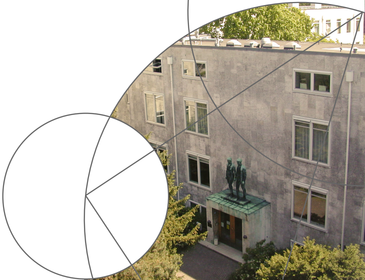
Programming Macro Tree Transducers

Patrick Bahr[1] Laurence E. Day[2]

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

[2]University of Nottingham,
Functional Programming Laboratory
led@cs.nott.ac.uk



Macro Tree Transducers on One Slide

Tree Transducers in FP

- automaton transforming trees to trees
- states are interpreted as functions

$\rightsquigarrow$ tree transducer = set of mutually recursive functions

Macro Tree Transducers on One Slide

Tree Transducers in FP

- automaton transforming trees to trees
- states are interpreted as functions

$\rightsquigarrow$ tree transducer = set of mutually recursive functions

Macro tree transducers

- extension of tree transducers
- each function may have accumulation parameters

This Paper: A Different Interpretation of MTT

still: MTTs as generalisation of top-down tree transducers

This Paper: A Different Interpretation of MTT

still: MTTs as generalisation of top-down tree transducers

Our interpretation of tree transducers

- literal interpretation: states are still states
- hence: a single function $\rightsquigarrow$ meta programming

This Paper: A Different Interpretation of MTT

still: MTTs as generalisation of top-down tree transducers

Our interpretation of tree transducers

- literal interpretation: states are still states
- hence: a single function $\rightsquigarrow$ meta programming

How so?

This Paper: A Different Interpretation of MTT

still: MTTs as generalisation of top-down tree transducers

Our interpretation of tree transducers

- literal interpretation: states are still states
- hence: a single function $\rightsquigarrow$ meta programming

How so?

$$\text{Macro Tree Transducers} = \text{Tree Transducers} + \text{parametricity}$$

Agenda

1. From String Acceptors to Tree Transducers

Agenda

1. From String Acceptors to Tree Transducers
2. Programming with Tree Transducers in Haskell

Agenda

1. From String Acceptors to Tree Transducers
2. Programming with Tree Transducers in Haskell
3. Tree Transducers with Polymorphic State Space

Agenda

1. From String Acceptors to Tree Transducers
2. Programming with Tree Transducers in Haskell
3. Tree Transducers with Polymorphic State Space
4. Macro Tree Transducers
   (= Tree Transducers with Accumulation Parameters)

Finite State Automata – On Strings

w o r d

Finite State Automata – On Strings

q_0 w o r d

Finite State Automata – On Strings

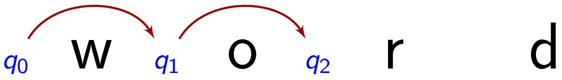
q_0 w o r d

$$q, s \rightarrow q'$$

Finite State Automata – On Strings

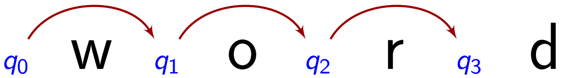
$$q, s \rightarrow q'$$

Finite State Automata – On Strings



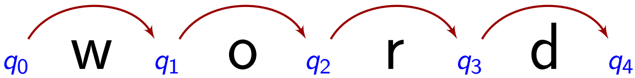
$$q, s \rightarrow q'$$

Finite State Automata – On Strings



$$q, s \rightarrow q'$$

Finite State Automata – On Strings



$$q, s \rightarrow q'$$

Finite State Automata – On Strings

$$q_0 \quad \mathsf{w} \quad q_1 \quad \mathsf{o} \quad q_2 \quad \mathsf{r} \quad q_3 \quad \mathsf{d} \quad q_4 \in Q_F?$$

$$q, s \quad \rightarrow \quad q'$$

Finite State Automata – On Strings

$$q_0 \;\; \mathrm{w} \;\; q_1 \;\; \mathrm{o} \;\; q_2 \;\; \mathrm{r} \;\; q_3 \;\; \mathrm{d} \;\; q_4 \in Q_F?$$

Acceptor

$$q, s \rightarrow q'$$

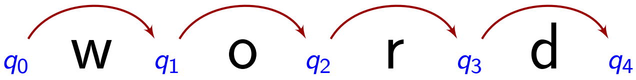
Finite State Automata – On Strings

q_0 w q_1 o q_2 r q_3 d $q_4 \in Q_F$?

Transducer?

$$q, s \quad \rightarrow \quad q'$$

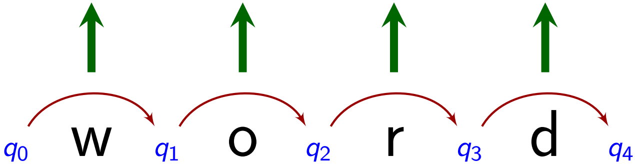
Finite State Automata – On Strings



Transducer

$$q, s \rightarrow q', w$$

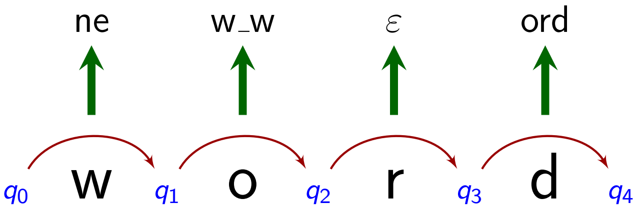
Finite State Automata – On Strings



Transducer

$$q, s \quad \rightarrow \quad q', w$$

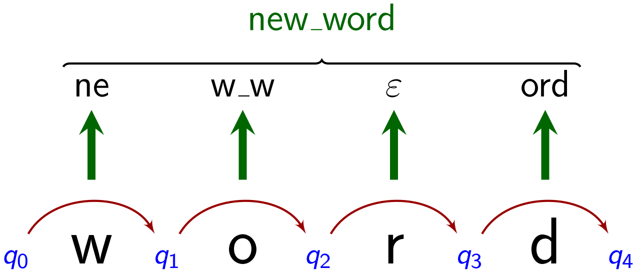
Finite State Automata – On Strings



Transducer

$$q, s \quad \rightarrow \quad q', w$$

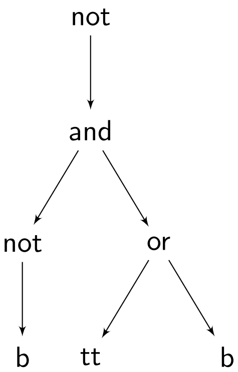
Finite State Automata – On Strings



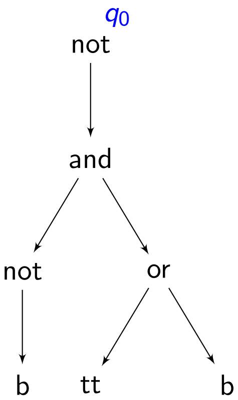
Transducer

$$q, s \quad \rightarrow \quad q', w$$

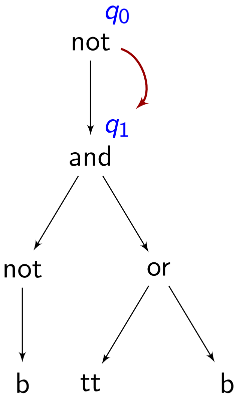
Now on Trees!



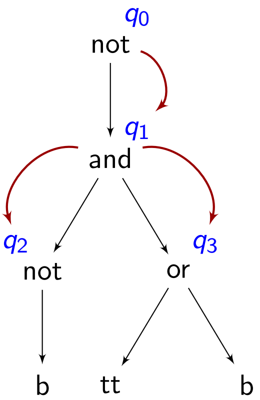
Now on Trees!



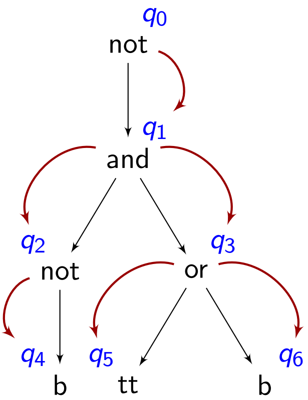
Now on Trees!



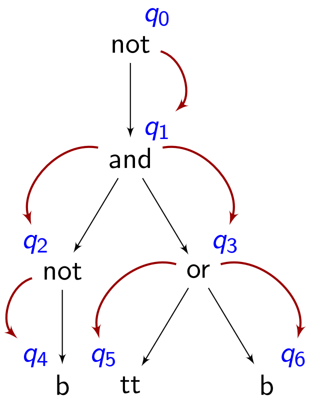
Now on Trees!



Now on Trees!

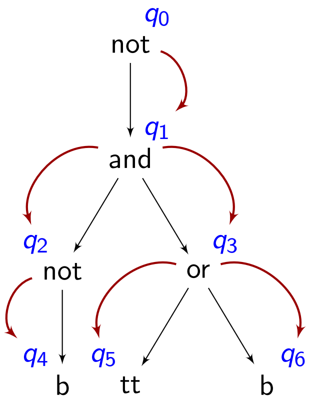


Now on Trees!



$$q, f \rightarrow q_1, \ldots, q_n$$

Now on Trees!



$$q, f \rightarrow q_1, \ldots, q_n$$

Often rendered as a rewrite rule:

$$q(f(x_1, \ldots, x_n)) \rightarrow f(q_1(x_1), \ldots, q_n(x_n))$$

Tree Transducers

$$q(f(x_1, \ldots, x_n)) \quad \rightarrow \quad f(q_1(x_1), \ldots, q_n(x_n))$$

Tree Transducers

$$q(f(x_1, \ldots, x_n)) \quad \rightarrow \quad f(q_1(x_1), \ldots, q_n(x_n))$$

Tree Transducers

$$q(f(x_1, \ldots, x_n)) \quad \rightarrow \quad t[q'(x_i) | q' \in Q, 1 \leq i \leq n]$$

And now in Haskell

And now in Haskell

Representation in Haskell

$$\textbf{type}\ \mathit{Trans}_{\mathrm{D}}\ f\ q\ g = \forall a\,.\,(q, f\,a) \to g^{*}(q, a)$$

And now in Haskell

Representation in Haskell

$$\textbf{type } \mathit{Trans}_{\mathrm{D}}\ f\ q\ g = \forall a\,.\,(q, f\,a) \to g^{*}(q, a)$$

Free Monad of a Functor g

$$\textbf{data } g^{*}\ a = \mathit{Re}\ a \mid \mathit{In}\ (g\ (g^{*}\ a))$$

Example: Substitution

$$\textbf{type}\ \mathit{Var} = \mathit{String}$$

$$\textbf{data}\ \mathit{Sig}\ a = \mathit{Add}\ a\ a \mid \mathit{Val}\ \mathit{Int} \mid \mathit{Let}\ \mathit{Var}\ a\ a \mid \mathit{Var}\ \mathit{Var}$$

Example: Substitution

$$\textbf{type}\ \mathit{Var} = \mathit{String}$$

$$\textbf{data}\ \mathit{Sig}\ a = \mathit{Add}\ a\ a \mid \mathit{Val}\ \mathit{Int} \mid \mathit{Let}\ \mathit{Var}\ a\ a \mid \mathit{Var}\ \mathit{Var}$$

$$
\begin{aligned}
&\mathit{trans}_{\mathrm{subst}} :: \ \mathit{Trans}_{\mathrm{D}}\ \ \mathit{Sig}\ (\mathit{Map}\ \mathit{Var}\ (\ \mu\mathit{Sig}\ ))\ \mathit{Sig}\\
&\mathit{trans}_{\mathrm{subst}}\ (m, \mathit{Var}\ v) &&= \textbf{case}\ \mathit{Map.lookup}\ v\ m\ \textbf{of}\\
& && \qquad \mathit{Nothing} \rightarrow \mathit{iVar}\ v\\
& && \qquad \mathit{Just}\ t \quad\ \rightarrow \mathit{toFree}\ t\\
&\mathit{trans}_{\mathrm{subst}}\ (m, \mathit{Let}\ v\ b\ s) &&= \mathit{iLet}\ v\ (\mathit{Re}\ (m, b))\\
& && \qquad\qquad\quad (\mathit{Re}\ (m \setminus v, s))\\
&\mathit{trans}_{\mathrm{subst}}\ (m, \mathit{Val}\ n) &&= \mathit{iVal}\ n\\
&\mathit{trans}_{\mathrm{subst}}\ (m, \mathit{Add}\ x\ y) &&= \mathit{Re}\ (m, x)\ \text{`}\mathit{iAdd}\text{`}\ \mathit{Re}\ (m, y)
\end{aligned}
$$

Example: Substitution

$$\textbf{type}\ Var = String$$

$$\textbf{data}\ Sig\ a = Add\ a\ a \mid Val\ Int \mid Let\ Var\ a\ a \mid Var\ Var$$

$$\textbf{type}\ \mu f = f^{*}\ Empty$$

$$trans_{\mathsf{subst}} :: Trans_{\mathsf{D}}\ Sig\ (Map\ Var\ (\mu Sig))\ Sig$$

$$
\begin{aligned}
trans_{\mathsf{subst}}\ (m, Var\ v) &= \textbf{case}\ Map.lookup\ v\ m\ \textbf{of} \\
&\qquad Nothing \to iVar\ v \\
&\qquad Just\ t \quad\ \to toFree\ t \\
trans_{\mathsf{subst}}\ (m, Let\ v\ b\ s) &= iLet\ v\ (Re\ (m, b)) \\
&\qquad\qquad (Re\ (m \setminus v, s)) \\
trans_{\mathsf{subst}}\ (m, Val\ n) &= iVal\ n \\
trans_{\mathsf{subst}}\ (m, Add\ x\ y) &= Re\ (m, x)\ `iAdd`\ Re\ (m, y)
\end{aligned}
$$

Example: Substitution

$$\textbf{type}\ \mathit{Var} = \mathit{String}$$

$$\textbf{data}\ \mathit{Sig}\ a = \mathit{Add}\ a\ a \mid \mathit{Val}\ \mathit{Int} \mid \mathit{Let}\ \mathit{Var}\ a\ a \mid \mathit{Var}\ \mathit{Var}$$

$$\textbf{type}\ \mathit{Trans}_{\mathrm{D}}\ f\ q\ g = \forall a\,.\,(q, f\ a) \to g^{*}(q, a)$$

$$\mathit{trans}_{\mathrm{subst}} :: \mathit{Trans}_{\mathrm{D}}\ \mathit{Sig}\ (\mathit{Map}\ \mathit{Var}\ (\mu\mathit{Sig}))\ \mathit{Sig}$$

$$\begin{aligned}
\mathit{trans}_{\mathrm{subst}}\ (m, \mathit{Var}\ v) &= \textbf{case}\ \mathit{Map.lookup}\ v\ m\ \textbf{of} \\
&\qquad \mathit{Nothing} \to \mathit{iVar}\ v \\
&\qquad \mathit{Just}\ t \to \mathit{toFree}\ t \\
\mathit{trans}_{\mathrm{subst}}\ (m, \mathit{Let}\ v\ b\ s) &= \mathit{iLet}\ v\ (\mathit{Re}\ (m, b)) \\
&\qquad\qquad (\mathit{Re}\ (m \setminus v, s)) \\
\mathit{trans}_{\mathrm{subst}}\ (m, \mathit{Val}\ n) &= \mathit{iVal}\ n \\
\mathit{trans}_{\mathrm{subst}}\ (m, \mathit{Add}\ x\ y) &= \mathit{Re}\ (m, x)\ \text{`}\mathit{iAdd}\text{`}\ \mathit{Re}\ (m, y)
\end{aligned}$$

Example: Substitution

type *Var* = *String*

data *Sig a* = *Add a a* | *Val Int* | *Let Var a a* | *Var Var*

$$
\begin{aligned}
&\mathit{trans}_{\mathrm{subst}} :: \ \mathit{Trans}_{\mathrm{D}} \ \ \mathit{Sig}\ (\mathit{Map}\ \mathit{Var}\ (\ \mu \mathit{Sig}\ ))\ \mathit{Sig}\\
&\mathit{trans}_{\mathrm{subst}}\ (m, \mathit{Var}\ v) \quad = \mathbf{case}\ \mathit{Map.lookup}\ v\ m\ \mathbf{of}\\
&\qquad\qquad\qquad\qquad\qquad \mathit{Nothing} \rightarrow \mathit{iVar}\ v\\
&\qquad\qquad\qquad\qquad\qquad \mathit{Just}\ t \quad\ \rightarrow \mathit{toFree}\ t\\
&\mathit{trans}_{\mathrm{subst}}\ (m, \mathit{Let}\ v\ b\ s) = \mathit{iLet}\ v\ (\mathit{Re}\ (m, b))\\
&\qquad\qquad\qquad\qquad\qquad\qquad (\mathit{Re}\ (m \setminus v, s))\\
&\mathit{trans}_{\mathrm{subst}}\ (m, \mathit{Val}\ n) \quad = \mathit{iVal}\ n\\
&\mathit{trans}_{\mathrm{subst}}\ (m, \mathit{Add}\ x\ y) \ = \mathit{Re}\ (m, x)\ \text{`}\mathit{iAdd}\text{`}\ \mathit{Re}\ (m, y)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{subst} :: \mathit{Map}\ \mathit{Var}\ \mu \mathit{Sig} \rightarrow \mu \mathit{Sig} \rightarrow \mu \mathit{Sig}\\
&\mathit{subst} = [\![\mathit{trans}_{\mathrm{subst}}]\!]_{\mathrm{D}}
\end{aligned}
$$

Non-Example: Inlining

$$
\begin{aligned}
&\mathit{trans}_{\text{inline}} :: \mathit{Trans}_{\text{D}}\ \mathit{Sig}\ (\mathit{Map}\ \mathit{Var}\ \mu\mathit{Sig})\ \mathit{Sig}\\
&\mathit{trans}_{\text{inline}}\ (m, \mathit{Var}\ v) &&= \textbf{case}\ \mathit{Map.lookup}\ v\ m\ \textbf{of}\\
& &&\quad \mathit{Nothing} \to \mathit{iVar}\ v\\
& &&\quad \mathit{Just}\ e \quad \to \mathit{toFree}\ e\\
&\mathit{trans}_{\text{inline}}\ (m, \mathit{Let}\ v\ b\ s) &&= \mathit{Re}\ (m\,[v \mapsto b]\,, s)\\
&\mathit{trans}_{\text{inline}}\ (m, \mathit{Val}\ n) &&= \mathit{iVal}\ n\\
&\mathit{trans}_{\text{inline}}\ (m, \mathit{Add}\ x\ y) &&= \mathit{Re}\ (m, x)\ \text{`}\mathit{iAdd}\text{`}\ \mathit{Re}\ (m, y)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{inline} :: \mu\mathit{Sig} \to \mu\mathit{Sig}\\
&\mathit{inline} = [\![\mathit{trans}_{\text{inline}}]\!]_{\text{D}}\ \emptyset
\end{aligned}
$$

Non-Example: Inlining

$$
\begin{aligned}
&\mathit{trans}_{\mathrm{inline}} :: \mathit{Trans}_{\mathrm{D}}\ \mathit{Sig}\ (\mathit{Map}\ \mathit{Var}\ \mu\mathit{Sig})\ \mathit{Sig}\\
&\mathit{trans}_{\mathrm{inline}}\ (m, \mathit{Var}\ v) = \textbf{case}\ \mathit{Map.lookup}\ v\ m\ \textbf{of}\\
&\qquad \mathit{Nothing} \rightarrow \mathit{iVar}\ v\\
&\qquad \mathit{Just}\ e \rightarrow \mathit{toFree}\ e\\
&\mathit{trans}_{\mathrm{inline}}\ (m, \mathit{Let}\ v\ b\ s) = \mathit{Re}\ (m\,[v \mapsto b]\,, s)\\
&\mathit{trans}_{\mathrm{inline}}\ (m, \mathit{Val}\ n) = \mathit{iVal}\ n\\
&\mathit{trans}_{\mathrm{inline}}\ (m, \mathit{Add}\ x\ y) = \mathit{Re}\ (m, x)\ \text{`}\mathit{iAdd}\text{`}\ \mathit{Re}\ (m, y)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{inline} :: \mu\mathit{Sig} \rightarrow \mu\mathit{Sig}\\
&\mathit{inline} = [\![\mathit{trans}_{\mathrm{inline}}]\!]_{\mathrm{D}}\ \emptyset
\end{aligned}
$$

Recall the type $\mathit{Trans}_{\mathrm{D}}$

$$\textbf{type}\ \mathit{Trans}_{\mathrm{D}}\ f\ q\ g = \forall a\,.\,(q, f\,a) \rightarrow g^{*}(q, a)$$

Transducers with Polymorphic State Space

The original type $Trans_{\mathrm{D}}$

type $Trans_{\mathrm{D}}\ f\ q\ g = \forall a\,.\,(q, f\,a) \rightarrow g^{*}(q, a)$

Transducers with Polymorphic State Space

The original type $Trans_D$

type $Trans_D\ f\ q\ g = \forall a\,.\,(q, f\,a) \to g^*(q, a)$

An equivalent representation

type $Trans_D\ f\ q\ g = \forall a . q \to f \quad a \quad \to g^*(q, a)$

Transducers with Polymorphic State Space

The original type $Trans_D$

type $Trans_D\ f\ q\ g = \forall a\,.\,(q, f\,a) \rightarrow g^*(q, a)$

An equivalent representation

type $Trans_D\ f\ q\ g = \forall a . q \rightarrow f(q \rightarrow a)\ \rightarrow g^* \quad a$

Transducers with Polymorphic State Space

The original type $Trans_{\mathrm{D}}$

type $Trans_{\mathrm{D}}\ f\ q\ g = \forall a\,.\,(q, f\,a) \rightarrow g^{*}(q, a)$

An equivalent representation

type $Trans_{\mathrm{D}}\ f\ q\ g = \forall a.q \rightarrow f(q \rightarrow a)\ \rightarrow g^{*}\quad a$

Deriving the type $Trans_{\mathrm{M}}$

type $Trans_{\mathrm{M}}\ f\ q\ g = \forall\ a.q\ a \rightarrow f(q\qquad a\ \rightarrow a) \rightarrow g^{*}\ a$

Transducers with Polymorphic State Space

The original type $Trans_{\mathrm{D}}$

$$\textbf{type } Trans_{\mathrm{D}}\ f\ q\ g = \forall a\ .\ (q, f\,a) \to g^{*}(q, a)$$

An equivalent representation

$$\textbf{type } Trans_{\mathrm{D}}\ f\ q\ g = \forall a . q \to f(q \to a)\ \to g^{*}\quad a$$

Deriving the type $Trans_{\mathrm{M}}$

$$\textbf{type } Trans_{\mathrm{M}}\ f\ q\ g = \forall\ a . q\ a \to f(q\ (g^{*}\ a) \to a) \to g^{*}\ a$$

Example: Inlining

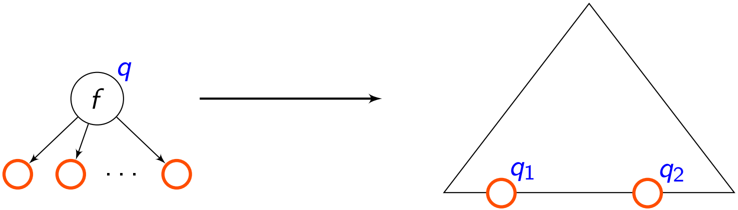
$$
\begin{aligned}
&\mathit{trans}_{\mathsf{inline}} :: \mathit{Trans}'_{\mathsf{M}}\ \mathit{Sig}\ (\mathit{Map}\ \mathit{Var})\ \mathit{Sig}\\
&\mathit{trans}_{\mathsf{inline}}\ m\ (\mathit{Var}\ v) = \textbf{case}\ \mathit{Map.lookup}\ v\ m\ \textbf{of}\\
&\qquad \mathit{Nothing} \rightarrow \mathit{iVar}\ v\\
&\qquad \mathit{Just}\ e \quad x \rightarrow e\\
&\mathit{trans}_{\mathsf{inline}}\ m\ (\mathit{Let}\ v\ b\ s) = s\ (m\,[v \mapsto b\ m])\\
&\mathit{trans}_{\mathsf{inline}}\ m\ (\mathit{Val}\ n) = \mathit{iVal}\ n\\
&\mathit{trans}_{\mathsf{inline}}\ m\ (\mathit{Add}\ x\ y) = x\ m\,\text{`}\mathit{iAdd}\text{`}\ y\ m
\end{aligned}
$$

Example: Inlining

$$
\begin{aligned}
&\mathit{trans}_{\mathsf{inline}} :: \mathit{Trans}'_{\mathsf{M}}\ \mathit{Sig}\ (\mathit{Map}\ \mathit{Var})\ \mathit{Sig}\\
&\mathit{trans}_{\mathsf{inline}}\ m\ (\mathit{Var}\ v) \quad\ = \textbf{case}\ \mathit{Map.lookup}\ v\ m\ \textbf{of}\\
&\qquad\qquad\qquad\qquad\qquad\quad \mathit{Nothing} \rightarrow \mathit{iVar}\ v\\
&\qquad\qquad\qquad\qquad\qquad\quad \mathit{Just}\ e \quad x \rightarrow e\\
&\mathit{trans}_{\mathsf{inline}}\ m\ (\mathit{Let}\ v\ b\ s) = s\ (m\,[v \mapsto b\ m])\\
&\mathit{trans}_{\mathsf{inline}}\ m\ (\mathit{Val}\ n) \quad\ = \mathit{iVal}\ n\\
&\mathit{trans}_{\mathsf{inline}}\ m\ (\mathit{Add}\ x\ y) \ = x\ m\ \text{`}\mathit{iAdd}\text{`}\ y\ m
\end{aligned}
$$

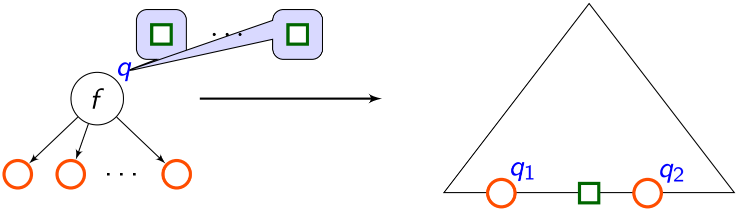
$$
\begin{aligned}
&\mathit{inline} :: \mu\mathit{Sig} \rightarrow \mu\mathit{Sig}\\
&\mathit{inline} = [\![\mathit{trans}_{\mathsf{inline}}]\!]_{\mathsf{M}}\ \emptyset
\end{aligned}
$$

Macro Tree Transduction Rule Illustrated



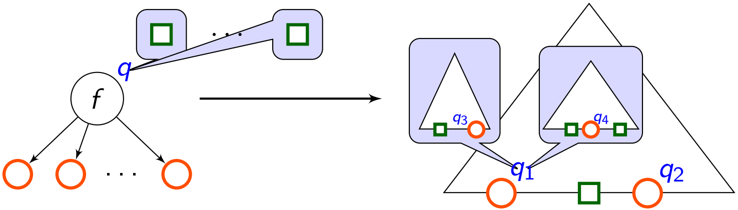
$$\textbf{type } \mathit{Trans}_{\mathrm{M}}\ f\ q\ g = \forall\ a.\ q\ a \to f(\underbrace{q\ (g^{*}\ a) \to a}_{\bigcirc}) \to g^{*}\ a$$

Macro Tree Transduction Rule Illustrated



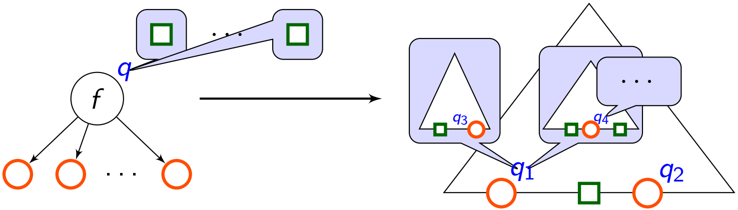
$$\textbf{type}\ Trans_{\mathrm{M}}\ f\ q\ g = \forall\ a.\ q\ a \to f(\underbrace{q\ (g^*\ a) \to a}_{\bigcirc}) \to g^*\ a$$

Macro Tree Transduction Rule Illustrated



$$\textbf{type } \mathit{Trans}_{\mathrm{M}}\ f\ q\ g = \forall\ a.\ q\ a \to f(\underbrace{q\ (g^*\ a) \to a}_{\bigcirc}) \to g^*\ a$$

Macro Tree Transduction Rule Illustrated



$$\textbf{type}\ \mathit{Trans}_{\mathrm{M}}\ f\ q\ g = \forall\ a.\ q\ a \to f(\underbrace{q\ (g^*\ a) \to a}_{\bigcirc}) \to g^*\ a$$

So what?

What do we gain?

So what?

What do we gain?

Practice

- MTTs as a meta programming framework
- composition and manipulation of MTTs in a structured manner

So what?

What do we gain?

Practice

- MTTs as a meta programming framework
- composition and manipulation of MTTs in a structured manner

Theory

- more elegant proofs of compositionality results (using parametricity and fold fusion)
- monadic MTTs: generalisation of non-deterministic / partial MTTs

Conclusion

Implemented in the compositional data types library:

```
> cabal install compdata
```

Bonus Slide: Definition of Macro Tree Transducers

$$q(f(x_1, \ldots, x_n), y_1, \ldots, y_m) \to u$$

for each $f/n \in \mathcal{F}$
and $q/(m+1) \in Q$

Bonus Slide: Definition of Macro Tree Transducers

$$q(f(x_1, \ldots, x_n), y_1, \ldots, y_m) \rightarrow u$$

for each $f/n \in \mathcal{F}$
and $q/(m+1) \in Q$

Where $u \in RHS_{n,m}$, which is defined as follows:

$$\frac{1 \leq i \leq m}{y_i \in RHS_{n,m}} \qquad \frac{g/k \in \mathcal{G} \quad u_1, \ldots, u_k \in RHS_{n,m}}{g(u_1, \ldots, u_k) \in RHS_{n,m}}$$

$$\frac{1 \leq i \leq n \quad q'/(k+1) \in Q \quad u_1, \ldots, u_k \in RHS_{n,m}}{q'(x_i, u_1, \ldots, u_k) \in RHS_{n,m}}$$