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

NB: This breaks locality and has to be carefully restricted!

But it is convenient/necessary for compositionality and expressivity.
How do we achieve these goals?

<table>
<thead>
<tr>
<th>Locality</th>
<th>simple syntax-directed functions are local in nature</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compositionality</td>
<td>syntax-directed functions can be combined and composed</td>
</tr>
</tbody>
</table>
How do we achieve these goals?

<table>
<thead>
<tr>
<th>Locality</th>
<th>simple syntax-directed functions are local in nature</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compositionality</td>
<td>syntax-directed functions can be combined and composed</td>
</tr>
<tr>
<td>Contextuality</td>
<td>syntax-directed functions may depend on (the result of) others</td>
</tr>
</tbody>
</table>
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.

\[
f(q_1(x_1), q_2(x_2), \ldots, q_n(x_n)) \rightarrow t[x_1, x_2, \ldots, x_n]
\]
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.

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

\[
\mu F_1 \xrightarrow{[A_1]} \mu F_2 \xrightarrow{[A_2]} \mu F_3
\]

input signature: the type of the AST

\[
\mu F_1 : \mu F \rightarrow R
\]

\[
\mu F_2 : \mu F \rightarrow \mathbb{S}
\]

output type: tupling / product automaton construction

\[
\mu F_1 \times \mu F_2 : \mu F \rightarrow R \times \mathbb{S}
\]
Compositionality
We shall compose tree automata along 3 different dimensions.

**sequential composition:** a.k.a. deforestation

\[
\begin{align*}
\mu F_1 & \xrightarrow{[A_1]} \mu F_2 \xrightarrow{[A_2]} \mu F_3 \\
[ A_1 \circ A_2 ] & 
\end{align*}
\]
Compositionality

We shall compose tree automata along 3 different dimensions.

**sequential composition:** a.k.a. deforestation

\[
\mu F_1 \rightarrow [A_1] \rightarrow \mu F_2 \rightarrow [A_2] \rightarrow \mu F_3
\]

\[
[A_1 \circ A_2]
\]

**input signature:** the type of the AST

\[
[A_1] : \mu F \rightarrow R
\]

\[
[A_2] : \mu G \rightarrow R
\]
Compositionality
We shall compose tree automata along 3 different dimensions.

**Sequential composition:** a.k.a. deforestation

\[
\begin{align*}
\mu F_1 & \xrightarrow{[A_1]} \mu F_2 \xrightarrow{[A_2]} \mu F_3 \\
[A_1 \circ A_2] & \quad \\
\end{align*}
\]

**Input signature:** the type of the AST

\[
\begin{align*}
[A_1] : \mu F & \to R \\
[A_2] : \mu G & \to R \\
\implies & \\
[A_1 + A_2] : \mu (F + G) & \to R
\end{align*}
\]
Compositionality

We shall compose tree automata along 3 different dimensions.

**sequential composition**: a.k.a. deforestation

\[
\begin{align*}
\mu \mathcal{F}_1 & \xrightarrow{[\mathcal{A}_1]} \mu \mathcal{F}_2 & \xrightarrow{[\mathcal{A}_2]} \mu \mathcal{F}_3 \\
& \Rightarrow [\mathcal{A}_1 \circ \mathcal{A}_2]
\end{align*}
\]

**input signature**: the type of the AST

\[
\begin{align*}
[\mathcal{A}_1] : \mu \mathcal{F} & \rightarrow R \\
[\mathcal{A}_2] : \mu \mathcal{G} & \rightarrow R
\end{align*}
\Rightarrow

[\mathcal{A}_1 + \mathcal{A}_2] : \mu (\mathcal{F} + \mathcal{G}) & \rightarrow R
\]

**output type**: tupling / product automaton construction

\[
\begin{align*}
[\mathcal{A}_1] : \mu \mathcal{F} & \rightarrow R \\
[\mathcal{A}_2] : \mu \mathcal{F} & \rightarrow S
\end{align*}
\]
Compositionality
We shall compose tree automata along 3 different dimensions.

**sequential composition:** a.k.a. deforestation

\[ \mu F_1 \xrightarrow{[A_1]} \mu F_2 \xrightarrow{[A_2]} \mu F_3 \]

\[ \mu F_1 \mu F_2 \xrightarrow{[A_1 \circ A_2]} \]

**input signature:** the type of the AST

\[ [A_1] : \mu F \rightarrow R \]
\[ [A_2] : \mu G \rightarrow R \]

\[ \Rightarrow \]

\[ [A_1 + A_2] : \mu (F + G) \rightarrow R \]

**output type:** tupling / product automaton construction

\[ [A_1] : \mu F \rightarrow R \]
\[ [A_2] : \mu F \rightarrow S \]

\[ \Rightarrow \]

\[ [A_1 \times A_2] : \mu F \rightarrow R \times S \]
Contextuality

tupling / product automaton construction

\[ [A_1] : \mu F \to R \]
\[ [A_2] : \mu F \to S \]

\[ \implies [A_1 \times A_2] : \mu(F) \to R \times S \]
## Contextuality

### Tupling / Product Automaton Construction

| $A_1 : \mathcal{F} \rightarrow R$ | $A_2 : \mathcal{F} \rightarrow S$ | $\Rightarrow$ | $A_1 \times A_2 : \mathcal{F} \rightarrow R \times S$ |
Contextuality

tupling / product automaton construction

\[ A_1 : \mathcal{F} \to R \]
\[ A_2 : \mathcal{F} \to S \]
\[ \implies A_1 \times A_2 : \mathcal{F} \to R \times S \]

mutumorphisms / dependent product automata

\[ A_1 : \mathcal{F} \to R \]
\[ A_2 : R \Rightarrow \mathcal{F} \to S \]
## Contextuality

### Tupling / Product Automaton Construction

<table>
<thead>
<tr>
<th>Automaton $A_1$</th>
<th>$F \rightarrow R$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Automaton $A_2$</td>
<td>$F \rightarrow S$</td>
</tr>
</tbody>
</table>

$A_1 \times A_2 : F \rightarrow R \times S$

### Mutationmorphisms / Dependent Product Automata

<table>
<thead>
<tr>
<th>Automaton $A_1$</th>
<th>$F \rightarrow R$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Automaton $A_2$</td>
<td>$R \Rightarrow F \rightarrow S$</td>
</tr>
</tbody>
</table>

$A_1 \times A_2 : F \rightarrow R \times S$
Contextuality

**tupling / product automaton construction**

\[ A_1 : \mathcal{F} \to R \]
\[ A_2 : \mathcal{F} \to S \]

\[ \implies \quad A_1 \times A_2 : \mathcal{F} \to R \times S \]

**mutumorphisms / dependent product automata**

\[ A_1 : S \Rightarrow \mathcal{F} \to R \]
\[ A_2 : R \Rightarrow \mathcal{F} \to S \]

\[ \implies \quad A_1 \times A_2 : \mathcal{F} \to R \times S \]
Outline

1. Introduction

2. State Transition Functions
   - Composing State Spaces
   - Compositional Signatures

3. Tree Transducers
   - Bottom-Up Tree Transducers
   - Decomposing Tree Transducers

4. Conclusions
Terms in Haskell

Data types as fixed points of functors

```
data Term f = In (f (Term f))
```
Terms in Haskell

Data types as fixed points of functors

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

Functors

class Functor f where
  fmap :: (a → b) → f a → f b
Bottom-Up State Transitions in Haskell

\[
\begin{align*}
\text{type } & \text{UpState } f \ q = f \ q \\
\text{runUpState} & : \text{Functor } f \Rightarrow \text{UpState } f \ q \to \text{Term } f \to q
\end{align*}
\]

\[
\text{runUpState } \phi \ (\text{In } t) = \phi (\text{fmap } (\text{runUpState } \phi) t)
\]
Bottom-Up State Transitions in Haskell

Bottom-up state transition rules as algebras

\[
\text{type} \quad \text{UpState} f q = f q \rightarrow q
\]

\[
\text{runUpState} :: \text{Functor} f \Rightarrow \text{UpState} f q \rightarrow \text{Term} f \rightarrow q
\]

\[
\text{runUpState} \varphi (\text{In} \, t) = \varphi (fmap (\text{runUpState} \varphi) t)
\]
Bottom-Up State Transitions in Haskell

Bottom-up state transition rules as algebras

\[
\text{type } \text{UpState } f \ q = f \ q \to q
\]

\[
\text{runUpState} \ : \ \text{Functor } f \Rightarrow \text{UpState } f \ q \to \text{Term } f \to q
\]

\[
\text{runUpState} \ \phi \ ((\text{In } t)) = \phi ((\text{fmap } (\text{runUpState} \ \phi) \ t))
\]
Bottom-Up State Transitions in Haskell

Bottom-up state transition rules as algebras

\[ \text{type } \textit{UpState} \, f \, q = f \, q \rightarrow q \]
Bottom-Up State Transitions in Haskell

Bottom-up state transition rules as algebras

\[
\text{type } \text{UpState } f \ q = f \ q \to q
\]

\[
\text{runUpState } :: \text{Functor } f \Rightarrow \text{UpState } f \ q \to \text{Term } f \to q
\]

\[
\text{runUpState } \phi (\text{In } t) = \phi (\text{fmap (runUpState } \phi) t)
\]
Bottom-Up State Transitions in Haskell

Bottom-up state transition rules as algebras

\[
\text{type } \text{UpState } f \ q = f \ q \rightarrow q
\]

\[
\text{runUpState} :: \text{Functor } f \Rightarrow \text{UpState } f \ q \rightarrow \text{Term } f \rightarrow q
\]

\[
\text{runUpState } \phi \ (\text{In } t) = \phi \ (\text{fmap} \ (\text{runUpState } \phi) \ t)
\]
A simple expression language

data Sig e = Val Int | Plus e e
Composing State Spaces – Motivating Example

A simple expression language

```haskell
data Sig e = Val Int | Plus e e
```

Task: writing a code generator

```haskell
type Addr = Int
data Instr = Acc Int | Load Addr | Store Addr | Add Addr
type Code = [Instr]
```
A simple expression language

```haskell
data Sig e = Val Int | Plus e e
```

Task: writing a code generator

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

The problem

```haskell
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]
```

The problem

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

\[Sig \rightarrow Code\]
Tupling

Tuple the code with an address counter

\[
\text{codeAddrSt :: UpState Sig (Code, Addr)} \\
\text{codeAddrSt (Val i)} = ([\text{Acc } i], 0) \\
\text{codeAddrSt (Plus (x, a') (y, a))} = (x + \text{[Store } a\text{]} + y + \text{[Add } a\text{]}, 1 + \text{max } a \text{ a'})
\]
Tupling

Tuple the code with an address counter

\[
\text{codeAddrSt} :: \text{UpState Sig} (\text{Code, Addr}) \\
\text{codeAddrSt} (\text{Val } i) = ([\text{Acc } i], 0) \\
\text{codeAddrSt} (\text{Plus } (x, a') (y, a)) = (x + [\text{Store } a] + y + [\text{Add } a], 1 + \max a a')
\]

Run the automaton

\[
\text{code} :: \text{Term Sig} \rightarrow (\text{Code, Addr}) \\
\text{code} = \text{runUpState codeAddrSt}
\]
**Tupling**

Tuple the code with an address counter

\[
\text{codeAddrSt} :: \text{UpState Sig (Code, Addr)}
\]

\[
\text{codeAddrSt (Val i)} = ([\text{Acc } i], 0)
\]

\[
\text{codeAddrSt (Plus (x, a') (y, a))} = (x \uplus [\text{Store a}] \uplus y \uplus [\text{Add a}], 1 + \max a a')
\]

Run the automaton

\[
\text{code :: Term Sig} \rightarrow (\text{Code, Addr})
\]

\[
\text{code} = \text{fst . runUpState codeAddrSt}
\]
Tupling

Tuple the code with an address counter

\[
\begin{align*}
\text{codeAddrSt} :: \text{UpState Sig} & (\text{Code}, \text{Addr}) \\
\text{codeAddrSt} (\text{Val} i) & = ([\text{Acc} i], 0) \\
\text{codeAddrSt} (\text{Plus} (x, a') (y, a)) & = (x + [\text{Store} a] + y + [\text{Add} a], \\
& \quad 1 + \max a a')
\end{align*}
\]

Run the automaton

\[
\begin{align*}
\text{code} :: \text{Term Sig} & \rightarrow \text{Code} \\
\text{code} & = \text{fst} \ . \ \text{runUpState} \ \text{codeAddrSt}
\end{align*}
\]
Product Automata

<table>
<thead>
<tr>
<th>Deriving projections</th>
</tr>
</thead>
<tbody>
<tr>
<td>class $a \in b$ where</td>
</tr>
<tr>
<td>$pr :: b \to a$</td>
</tr>
</tbody>
</table>
# Product Automata

## Deriving projections

```haskell
class a ∈ b where
  pr :: b → a
dec a ∈ b iff
  - b is of the form \((b_1, (b_2, ..., b_n))\) and
  - \(a = b_i\) for some \(i\)
```

For example:

- \(\text{Addr} \in (\text{Code}, \text{Addr})\)

## Dependent state transition functions

- \(\text{UpState } f q = f q \rightarrow q\)
- \(\text{DUpState } f p q = (q \in p) \Rightarrow f p \rightarrow q\)

## Product state transition

\(\odot :: (p \in c, q \in c) \Rightarrow \text{DUpState } f c p \rightarrow \text{DUpState } f c q \rightarrow \text{DUpState } f c (p, q)\)

\((s \odot s') t = (s t, s' t)\)
Product Automata

Deriving projections

\[
\text{class } a \in b \text{ where } \\
pr :: b \rightarrow a
\]

\[a \in b \quad \text{iff} \]
- \[b \text{ is of the form } (b_1, (b_2, \ldots, b_n)) \text{ and} \]
- \[a = b_i \text{ for some } i \]

For example: \(Addr \in (Code, Addr)\)
Product Automata

Deriving projections

```
class a ∈ b where
  pr :: b → a
```

\[ a ∈ b \text{ iff } b \text{ is of the form } (b_1, (b_2, \ldots b_n)) \text{ and } a = b_i \text{ for some } i \]

For example: \( \text{Addr} ∈ (\text{Code, Addr}) \)

Dependent state transition functions

```
type UpState f q = f q → q
```

Product Automata

Deriving projections

class \( a \in b \) where 
\[ \text{pr} :: b \to a \]
\( a \in b \) 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

```haskell
class a ∈ b where
    a ∈ b  iff
    pr :: b → a

    • b is of the form (b₁, (b₂, ... bₙ)) and
    • a = bᵢ for some i
```

For example: Addr ∈ (Code, Addr)

Dependent state transition functions

```haskell
type UpState f q = f q → q

type DUPState f p q = (q ∈ p) ⇒ f p → q
```
**Product Automata**

### Deriving projections

```haskell
class a ∈ b where
    pr :: b → a

pr ∈ 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

```haskell
type UpState f q = f q → q

type DUpState f p q = (q ∈ p) ⇒ f p → q
```
Product Automata

Deriving projections

\textbf{class} \( a \in b \ \text{where} \)
\[
\text{pr} :: b \to a
\]

\( a \in b \) \ iff
\begin{itemize}
\item \( b \) is of the form \((b_1, (b_2, \ldots b_n))\) and
\item \( a = b_i \) for some \( i \)
\end{itemize}

For example: \( \text{Addr} \in (\text{Code}, \text{Addr}) \)

Dependent state transition functions

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

Product state transition

\((\otimes) :: (p \in c, q \in c) \Rightarrow \text{DUpState} \ f \ c \ p \to \text{DUpState} \ f \ c \ q \)
\[
\Rightarrow \text{DUpState} \ f \ c \ (p, q)
\]
\[
(sp \otimes sq) \ t = (sp \ t, sq \ t)
\]
Running Dependent State Transition Functions

The types

\[
\begin{align*}
\text{type } UpState & \quad f \ q \ = \quad f \ q \rightarrow q \\
\text{type } DU\text{UpState} & \quad f \ p \ q \ = \ (q \in p) \Rightarrow f \ p \rightarrow q
\end{align*}
\]
Running Dependent State Transition Functions

The types

\[
\text{type} \ UpState \ f \ q = f \ q \to q \\
\text{type} \ DUpState \ f \ p \ q = (q \in p) \Rightarrow f \ p \to q
\]

Running dependent state transitions

\[
\text{runDUpState} :: \text{Functor} \ f \Rightarrow DUpState \ f \ q \ q \to \text{Term} \ f \to q \\
\text{runDUpState} \ f = \text{runUpState} \ f
\]
Running Dependent State Transition Functions

The types

\[
\begin{align*}
\text{type } \text{UpState} \ f \ q &= f \ q \to q \\
\text{type } \text{DUpState} \ f \ p \ q &= (q \in p) \Rightarrow f \ p \to q
\end{align*}
\]

Running dependent state transitions

\[
\text{runDUpState} :: \text{Functor } f \Rightarrow \text{DUpState } f \ q \ q \to \text{Term } f \to q
\]
\[
\text{runDUpState } f = \text{runUpState } f
\]

From state transition to dependent state transition

\[
\text{dUpState} :: \text{Functor } f \Rightarrow \text{UpState } f \ q \rightarrow \text{DUpState } f \ p \ q
\]
\[
\text{dUpState } st = st . \text{fmap } \text{pr}
\]
The Code Generator Example

The code generator

\[
\text{codeSt :: (Int} \in \mathbb{q}) \Rightarrow \text{DupState} \ \text{Sig} \ q \ \text{Code} \\
\text{codeSt (Val} \ i) = [\text{Acc} \ i] \\
\text{codeSt (Plus} \ x \ y) = \text{pr} \ x \ + + [\text{Store} \ a] \ + + \text{pr} \ y \ + + [\text{Add} \ a] \\
\text{where} \ a = \text{pr} \ y
\]
The Code Generator Example

The code generator

\[
\text{codeSt} :: (\text{Int} \in q) \Rightarrow \text{DUpState Sig q Code}
\]

\[
\text{codeSt} (\text{Val} \ i) = [\text{Acc} \ i]
\]

\[
\text{codeSt} (\text{Plus} \ x \ y) = \text{pr} \ x \ \mathbf{++} \ [\text{Store} \ a] \ \mathbf{++} \ \text{pr} \ y \ \mathbf{++} \ [\text{Add} \ a]
\]

\textbf{where} \ a = \text{pr} \ y

Generating fresh addresses

\[
\text{heightSt} :: \text{UpState Sig Int}
\]

\[
\text{heightSt} (\text{Val} \ _) = 0
\]

\[
\text{heightSt} (\text{Plus} \ x \ y) = 1 + \text{max} \ x \ y
\]
The Code Generator Example

The code generator

\[\text{codeSt} :: (\text{Int} \in \text{q}) \Rightarrow \text{DUpState Sig} \text{ q Code} \]
\[\text{codeSt} (\text{Val} \ i) = [\text{Acc} \ i] \]
\[\text{codeSt} (\text{Plus} \ x \ y) = \text{pr} \ x \ !+! [\text{Store} \ a] \ !+! \text{pr} \ y \ !+! [\text{Add} \ a] \]
\[\text{where} \ a = \text{pr} \ y \]

Generating fresh addresses

\[\text{heightSt} :: \text{UpState Sig} \text{ Int} \]
\[\text{heightSt} (\text{Val} \ _) = 0 \]
\[\text{heightSt} (\text{Plus} \ x \ y) = 1 + \text{max} \ x \ y \]

Combining the components

\[\text{code} :: \text{Term Sig} \rightarrow \text{Code} \]
\[\text{code} = \text{fst} . \text{runUpState} (\text{codeSt} \otimes \text{dUpState heightSt}) \]
The Code Generator Example

The code generator

\[\text{codeSt} :: (\text{Int} \in q) \Rightarrow \text{DUpState} \ \text{Sig} \ q \ \text{Code}\]
\[\text{codeSt} (\text{Val} \ i) = [\text{Acc} \ i]\]
\[\text{codeSt} (\text{Plus} \ x \ y) = \text{pr} \ x + + [\text{Store} \ a] + + \text{pr} \ y + + [\text{Add} \ a]\]
where \( a = \text{pr} \ y \)

Generating fresh addresses

\[\text{heightSt} :: \text{UpState} \ \text{Sig} \ \text{Int}\]
\[\text{heightSt} (\text{Val} \ _) = 0\]
\[\text{heightSt} (\text{Plus} \ x \ y) = 1 + + \max \ x \ y\]

Combining the components

\[\text{code} :: \text{Term} \ \text{Sig} \rightarrow \text{Code}\]
\[\text{code} = \text{fst} \ . \ \text{runUpState} \ (\text{codeSt} \ \otimes \ \text{dUpState} \ \text{heightSt})\]
Outline

1. Introduction

2. State Transition Functions
   - Composing State Spaces
   - Compositional Signatures

3. Tree Transducers
   - Bottom-Up Tree Transducers
   - Decomposing Tree Transducers

4. Conclusions
Combining Signatures

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

Coproduct of signatures

data (f ⊕ g) e = Inl (f e) | Inr (g e)

Example
data Inc e = Inc e
type Sig′ = Inc ⊕ Sig

Subsignature type class
class f ⪯ g where inj :: f a → g a

For example:
Inc ⪯ Sig′

f ⪯ g iff g = g₁ ⊕ g₂ ⊕ ... ⊕ gₙ and f = gᵢ, 0 < i ≤ n
Combining Signatures

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

Coproduct of signatures

\[
\text{data } (f \oplus g) \ e = \text{Inl } (f \ e) \mid \text{Inr } (g \ e)
\]
Combining Signatures

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

**Coproduct of signatures**

\[
\text{data} \ (f \oplus g) \ e = \text{Inl} \ (f \ e) \mid \text{Inr} \ (g \ e)
\]

**Example**

\[
\text{data} \ \text{Inc} \ e = \text{Inc} \ e \\
\text{type} \ \text{Sig}' = \text{Inc} \oplus \text{Sig}
\]
Combining Signatures

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

Coproduct of signatures

```
data (f ⊕ g) e = Inl (f e) | Inr (g e)
```

Example

```
data Inc e = Inc e
type Sig' = Inc ⊕ Sig
```

Subsignature type class

```
class f ⪯ g where
    inj :: f a → g a
```
Combining Signatures

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

Coproduct of signatures

\[
\text{data} \ (f \oplus g) \ e = \text{Inl} \ (f \ e) \mid \text{Inr} \ (g \ e)
\]

Example

\[
\text{data} \ \text{Inc} \ e = \text{Inc} \ e \\
\text{type} \ Sig' = \text{Inc} \oplus Sig
\]

Subsignature type class

\[
\text{class} \ f \preceq g \ \text{where} \inj :: f \ a \rightarrow g \ a
\]

\[
f \preceq g \ \text{iff} \\
\quad \bullet \ g = g_1 \oplus g_2 \oplus \ldots \oplus g_n \ \text{and} \\
\quad \bullet \ f = g_i, \quad 0 < i \leq n
\]
Combining Signatures

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

Coproduct of signatures

\[
\text{data } (f \oplus g) \ e = \text{lnl } (f \ e) \mid \text{lnr } (g \ e)
\]

Example

\[
\text{data } \text{Inc } e = \text{Inc } e
\]
\[
\text{type } \text{Sig}' = \text{Inc } \oplus \text{Sig}
\]

Subsignature type class

\[
\text{class } f \preceq g \text{ where
}
\text{inj :: } f \ a \rightarrow g \ a
\]

For example: \text{Inc } \preceq \text{Sig}'

\[f \preceq g \text{ iff}
\]
- \[g = g_1 \oplus g_2 \oplus \ldots \oplus g_n \text{ and}
\]
- \[f = g_i, \quad 0 < i \leq n\]
Combining Automata

Making the height compositional

```haskell
class HeightSt f where
    heightSt :: DUPState f q Int

instance (HeightSt f, HeightSt g) ⇒ HeightSt (f ⊕ g) where
    heightSt (Inl x) = heightSt x
    heightSt (Inr x) = heightSt x
```

Defining the height on Sig

```haskell
instance HeightSt Sig where
    heightSt (Val) = 0
    heightSt (Plus x y) = 1 + max x y
```

Defining the height on Inc

```haskell
instance HeightSt Inc where
    heightSt (Inc x) = 1 + x
```
Combining Automata

Making the height compositional

class HeightSt f where
    heightSt :: DUpState f q Int

instance (HeightSt f, HeightSt g) ⇒ HeightSt (f ⊕ g) where
    heightSt (Inl x) = heightSt x
    heightSt (Inr x) = heightSt x

Defining the height on Sig

instance HeightSt Sig where
    heightSt (Val _)    = 0
    heightSt (Plus x y) = 1 + max x y
Combining Automata

Making the height compositional

```haskell
class HeightSt f where
  heightSt :: DUppState f q Int

instance (HeightSt f, HeightSt g) ⇒ HeightSt (f ⊕ g) where
  heightSt (Inl x) = heightSt x
  heightSt (Inr x) = heightSt x
```

Defining the height on Sig

```haskell
instance HeightSt Sig where
  heightSt (Val _) = 0
  heightSt (Plus x y) = 1 + max x y
```

Defining the height on Inc

```haskell
instance HeightSt Inc where
  heightSt (Inc x) = 1 + x
```
Outline

1. Introduction

2. State Transition Functions
   - Composing State Spaces
   - Compositional Signatures

3. Tree Transducers
   - Bottom-Up Tree Transducers
   - Decomposing Tree Transducers

4. Conclusions
Bottom-Up Tree Transducers

From terms to contexts:

data Term \( f = \text{In}(f \text{Term}) \)

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

Representing transduction rules, \cite{Hasuo et al. 2007}:

\[
\text{type \ UpTrans} f q g = \forall a. f(q, a) \rightarrow (q, \text{Context} g a)
\]
Bottom-Up Tree Transducers

\[
\text{data} \quad \text{Term} f = \text{In} (f (\text{Term } f))
\]

\[
\text{data} \quad \text{Context } f a = \text{In} (f (\text{Context } f a)) \mid \text{Hole } a
\]

Representing transduction rules, [Hasuo et al. 2007]

\[
\text{UpTrans } f q g = \forall a. f (q, a) \rightarrow (q, \text{Context } g a)
\]
Bottom-Up Tree Transducers

From terms to contexts

Data: \( \text{Term} \ f \ = \ \text{In} (f (\text{Term} \ f)) \)

Data: \( \text{Context} \ f \ a \ = \ \text{In} (f (\text{Context} \ f \ a)) \mid \text{Hole} \ a \)
Bottom-Up Tree Transducers

From terms to contexts

\[ \text{data } \text{Term } f = \ln ( f ( \text{Term } f )) \]
\[ \text{data } \text{Context } f a = \ln ( f ( \text{Context } f a)) \mid \text{Hole } a \]
Bottom-Up Tree Transducers

From terms to contexts

**data** Term \( f \) = \text{In} (f (\text{Term} \ f ))

**data** Context \( f \ a \) = \text{In} (f (\text{Context} \ f \ a)) \mid \text{Hole} \ a

Representing transduction rules, [Hasuo et al. 2007]

**type** UpTrans \( f \ q \ g \) = \forall \ a. f (q, a) \rightarrow (q, \text{Context} \ g \ a)
Bottom-Up Tree Transducers

From terms to contexts

| data Term f = ln (f (Term f )) |
| data Context f a = ln (f (Context f a)) | Hole a |

Representing transduction rules, [Hasuo et al. 2007]

| type UpTrans f q g = ∀ a.f (q, a) → (q, Context g a) |
Tree Homomorphisms

type \textit{UpTrans} \; f \; q \; g = \forall \; a \cdot f \; (q, \; a) \rightarrow (q, \; \text{Context} \; g \; a)
Tree Homomorphisms

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

type $\text{Hom } f \rightarrow g = \forall a. f \ a \rightarrow \text{Context } g \ a$
Tree Homomorphisms

\[
\text{type } Hom \ f \ g = \forall a . \ f \ a \to Context \ g \ a
\]

Example (Desugaring)

\[
\text{class } DesugHom \ f \ g \text{ where }
\]
\[
\text{desugHom :: Hom } f \ g
\]
\[
\text{desugar :: (Functor } f , \text{ Functor } g , \text{ DesugHom } f \ g) \Rightarrow \text{ Term } f \to \text{ Term } g
\]
\[
\text{desugar = runHom desugHom}
\]
Tree Homomorphisms

type \( \text{Hom} \quad f \quad g = \forall a . f \ a \rightarrow \text{Context} \ g \ a \)

Example (Desugaring)

class \( \text{DesugHom} \ f \ g \) where

\[
\text{desugHom} :: \text{Hom} \ f \ g
\]

\[
\text{desugar} :: (\text{Functor} \ f , \text{Functor} \ g , \text{DesugHom} \ f \ g) \Rightarrow \text{Term} \ f \rightarrow \text{Term} \ g
\]

\[
\text{desugar} = \text{runHom} \ \text{desugHom}
\]

instance \( (\text{Sig} \leq g) \Rightarrow \text{DesugHom} \ \text{Inc} \ g \) where

\[
\text{desugHom} (\text{Inc} \ x) = \text{Hole} \ x \ \text{\textquoteleft plus\textquoteprime } \text{val} \ 1
\]

instance \( (\text{Functor} \ g , f \leq g) \Rightarrow \text{DesugHom} \ f \ g \) where

\[
\text{desugHom} = \text{simpCxt} \ . \ \text{inj}
\]
Tree Homomorphisms

type Hom f g = ∀ a . f a → Context g a

Example (Desugaring)

class DesugHom f g where
  desugHom :: Hom f g

desugar :: (Functor f, Functor g, DesugHom f g) ⇒ Term f → Term g
desugar = runHom desugHom

instance (Sig ⪯ g) ⇒ DesugHom Inc g where
  desugHom (Inc x) = Hole x' plus val 1

instance (Functor g, f ⪯ g) ⇒ DesugHom f g where
  desugHom = simpCxt . inj

simpCxt :: Functor g ⇒ g a → Context g a
simpCxt t = In (fmap Hole t)
Stateful Tree Homomorphisms

Decomposing tree transducers

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

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

type $\text{UpTrans} \ f \ q \ g = \forall \ a . \ f \ (q, a) \rightarrow (q, \text{Context} \ g \ a)$
Stateful Tree Homomorphisms

Decomposing tree transducers

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

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

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

Making homomorphisms dependent on a state

type $QHom \ f \ q \ g = \forall \ a . f a \rightarrow Context \ g \ a$
Stateful Tree Homomorphisms

### Decomposing tree transducers

<table>
<thead>
<tr>
<th>Type</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Hom f g</code></td>
<td><code>\forall a. f a \rightarrow \text{Context } g a</code></td>
</tr>
<tr>
<td><code>UpState f q</code></td>
<td><code>f q \rightarrow q</code></td>
</tr>
<tr>
<td><code>UpTrans f q g</code></td>
<td><code>\forall a. f (q, a) \rightarrow (q, \text{Context } g a)</code></td>
</tr>
</tbody>
</table>

### Making homomorphisms dependent on a state

<table>
<thead>
<tr>
<th>Type</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>QHom f q g</code></td>
<td><code>\forall a. f(q, a) \rightarrow \text{Context } g a</code></td>
</tr>
</tbody>
</table>
Stateful Tree Homomorphisms

Decomposing tree transducers

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

Making homomorphisms dependent on a state

type \( \text{QHom} \ f \ q \ g = \forall \ a . (a \to q) \to f \ a \to \text{Context} \ g \ a \)
Stateful Tree Homomorphisms

Decomposing tree transducers

\[ \text{type } \text{Hom } f \rightarrow g = \forall a . f \rightarrow a \rightarrow \text{Context } g \rightarrow a \]

\[ \text{type } \text{UpState } f \rightarrow q = f \rightarrow q \rightarrow q \]

\[ \text{type } \text{UpTrans } f \rightarrow q \rightarrow g = \forall a . f \rightarrow (q, a) \rightarrow (q, \text{Context } g \rightarrow a) \]

Making homomorphisms dependent on a state

\[ \text{type } \text{QHom } f \rightarrow q \rightarrow g = \forall a . (a \rightarrow q) \rightarrow f \rightarrow a \rightarrow \text{Context } g \rightarrow a \]

From stateful homomorphisms to tree transducers

\[ \text{upTrans :: } (\text{Functor } f, \text{Functor } g) \Rightarrow \]
\[ \text{UpState } f \rightarrow q \rightarrow \text{QHom } f \rightarrow q \rightarrow g \rightarrow \text{UpTrans } f \rightarrow q \rightarrow g \]

\[ \text{upTrans } st \rightarrow \text{hom } t = (q, c) \text{ where} \]
\[ q = st \ (\text{fmap } \text{fst } t) \]
\[ c = \text{fmap } \text{snd } (\text{hom } \text{fst } t) \]
An Example

Extending the signature with let bindings

type Name = String
data Let e = LetIn Name e e | Var Name
type LetSig = Let ⊕ Sig
An Example

Extending the signature with let bindings

```haskell
type Name = String
data Let e = LetIn Name e e | Var Name
type LetSig = Let ⊕ Sig


```
An Example

Extending the signature with let bindings

type Name = String

data Let e = LetIn Name e e | Var Name

type LetSig = Let ⊕ Sig

type Vars = Set Name

class FreeVarsSt f where
  freeVarsSt :: UpState f Vars

instance FreeVarsSt Sig where
  freeVarsSt (Plus x y) = x `union` y
  freeVarsSt (Val _) = empty

instance FreeVarsSt Let where
  freeVarsSt (Var v) = singleton v
  freeVarsSt (LetIn v e s) = if v `member` s then e `union` delete v s
                             else s
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 _ = simpCxt . inj
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 _ = simpCxt . inj

Combining state transition and homomorphism

remLet :: (Functor f, FreeVarsSt f, RemLetHom f Vars f)
  ⇒ Term f → (Vars, Term f)
remLet = runUpHom . freeVarsSt . remLetHom
An Example (Cont’d)

```haskell
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 _
```

Combining state transition and homomorphism

remLet :: (Functor f, FreeVarsSt f, RemLetHom f Vars f)
        ⇒ Term f → (Vars, Term f)
remLet = runUpHom · freeVarsSt · remLetHom

runUpHom :: UpState f q → QHom f q g
         → Term f → Term g
runUpHom st hom = runUpTrans (upTrans st hom)
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 _ = simpCxt . inj

Combining state transition and homomorphism

remLet :: (Functor f, FreeVarsSt f, RemLetHom f Vars f)
  ⇒ Term f → (Vars, Term f)
remLet = runUpHom freeVarsSt remLetHom

remLet :: Term LetSig → Term LetSig
remLet :: Term (Inc ⊕ LetSig) → Term (Inc ⊕ LetSig)
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
<table>
<thead>
<tr>
<th>What have we seen?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bottom-up tree acceptors (a.k.a. folds)</td>
</tr>
<tr>
<td>Bottom-up tree transducers</td>
</tr>
<tr>
<td>“dependent” versions thereof</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Other Tree recursion schemes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Top-down tree acceptors</td>
</tr>
<tr>
<td>Top-down tree transducers</td>
</tr>
<tr>
<td>“dependent” versions thereof</td>
</tr>
<tr>
<td>automata with bidirectional state propagation</td>
</tr>
<tr>
<td>(restricted versions of macro tree transducers)</td>
</tr>
</tbody>
</table>
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