Programming Macro Tree Transducers

Patrick Bahr¹  Laurence E. Day²

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

²University 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

\[\text{tree transducer} = \text{set of mutually recursive functions}\]
Macro Tree Transducers on One Slide

Tree Transducers in FP
- automaton transforming trees to trees
- states are interpreted as functions
  ⇝ 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 $\leadsto$ 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 $\mapsto$ 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 $\mapsto$ meta programming

How so?

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

word
Finite State Automata – On Strings

$q_0 \xrightarrow{\text{word}} q_4 \in Q_F$
Finite State Automata – On Strings

$q_0 \xrightarrow{\text{word}, s} q'$
Finite State Automata – On Strings

\[ q_0 \xrightarrow{\text{word}} q_1 \in Q_F \]

\[ q, s \rightarrow q' \]
Finite State Automata – On Strings

$q_0 \xrightarrow{w} q_1 \xrightarrow{o} q_2 \xrightarrow{r} \xrightarrow{d}

q, s \rightarrow q'$
Finite State Automata – On Strings

\[ q_0 \quad w \quad q_1 \quad o \quad q_2 \quad r \quad q_3 \quad d \]

\[ q, s \rightarrow q' \]
Finite State Automata – On Strings

$w \in \mathcal{A}$

$q_0 \rightarrow q_1 \rightarrow q_2 \rightarrow q_3 \rightarrow q_4$

$q, s \rightarrow q'$
Finite State Automata – On Strings

\[ q_0 \xrightarrow{\text{w}} q_1 \xrightarrow{\text{o}} q_2 \xrightarrow{\text{r}} q_3 \xrightarrow{\text{d}} q_4 \in Q_F? \]

\[ q, s \rightarrow q' \]
Finite State Automata – On Strings

\[ q_0 \xrightarrow{\text{w}} q_1 \xrightarrow{\text{o}} q_2 \xrightarrow{\text{r}} q_3 \xrightarrow{\text{d}} q_4 \in Q_F? \]

Acceptor

\[ q, s \rightarrow q' \]
Finite State Automata – On Strings

Transducer?

$q, s \rightarrow q'$
Finite State Automata – On Strings

Transducer

\[
q, s \rightarrow q', w
\]
Finite State Automata – On Strings

Transducer

\[ q, s \rightarrow q', w \]
Finite State Automata – On Strings

Transducer

$$q, s \rightarrow q', w$$
Finite State Automata – On Strings

new_word

q₀ → q₁
w → q₂
ε → q₃
ord → q₄

Transducer

q, s → q', w
Now on Trees!

Often rendered as a rewrite rule:

\[ q(f(x_1, \ldots, x_n)) \rightarrow f(q_1(x_1), \ldots, q_n(x_n)) \]
Now on Trees!

$q_0$

not

and

not or

not tt b

$\rightarrow$
Now on Trees!

\[
\begin{align*}
q_0 & \quad \text{not} \\
q_1 & \quad \text{and} \\
& \quad \text{not} \\
& \quad \text{or} \\
& \quad b \\
& \quad \text{tt} \\
& \quad b
\end{align*}
\]

Often rendered as a rewrite rule:

\[
f(x_1, \ldots, x_n) \rightarrow f(q_1(x_1), \ldots, q_n(x_n))
\]
Now on Trees!

\[
q_0 
\]

\[
\text{not} 
\]

\[
q_1 
\]

\[
\text{and} 
\]

\[
q_2 
\]

\[
q_3 
\]

\[
\text{not} 
\]

\[
\text{or} 
\]

\[
b 
\]

\[	t 
\]

\[
b 
\]
Now on Trees!

```

```

often rendered as a rewrite rule:

```
q(f(x1, ... , xn)) \rightarrow f(q1(x1), ... , qn(xn))
```

Patrick Bahr, Laurence E. Day — Programming Macro Tree Transducers — WGP '13, 28th September, Boston, Massachusetts
Now on Trees!

Often rendered as a rewrite rule:

\[ q, f \rightarrow q_1, \ldots, q_n \]
Now on Trees!

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

\[ q(f(x_1, \ldots, x_n)) \rightarrow f(q_1(x_1), \ldots, q_n(x_n)) \]
Tree Transducers

\[ q(f(x_1, \ldots, x_n)) \to f(q_1(x_1), \ldots, q_n(x_n)) \]
Tree Transducers

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

\[ q \]

\[ f \]

\[ q' \quad q'' \]
And now in Haskell

\[ \forall a . \ (q, f a) \rightarrow g^* (q, a) \]

Representation in Haskell
And now in Haskell

Representation in Haskell

\[ \text{type } \mathit{Trans}_D f q g = \forall a . (q, f a) \rightarrow g^*(q, a) \]

Free Monad of a Functor \(g\)

\[ \text{data } g^* a = \text{Re } a \mid \text{ln } (g (g^* a)) \]
Example: Substitution

\[
\text{type } Var = String \\
\text{data } Sig a = Add a a | Val Int | Let Var a a | Var Var
\]
Example: Substitution

type Var = String

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

trans subst :: TransD Sig (Map Var (μSig)) Sig

trans subst (m, Var v) = case Map.lookup v m of
  Nothing → iVar v
  Just t → toFree t

trans subst (m, Let v b s) = iLet v (Re (m, b))
  (Re (m \ v, s))

trans subst (m, Val n) = iVal n

trans subst (m, Add x y) = Re (m, x) 'iAdd' Re (m, y)
Example: Substitution

type Var = String

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

trans subst :: TransD Sig (Map Var (µSig)) Sig

trans subst (m, Var v) = case Map.lookup v m of
  Nothing → iVar v
  Just t → toFree t

trans subst (m, Let v b s) = iLet v (Re (m, b))
  (Re (m \ v, s))

trans subst (m, Val n) = iVal n
trans subst (m, Add x y) = Re (m, x) ‘iAdd’ Re (m, y)

type µf = f* Empty
Example: Substitution

type Var = String

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

type Trans_D f q g = ∀a . (q, f a) → g*(q, a)

trans subst :: Trans_D Sig (Map Var (µSig)) Sig

trans subst (m, Var v) = case Map.lookup v m of
  Nothing → iVar v
  Just t → toFree t

trans subst (m, Let v b s) = iLet v (Re (m, b))
                          (Re (m \ v, s))

trans subst (m, Val n) = iVar n

trans subst (m, Add x y) = Re (m, x) 'iAdd' Re (m, y)
Example: Substitution

```haskell
type Var = String
data Sig a = Add a a | Val Int | Let Var a a | Var Var

trans_subst :: TransD Sig (Map Var (\muSig)) Sig
trans_subst (m, Var v) = case Map.lookup v m of
  Nothing → iVar v
  Just t → toFree t
trans_subst (m, Let v b s) = iLet v (Re (m, b))
                           (Re (m \ v, s))
trans_subst (m, Val n) = iVal n
trans_subst (m, Add x y) = Re (m, x) 'iAdd' Re (m, y)

subst :: Map Var \muSig → \muSig → \muSig
subst = \trans_subst\D
```

Patrick Bahr, Laurence E. Day — Programming Macro Tree Transducers — WGP ’13, 28th September, Boston, Massachusetts
Non-Example: Inlining

\[\text{trans}_{\text{inline}} :: \text{Trans}_D \, \text{Sig} \, (\text{Map} \, \text{Var} \, \mu \text{Sig}) \, \text{Sig}\]

\[\text{trans}_{\text{inline}} (m, \text{Var} \, v) = \text{case} \, \text{Map}.\text{lookup} \, v \, m \, \text{of}\]

\[\text{Nothing} \rightarrow \text{iVar} \, v \]
\[\text{Just} \, e \rightarrow \text{toFree} \, e\]

\[\text{trans}_{\text{inline}} (m, \text{Let} \, v \, b \, s) = \text{Re} \, (m \, [v \mapsto b], s)\]

\[\text{trans}_{\text{inline}} (m, \text{Val} \, n) = \text{iVal} \, n\]

\[\text{trans}_{\text{inline}} (m, \text{Add} \, x \, y) = \text{Re} \, (m, x) \, 'iAdd' \, \text{Re} \, (m, y)\]

\[\text{inline} :: \mu \text{Sig} \rightarrow \mu \text{Sig}\]

\[\text{inline} = \left[\text{trans}_{\text{inline}}\right]_D \, \emptyset\]
Non-Example: Inlining

\[
\begin{align*}
\text{trans}_{\text{inline}} &:: Trans_D \ Sig \ (\text{Map Var } \mu \text{Sig}) \ \text{Sig} \\
\text{trans}_{\text{inline}} (m, \text{Var } v) &= \text{case} \ \text{Map.lookup } v \ m \ \text{of} \\
&\quad \quad \text{Nothing} \rightarrow \text{iVar } v \\
&\quad \quad \text{Just } e \quad \rightarrow \text{toFree } e \\
\text{trans}_{\text{inline}} (m, \text{Let } v \ b \ s) &= \text{Re } (m [v \mapsto b], s) \\
\text{trans}_{\text{inline}} (m, \text{Val } n) &= \text{iVal } n \\
\text{trans}_{\text{inline}} (m, \text{Add } x \ y) &= \text{Re } (m, x) \ 'iAdd' \ \text{Re } (m, y) \\
\text{inline} &:: \mu \text{Sig} \rightarrow \mu \text{Sig} \\
\text{inline} &= \bigl[\text{trans}_{\text{inline}}\bigr]_D \ \emptyset
\end{align*}
\]

Recall the type \textit{Trans}_D

\[
\text{type } Trans_D \ f \ q \ g = \forall a \ . \ (q, f a) \rightarrow g^*(q, a)
\]
Transducers with Polymorphic State Space

The original type $\text{Trans}_D$

$$\text{type } \text{Trans}_D \ f \ q \ g = \forall a . (q, f a) \rightarrow g^*(q, a)$$
Transducers with Polymorphic State Space

The original type $Trans_D$

\[
\text{type } Trans_D \ f \ q \ g = \forall a . (q, f \ a) \rightarrow g^*(q, a)
\]

An equivalent representation

\[
\text{type } Trans_D \ f \ q \ g = \forall a. q \rightarrow f \quad a \rightarrow g^*(q, a)
\]
Transducers with Polymorphic State Space

The original type $\mathit{Trans}_D$

```
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^* a
```
Transducers with Polymorphic State Space

The original type $\text{Trans}_D$

```
type Trans_D f q g = ∀a.(q, f a) → g*(q, a)
```

An equivalent representation

```
type Trans_D f q g = ∀a.q → f(q → a) → g* a
```

Deriving the type $\text{Trans}_M$

```
type Trans_M f q g = ∀a.q a → f(q a → a) → g* a
```
Transducers with Polymorphic State Space

The original type $\text{Trans}_D$

\[
\text{type } \text{Trans}_D \ f \ q \ g = \forall a . (q, f a) \to g^*(q, a)
\]

An equivalent representation

\[
\text{type } \text{Trans}_D \ f \ q \ g = \forall a . q \to f(q \to a) \to g^* a
\]

Deriving the type $\text{Trans}_M$

\[
\text{type } \text{Trans}_M \ f \ q \ g = \forall a . q a \to f(q (g^* a) \to a) \to g^* a
\]
Example: Inlining

\[
\text{trans}_{\text{inline}} :: \ Trans^I_{\mathcal{M}} \ Sig \ (\text{Map Var}) \ Sig
\]

\[
\text{trans}_{\text{inline}} \ m \ (\text{Var } v) = \text{case Map.lookup } v \ m \ \text{of}
\]
\[
\text{Nothing} \rightarrow \text{iVar } v
\]
\[
\text{Just } e \times \rightarrow e
\]

\[
\text{trans}_{\text{inline}} \ m \ (\text{Let } v \ b \ s) = s \ (m[v \mapsto b \ m])
\]

\[
\text{trans}_{\text{inline}} \ m \ (\text{Val } n) = \text{iVal } n
\]

\[
\text{trans}_{\text{inline}} \ m \ (\text{Add } x \ y) = x \ m \ \text{‘iAdd‘} \ y \ m
\]
Example: Inlining

\[
\text{trans}_{\text{inline}} :: \text{Trans}_M \rightarrow \text{Map Var} \rightarrow \text{Sig} \\
\text{trans}_{\text{inline}} m (\text{Var } v) = \text{case Map.lookup } v m \text{ of} \\
\quad \text{Nothing} \rightarrow \text{iVar } v \\
\quad \text{Just } e \rightarrow e \\
\text{trans}_{\text{inline}} m (\text{Let } v b s) = s (m[v \mapsto b m]) \\
\text{trans}_{\text{inline}} m (\text{Val } n) = \text{iVal } n \\
\text{trans}_{\text{inline}} m (\text{Add } x y) = x m \text{ 'iAdd' } y m
\]

\[
\text{inline} :: \mu \text{Sig} \rightarrow \mu \text{Sig} \\
\text{inline} = \left[ \text{trans}_{\text{inline}} \right]_M \emptyset
\]
Macro Tree Transduction Rule Illustrated

\[ \text{type } \text{Trans}_M f \ q \ g = \forall a. \ q \ (g^* a) \rightarrow f \ (q \ (g^* a) \rightarrow a) \rightarrow g^* a \]
Macro Tree Transduction Rule Illustrated

\[
\text{type } \text{Trans}_M \ f \ q \ g = \forall a. q \ a \to f(q (g^* a) \to a) \to g^* a
\]
Macro Tree Transduction Rule Illustrated

\[
\text{type } \text{Trans}_M f \ q \ g = \forall \ a. \ q \ a \rightarrow f(q \ (g^* a) \rightarrow a) \rightarrow g^* a
\]
Macro Tree Transduction Rule Illustrated

\[
\text{type } Trans_M f q g = \forall a. \ q a \to f(q (g^* a) \to a) \to g^* a
\]
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

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

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:

\[
\begin{align*}
1 \leq i \leq m & \quad g/k \in G & u_1, \ldots, u_k \in RHS_{n,m} \\
y_i \in RHS_{n,m} & \quad g(u_1, \ldots, u_k) \in RHS_{n,m}
\end{align*}
\]

\[
1 \leq i \leq n \quad q'(k + 1) \in Q & \quad u_1, \ldots, u_k \in RHS_{n,m} \\
q'(x_i, u_1, \ldots, u_k) \in RHS_{n,m}
\]