

Tree Homomorphisms

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

Example (Desugaring)

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

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

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

$\mathit{simpCxt} :: \mathit{Functor}\ g \Rightarrow g\ a \to \mathit{Context}\ g\ a$
$\mathit{simpCxt}\ t = \mathit{In}\ (\mathit{fmap}\ \mathit{Hole}\ t)$

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

Stateful Tree Homomorphisms

Decomposing tree transducers

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

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

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

Stateful Tree Homomorphisms

Decomposing tree transducers

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

Making homomorphisms dependent on a state

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

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

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

Making homomorphisms dependent on a state

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

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

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

Using implicit parameters

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

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

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

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

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

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)

$$
\begin{array}{l}
\textbf{class}\ \mathit{RemLetHom}\ f\ q\ g\ \textbf{where} \\
\quad \mathit{remLetHom} :: \mathit{QHom}\ f\ q\ g \\[1ex]
\textbf{instance}\ (\mathit{Vars} \in q, \mathit{Let} \preceq g, \mathit{Functor}\ g) \Rightarrow \mathit{RemLetHom}\ \mathit{Let}\ q\ g\ \textbf{where} \\
\quad \mathit{remLetHom}\ (\mathit{LetIn}\ v\ \_\ s) \mid \neg\,(v\ \text{`}\mathit{member}\text{`}\ \mathit{below}\ s) = \mathit{Hole}\ s \\
\quad \mathit{remLetHom}\ t \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad\ = \mathit{simpCxt}\ (\mathit{inj}\ t) \\[1ex]
\textbf{instance}\ (\mathit{Functor}\ f, \mathit{Functor}\ g, f \preceq g) \Rightarrow \mathit{RemLetHom}\ f\ q\ g\ \textbf{where} \\
\quad \mathit{remLetHom} = \mathit{simpCxt}\ .\ \mathit{inj}
\end{array}
$$

An Example (Cont'd)

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

Combining state transition and homomorphism

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

An Example (Cont'd)

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

Combining state transition and homomorphism

$$\mathit{runUpHom}\ \mathit{st}\ \mathit{hom} = \mathit{runUpTrans}\ (\mathit{upTrans}\ \mathit{st}\ \mathit{hom})$$

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

An Example (Cont'd)

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

Combining state transition and homomorphism

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

$$\begin{aligned}
&\mathit{remLet} :: \mathit{Term}\ \mathit{LetSig} &&\to \mathit{Term}\ \mathit{LetSig}\\
&\mathit{remLet} :: \mathit{Term}\ (\mathit{Inc} \oplus \mathit{LetSig}) &&\to \mathit{Term}\ (\mathit{Inc} \oplus \mathit{LetSig})
\end{aligned}$$

Outline

1. Tree Automata
   - Bottom-Up Tree Acceptors
   - Bottom-Up Tree Transducers
2. Introducing Modularity
   - Composing State Spaces
   - Compositional Signatures
   - Decomposing Tree Transducers
3. Other Automata

Top-Down Tree Transducers

Top-Down Tree Transducers

Top-Down Tree Transducers

$$q(f(x_1, x_2, \ldots, x_n)) \quad \longrightarrow \quad t$$

$f \in \mathcal{F}$ $\qquad t \in \mathcal{T}(\mathcal{G}, Q(\mathcal{X}))$ $\qquad Q(\mathcal{X}) = \{p(x_i) \mid p \in Q, 1 \leq i \leq n\}$

Top-Down Tree Transducers

$$q(f(x_1, x_2, \ldots, x_n)) \quad \longrightarrow \quad t$$

$$f \in \mathcal{F} \qquad t \in \mathcal{T}(\mathcal{G}, Q(\mathcal{X})) \qquad Q(\mathcal{X}) = \{p(x_i) \mid p \in Q, 1 \leq i \leq n\}$$

Representation in Haskell

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

Decomposing Top-Down Tree Transducers

State transition depends on transformation

Successor states are assigned to variable occurrences on right-hand side.

$$q_0(f(x)) \to g(q_1(x), q_2(x))$$

Decomposing Top-Down Tree Transducers

State transition depends on transformation

Successor states are assigned to variable occurrences on right-hand side.

$$q_0(f(x)) \rightarrow g(q_1(x), q_2(x))$$

Assignment to variables instead

restriction: if $q(x)$ and $p(x)$ occur on right-hand side, then $p = q$.

Decomposing Top-Down Tree Transducers

State transition depends on transformation

Successor states are assigned to variable occurrences on right-hand side.

$$q_0(f(x)) \to g(q_1(x), q_2(x))$$

Assignment to variables instead

restriction: if $q(x)$ and $p(x)$ occur on right-hand side, then $p = q$.

How to represent top-down state transformations?

- first try: **type** $\mathit{DownState}\ f\ q = \forall\, a\,.\,(q, f\ a) \to f\ q$

Decomposing Top-Down Tree Transducers

State transition depends on transformation

Successor states are assigned to variable occurrences on right-hand side.

$$q_0(f(x)) \to g(q_1(x), q_2(x))$$

Assignment to variables instead

restriction: if $q(x)$ and $p(x)$ occur on right-hand side, then $p = q$.

How to represent top-down state transformations?

- first try: **type** $\mathit{DownState}\ f\ q = \forall\, a\,.\,(q, f\ a) \to f\ q$
- permits changing the shape of the input:

$$\begin{aligned}
&\mathit{bad} :: \forall\, a\,.\,(q, \mathit{Sig}\ a) \to \mathit{Sig}\ q\\
&\mathit{bad}\ (q, \mathit{Plus}\ x\ y) = \mathit{Val}\ 1\\
&\mathit{bad}\ (q, \mathit{Val}\ i) \quad\ \ = \mathit{Val}\ 1
\end{aligned}$$

Top-Down State Transitions

Using explicit placeholders

type $DownState\ f\ q = \forall\ i\ .\ Ord\ i \Rightarrow (q, f\ i) \rightarrow Map\ i\ q$

Top-Down State Transitions

Using explicit placeholders

type $DownState\ f\ q = \forall\ i\ .\ Ord\ i \Rightarrow (q, f\ i) \rightarrow Map\ i\ q$

$\rightsquigarrow$ construct function of type $\forall\ a\ .\ (q, f\ a) \rightarrow f\ q$ that preserves the shape

Top-Down State Transitions

Using explicit placeholders

type $DownState\ f\ q = \forall\ i\ .\ Ord\ i \Rightarrow (q, f\ i) \rightarrow Map\ i\ q$

$\leadsto$ construct function of type $\forall\ a\ .\ (q, f\ a) \rightarrow f\ q$ that preserves the shape

Combining with stateful tree homomorphisms

type $QHom\ f\ q\ g = \forall\ a\ .\ (?above :: q, ?below :: a \rightarrow q) \Rightarrow f\ a \rightarrow Context\ g\ a$

Top-Down State Transitions

Using explicit placeholders

type $\mathit{DownState}\ f\ q = \forall\, i\,.\, \mathit{Ord}\ i \Rightarrow (q, f\ i) \rightarrow \mathit{Map}\ i\ q$

$\rightsquigarrow$ construct function of type $\forall\, a\,.\,(q, f\ a) \rightarrow f\ q$ that preserves the shape

Combining with stateful tree homomorphisms

type $\mathit{QHom}\ f\ q\ g = \forall\, a\,.\,(?\mathit{above} :: q, ?\mathit{below} :: a \rightarrow q) \Rightarrow f\ a \rightarrow \mathit{Context}\ g\ a$

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

Why Tree Transducers

More structure, more flexibility

Tree transducers can manipulated more easily

Why Tree Transducers

More structure, more flexibility

Tree transducers can manipulated more easily, e.g.

data $(f :\&: a)\ e = f\ e :\&: a$

Why Tree Transducers

More structure, more flexibility

Tree transducers can manipulated more easily, e.g.

data $(f :\&: a)\ e = f\ e :\&: a$

$lift :: UpTrans\ f\ q\ g \rightarrow UpTrans\ (f :\&: a)\ q\ (g :\&: a)$

Why Tree Transducers

More structure, more flexibility

Tree transducers can manipulated more easily, e.g.

$\mathbf{data}\ (f :\&: a)\ e = f\ e :\&: a$

$lift :: UpTrans\ f\ q\ g \to UpTrans\ (f :\&: a)\ q\ (g :\&: a)$

Tree transducers compose

We may leverage composition theorems for tree transducers.

$comp :: DownTrans\ g\ p\ h \to DownTrans\ f\ q\ g \to DownTrans\ f\ (q, p)\ h$

Why Tree Transducers

More structure, more flexibility

Tree transducers can manipulated more easily, e.g.

$\textbf{data}\ (f :\&: a)\ e = f\ e :\&: a$

$lift :: UpTrans\ f\ q\ g \to UpTrans\ (f :\&: a)\ q\ (g :\&: a)$

Tree transducers compose

We may leverage composition theorems for tree transducers.

$comp :: DownTrans\ g\ p\ h \to DownTrans\ f\ q\ g \to DownTrans\ f\ (q, p)\ h$

$\leadsto$ potential for fusion

Beyond Tree Transducers

Bidirectional state transitions

- bottom-up + top-down state transition
- bottom-up + top-down state transition + stateful homomorphism

Beyond Tree Transducers

Bidirectional state transitions

- bottom-up + top-down state transition
- bottom-up + top-down state transition + stateful homomorphism

Macro Tree Transducers

- states may have arguments taken from the term
- necessary for 'non-local' transformations, e.g. substitution, inlining

Beyond Tree Transducers

Bidirectional state transitions

- bottom-up + top-down state transition
- bottom-up + top-down state transition + stateful homomorphism

Macro Tree Transducers

- states may have arguments taken from the term
- necessary for 'non-local' transformations, e.g. substitution, inlining

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

Beyond Tree Transducers

Bidirectional state transitions

- bottom-up + top-down state transition
- bottom-up + top-down state transition + stateful homomorphism

Macro Tree Transducers

- states may have arguments taken from the term
- necessary for 'non-local' transformations, e.g. substitution, inlining

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

Beyond Tree Transducers

Bidirectional state transitions

- bottom-up + top-down state transition
- bottom-up + top-down state transition + stateful homomorphism

Macro Tree Transducers

- states may have arguments taken from the term
- necessary for 'non-local' transformations, e.g. substitution, inlining

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

Beyond Tree Transducers

Bidirectional state transitions

- bottom-up + top-down state transition
- bottom-up + top-down state transition + stateful homomorphism

Macro Tree Transducers

- states may have arguments taken from the term
- necessary for 'non-local' transformations, e.g. substitution, inlining

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

Beyond Tree Transducers

Bidirectional state transitions

- bottom-up + top-down state transition
- bottom-up + top-down state transition + stateful homomorphism

Macro Tree Transducers

- states may have arguments taken from the term
- necessary for 'non-local' transformations, e.g. substitution, inlining

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

e.g. for substitutions: $q = \mathit{Map}\ \mathit{Var}$

Beyond Tree Transducers

Bidirectional state transitions

- bottom-up + top-down state transition
- bottom-up + top-down state transition + stateful homomorphism

Macro Tree Transducers

- states may have arguments taken from the term
- necessary for 'non-local' transformations, e.g. substitution, inlining

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

e.g. for substitutions: $q = \mathit{Map}\ \mathit{Var}$

generic programming