Attribute Grammars in Haskell with UUAG

Andres Löh
joint work with S. Doaitse Swierstra and Arthur Baars

andres@cs.uu.nl

10 February 2005
A simplified view on compilers

- Input is transformed into output.
- Input and output language have little structure.
- During the process structure such as an Abstract Syntax Tree (AST) is created.

![Diagram of AST structure]

Universiteit Utrecht
Abstract syntax and grammars

- The structure in an AST is best described by a (context-free) grammar.
- A concrete value (program) is a word of the language defined by that grammar.

\[
\text{Expr} \rightarrow \text{Var} \quad \text{-- variable} \\
\quad | \quad \text{Expr} \ \text{Expr} \quad \text{-- application} \\
\quad | \quad \text{Var} \ \text{Expr} \quad \text{-- lambda abstraction}
\]

- The rules in a grammar are called \textbf{productions}. The right hand side of a rule is \textbf{derivable} from the left hand side.
- The symbols on the left hand side are called \textbf{nonterminals}.
- A word is in the language defined by the grammar if it is derivable from the \textbf{root nonterminal} in a finite number of steps.
Example grammar

In the following, we will use the following example grammar for a very simple language:

Root → Expr

Expr → Var -- variable
    | Expr Expr -- application
    | Var Expr -- \lambda
    | Decls Expr -- let

Decls → Decl Decls
    | \epsilon

Decl → Var Expr

Var → String -- name
Haskell: Algebraic datatypes

- In Haskell, you can define your own datatypes.
- Choice is encoded using multiple constructors.
- Constructors may contain fields.
- Types can be parametrized.
- Types can be recursive.

```haskell
data Bit = Zero | One
data Complex = Complex Real Real
data Maybe a = Just a | Nothing
data List a = Nil | Cons a (List a)
```
In Haskell, you can define your own datatypes.

Choice is encoded using multiple constructors.

Constructors may contain fields.

Types can be parametrized.

Types can be recursive.

data Bit = Zero | One
data Complex = Complex Real Real
data Maybe a = Just a | Nothing
data List a = Nil | Cons a (List a)
Haskell: Algebraic datatypes

- In Haskell, you can define your own datatypes.
- Choice is encoded using multiple constructors.
- Constructors may contain fields.
- Types can be parametrized.
- Types can be recursive.

```haskell
data Bit = Zero | One
data Complex = Complex Real Real
data Maybe a = Just a | Nothing
data List a = Nil | Cons a (List a)
```
In Haskell, you can define your own datatypes. Choice is encoded using multiple constructors. Constructors may contain fields. Types can be parametrized. Types can be recursive.

```haskell
data Bit = Zero | One
data Complex = Complex Real Real
data Maybe a = Just a | Nothing
data List a = Nil | Cons a (List a)
```
In Haskell, you can define your own datatypes.
Choice is encoded using multiple constructors.
Constructors may contain fields.
Types can be parametrized.
Types can be recursive.

```haskell
data Bit       = Zero | One
data Complex   = Complex Real Real
data Maybe a   = Just a | Nothing
data List a    = Nil | Cons a (List a)
```
There is a builtin list type with special syntax.

```haskell
data [a] = [] | a : [a]
[1, 2, 3, 4, 5] == (1 : (2 : (3 : (4 : (5 : []))))))
```
Grammars correspond to datatypes

- Given this power, each nonterminal can be seen as a data type.
- Productions correspond to definitions of constructors.
- For each constructor, we need a name.
- Type abstraction is not needed, but recursion is.
The example grammar translated

<table>
<thead>
<tr>
<th>Root</th>
<th>→</th>
<th>Expr</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expr</td>
<td>→</td>
<td>Var</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Expr Expr</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Var Expr</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Decls Expr</td>
</tr>
<tr>
<td>Decls</td>
<td>→</td>
<td>Decl Decls</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ε</td>
</tr>
<tr>
<td>Decl</td>
<td>→</td>
<td>Var Expr</td>
</tr>
<tr>
<td>Var</td>
<td>→</td>
<td>String</td>
</tr>
</tbody>
</table>

**data**

Root = Root Expr

Expr = Var Var

|       |    | App Expr Expr |
|       |    | Lam Var Expr |
|       |    | Let Decls Expr |

Decls = Cons Decls Decls

|       |    | Nil {- ε -} |

Decl = Decl Var Expr

Var = Ident String
The example grammar translated

\begin{align*}
\text{Root} & \rightarrow \text{Expr} \\
\text{Expr} & \rightarrow \text{Var} \\
& \quad \mid \text{Expr} \text{Expr} \\
& \quad \mid \text{Var} \text{Expr} \\
& \quad \mid \text{Decls} \text{Expr} \\
\text{Decls} & \rightarrow \text{Decl} \text{Decls} \\
& \quad \mid \epsilon \\
\text{Decl} & \rightarrow \text{Var} \text{Expr} \\
\text{Var} & \rightarrow \text{String} \\
\end{align*}

\begin{align*}
\text{DATA} \text{ Root} & \mid \text{Root} \ \text{Expr} \\
\text{DATA} \text{ Expr} & \mid \text{Var} \ \text{Var} \\
& \quad \mid \text{App} \ \text{fun} : \text{Expr} \ \text{arg} : \text{Expr} \\
& \quad \mid \text{Lam} \ \text{Var} \ \text{Expr} \\
& \quad \mid \text{Let} \ \text{Decls} \ \text{Expr} \\
\text{DATA} \text{ Decls} & \mid \text{Cons} \ \text{hd} : \text{Decls} \ \text{tl} : \text{Decls} \\
& \quad \mid \text{Nil} \ \{
\epsilon 
\}
\\
\text{DATA} \text{ Decl} & \mid \text{Decl} \ \text{Var} \ \text{Expr} \\
\text{DATA} \text{ Var} & \mid \text{Ident} \ \text{name} : \text{String}
\end{align*}
The example grammar translated

Root → Expr
Expr → Var
  | Expr Expr
  | Var Expr
  | Decl Expr
Decl → Decl Decl
| ε
Decl → Var Expr
Var → String

DATA Root | Root Expr
DATA Expr | Var Var
| App fun : Expr arg : Expr
| Lam Var Expr
| Let Decl Expr

TYPE Decl = [Decl]

DATA Decl | Decl Var Expr
DATA Var | Ident name : String
### UUAG datatypes

- Datatypes in UUAG are much like in Haskell.
- Constructors of different datatypes may have the same name.
- Some minor syntactical differences.
- Each field has a name. The type name is the default.

```plaintext
DATA Expr | Var  Var
           | App fun : Expr arg : Expr
           | Lam Var Expr
           | Let  Decls Expr

is an abbreviation of

DATA Expr | Var  var : Var
           | App fun : Expr arg : Expr
           | Lam var : Var expr : Expr
           | Let  decls : Decls expr : Expr
```
UUAG datatypes

- Datatypes in UUAG are much like in Haskell.
- Constructors of different datatypes may have the same name.
- Some minor syntactical differences.
- Each field has a name. The type name is the default.

```
DATA Expr | Var  Var
          | App fun : Expr arg : Expr
          | Lam  Var Expr
          | Let  Decls Expr
```

is an abbreviation of

```
DATA Expr | Var  var : Var
          | App fun : Expr arg : Expr
          | Lam  var : Var expr : Expr
          | Let  decls : Decls expr : Expr
```
An example value

Root (Let (Cons (Decl (Ident "k") (Var (Ident "const"))))
       (Cons (Decl (Ident "i") (Lam (Ident "x")
                        (Var (Ident "x")))))
       Nil))
       (App (Var (Ident "k")) (Var (Ident "i"))))

Haskell-like syntax:

```
let k = const
    i = λx → x
in  k i
```
AST

Root (Root)
  / |
Let (Expr)
  / |
  Cons (Decls)
    / | |
    Decl (Decl)
    / | |
    Ident (Var) Var (Expr)
    / | |
    Ident (Var) Ident (Var)Lam (Expr)
    / | |
    Ident (Var) Var (Expr)
    / |
    Ident (Var)

Suppose we have the following AST:

```
Root
  Let
    Cons
      Decl
        Ident
          Var
            App
              Var
                Decl
                  Ident
                    Var
                      Ident
                        Var
                          Nil
                            Decl
                                Decl
                                    Decl
                                        Decl
                                            Decl
                                                Decl
                                                    Decl
                                                        Decl
                                                            Decl
                                                                Decl
                                                                    Decl
                                                                        Decl
                                                                        Decl
                                                                Var
                                                                    Var
                                                Ident
                                        Ident
                                Ident
                        Ident
                  Ident
            Ident
    Var
```

Universiteit Utrecht
Computation follows structure

▶ Many computations can be expressed in a common way.
▶ Information is passed upwards.
▶ Constructors are replaced by operations.
▶ In the leaves, results are created.
▶ In the nodes, results are combined.
Synthesised attributes

- In UUAG (and in attribute grammars), computations are modelled by attributes.
- Each of the examples defines an attribute.
- Attributes that are computed bottom-up are called synthesised attributes.
Synthesised attribute computation in UUAG

**ATTR** Root Expr Decls Decl Var

```
[ | | allvars : { [String] } ]
```

**SEM** Root

```
| Root  lhs.allvars = @expr.allvars
```

**SEM** Expr

```
| Var   lhs.allvars = @var.allvars
| App   lhs.allvars = @fun.allvars \cup @arg.allvars
| Lam   lhs.allvars = @var.allvars \cup @expr.allvars
| Let    lhs.allvars = @decls.allvars \cup @expr.allvars
```

**SEM** Decls

```
| Cons  lhs.allvars = @hd.allvars \cup @tail.allvars
| Nil    lhs.allvars = []
```

**SEM** Decl

```
| Decl  lhs.allvars = @var.allvars \cup @expr.allvars
```

**SEM** Var

```
| Ident lhs.allvars = [@name]
```
Synthesised attribute computation in UUAG

**ATR** Root Expr Decls Decl Var

\[\text{allvars} : \{\text{String}\} \}\]

**SEM** Root

\[\text{Root } \text{lhs.allvars} = @\text{expr.allvars}\]

**SEM** Expr

| Var  \(\text{lhs.allvars} = @\text{var.allvars}\) |
| App  \(\text{lhs.allvars} = @\text{fun.allvars} \cup @\text{arg.allvars}\) |
| Lam  \(\text{lhs.allvars} = @\text{var.allvars} \cup @\text{expr.allvars}\) |
| Let  \(\text{lhs.allvars} = @\text{decls.allvars} \cup @\text{expr.allvars}\) |

**SEM** Decls

| Cons \(\text{lhs.allvars} = @\text{hd.allvars} \cup @\text{tail.allvars}\) |
| Nil  \(\text{lhs.allvars} = []\) |

**SEM** Decl

| Decl \(\text{lhs.allvars} = @\text{var.allvars} \cup @\text{expr.allvars}\) |

**SEM** Var

| Ident \(\text{lhs.allvars} = [@\text{name}]\) |
Synthesised attribute computation in UUAG

**ATTR** Root Expr Decls Decl Var
  [ | | allvars : { [String] }] |

**SEM** Expr
  |
  | App  lhs.allvars = @fun.allvars ∪ @arg.allvars
  | Lam   lhs.allvars = @var.allvars ∪ @expr.allvars
  | Let   lhs.allvars = @decls.allvars ∪ @expr.allvars

**SEM** Decls
  |
  | Cons  lhs.allvars = @hd.allvars ∪ @tail.allvars
  | Nil    lhs.allvars = []

**SEM** Decl
  |
  | Decl  lhs.allvars = @var.allvars ∪ @expr.allvars

**SEM** Var
  |
  | Ident lhs.allvars = [@name]
Synthesised attribute computation in UUAG

ATTR Root Expr Decls Decl Var
\[
\begin{bmatrix}
\mid | | \ allvars : \{ [\text{String}] \} \ USE \{ \cup \} \{ [] \} [ ]
\end{bmatrix}
\]

SEM Var
\[
\begin{align*}
| \text{Ident lhs.allvars} &= [\@\text{name}] \\
\end{align*}
\]
Synthesised attribute computation in UUAG

\textbf{ATTR} \ Root \ Expr \ Decls \ Decl \ Var
\begin{align*}
[ | | \textit{allvars} : \{ \texttt{[String]} \} & \textbf{USE} \{ \cup \} \{ \texttt{[]} \} ]
\end{align*}

\textbf{SEM Var}
\begin{align*}
| \textit{Ident} \ lhs.\textit{allvars} = \texttt{[@name]} 
\end{align*}
Synthesised attribute computation in UUAG

\[ \text{ATTR} \ast \]
\[ [ | | \text{allvars} : \{ [\text{String}] \} \text{ USE} \{ \cup \} \{ [ ] \} ] \]

\[ \text{SEM Var} \]
\[ | \text{Ident } \text{lhs} \text{.allvars} = \{ @\text{name} \} \]
Abbreviations

- UUAG allows the programmer to omit straight-forward propagation.
- For synthesised attributes, a synthesised attribute is by default propagated from the leftmost child that provides an attribute of the same name.
- If instead the results should be combined in a uniform way, a **USE** construct can be employed. This takes a constant which becomes the default for a leaf, and a binary operator which becomes the default combination operator.
Abbreviations

- UUAG allows the programmer to omit straight-forward propagation.
- For synthesised attributes, a synthesised attribute is by default propagated from the leftmost child that provides an attribute of the same name.
- If instead the results should be combined in a uniform way, a USE construct can be employed. This takes a constant which becomes the default for a leaf, and a binary operator which becomes the default combination operator.
Abbreviations

- UUAG allows the programmer to omit straight-forward propagation.
- For synthesised attributes, a synthesised attribute is by default propagated from the leftmost child that provides an attribute of the same name.
- If instead the results should be combined in a uniform way, a **USE** construct can be employed. This takes a constant which becomes the default for a leaf, and a binary operator which becomes the default combination operator.
Sets of nonterminals

\textbf{SET} All = Root Expr Decls Decl Var

\texttt{*}

--- implicitly defined All, contains all \textbf{DATA} types in scope

\textbf{SET} D = Decls Decl

All \smallsetminus D

--- set difference

Root \rightarrow Var

--- all nonterminals on paths from Root to Var, excluding Root

\> Such sets can be used as arguments to \texttt{ATTR} and \texttt{SEM}.  

Universiteit Utrecht
Combining computations

- Attributes can (mutually) depend on each other.

\[
\text{ATTR } * \\
\quad [ \mid \mid \text{freevars} : \{ \text{String} \} \text{ USE } \{ \cup \} \{ [] \} ]
\]

\[
\text{ATTR } D \\
\quad [ \mid \mid \text{defvars} : \{ \text{String} \} \text{ USE } \{ \pm \} \{ [] \} ]
\]

\[
\text{SEM } \text{Var} \\
\quad | \text{Ident } \text{lhs.freevars} = [@\text{name}]
\]

\[
\text{SEM } \text{Expr} \\
\quad | \text{Lam } \text{lhs.freevars} = \text{@expr.freevars} - \text{@var.freevars} \\
\quad | \text{Let } \text{lhs.freevars} = (\text{@expr.freevars} \cup \text{@decls.freevars}) \\
\quad \quad \quad \quad - \text{@decls.defvars}
\]

\[
\text{SEM } \text{Decl} \\
\quad | \text{Decl } \text{lhs.freevars} = \text{@expr.freevars} \quad \text{-- overriding USE } \\
\quad \text{lhs.defvars} = \text{@var.freevars}
\]
Distributing information

- Sometimes synthesised attributes depend on outside information.
- Examples: Options, parameters, environments, results of other computations.
- In these cases it is not sufficient to pass information bottom-up. We need top-down attributes, too!
- Such attributes are called **inherited attributes**.
A substitution environment

**ATTR** Root (Root → Expr)

\[ \text{substenv} : \{ \text{FiniteMap Var Expr} \} \mid \mid \]

**SEM** Root

\| Root expr.substenv = @lhs.substenv

**SEM** Expr

\| App fun.substenv = @lhs.substenv  
  app.substenv = @lhs.substenv

\| Lam expr.substenv = delListFromFM @lhs.substenv @var.freevars

\| Let loc.substenv = delListFromFM @lhs.substenv @decls.defvars
  decls.substenv = @loc.substenv
  expr.substenv = @loc.substenv

**SEM** Decls

\| Cons hd.substenv = @lhs.substenv
  tl.substenv = @lhs.substenv

**SEM** Decl

\| Decl expr.substenv = @lhs.substenv
A substitution environment

\textbf{ATTR} Root (Root → Expr)

\[ \text{substenv} : \{ \text{FiniteMap Var Expr} \} \]

\textbf{SEM} Root

| Root expr.substenv = @lhs.substenv |

\textbf{SEM} Expr

| App fun.substenv = @lhs.substenv |
| app.substenv = @lhs.substenv |
| Lam expr.substenv = delListFromFM @lhs.substenv @var.freevars |
| Let loc.substenv = delListFromFM @lhs.substenv @decls.defvars |
| decls.substenv = @loc.substenv |
| expr.substenv = @loc.substenv |

\textbf{SEM} Decls

| Cons hd.substenv = @lhs.substenv |
| tl.substenv = @lhs.substenv |

\textbf{SEM} Decl

| Decl expr.substenv = @lhs.substenv |
A substitution environment

ATTR Root (Root $\rightarrow$ Expr)
  $[\text{substenv} : \{\text{FiniteMap Var Expr}\}]$  ||  |

SEM Expr

  $| \text{Lam } \text{expr.substenv} = \text{delListFromFM @lhs.substenv @var.freevars}$
  $| \text{Let } \text{loc.substenv} = \text{delListFromFM @lhs.substenv @decls.defvars}$
Copy rules

- For inherited attributes, it is again possible to omit uninteresting cases.
- One can define local variables. Local variables are propagated in all directions with priority (i.e., they are propagated upwards if they have the name of a synthesised attribute, and downwards if they have the name of an inherited attribute).
- If no local variable is available, a required inherited attribute is propagated from the left hand side.
Performing a substitution

Of course, inherited attributes and synthesised attributes can interact.

\[
\text{ATTR} \ast - \text{Root} \\
\text{ATTR} \text{Root} [ | | \text{substituted : SELF}] \\
\text{ATTR} \text{Expr} [ | | \text{substituted : Expr}] \\
\text{ATTR} \text{Expr} | \text{Var lhs.substituted} = \text{case lookupFM @lhs.substenv} @var.substituted \text{ of} \\
\text{Just expr} \rightarrow \text{expr} \\
\text{Nothing} \rightarrow \text{Var @var.substituted}
\]
Generating a modified tree

- The **SELF** construct is another powerful built-in mechanism to support generating a modification of the original tree.
- A **SELF** attribute comes with default rules that reconstruct the original tree.
In functional languages functions are first-class values. In short: you can treat a function like any other value.

Functions can be results of functions.

\[
\begin{align*}
(+) &:: \text{Int} \to (\text{Int} \to \text{Int}) \\
(+) \ 2 &:: \text{Int} \to \text{Int} \\
(+) \ 2 \ 3 &:: \text{Int}
\end{align*}
\]

Functions can be arguments of functions.

\[
\begin{align*}
twice &:: (a \to a) \to (a \to a) \\
twice \ f \ x &= f \ (f \ x) \\
twice \ ((+\ 17)) \ 8 &= 42 \\
map &:: (a \to b) \to ([a] \to [b]) \\
map \ f \ [] &= [] \\
map \ f \ (x:xs) &= f \ x : map \ f \ xs
\end{align*}
\]
A **catamorphism** is a function that computes a result out of a value of a data type by
- replacing the constructors with operations
- replacing recursive occurrences by recursive calls to the catamorphism

Since Haskell provides algebraic data types, catamorphisms can be written easily in Haskell.

Synthesized attributes can be translated into “catamorphic form” in a straight-forward way.
Example translation

\[
\begin{align*}
\text{allvars\_Root} & : \text{Root} \rightarrow \mathbb{[}\text{String}] \\
\text{allvars\_Root}\ (\text{Root}\ \text{expr}) & = \text{allvars\_Expr}\ \text{expr} \\
\text{allvars\_Expr} & : \text{Expr} \rightarrow \mathbb{[}\text{String}] \\
\text{allvars\_Expr}\ (\text{Var}\ \text{var}) & = \text{allvars\_Var}\ \text{var} \\
\text{allvars\_Expr}\ (\text{App}\ \text{fun}\ \text{arg}) & = \text{let}\ \text{fun\_allvars} = \text{allvars\_Expr}\ \text{fun} \\
& \quad \text{arg\_allvars} = \text{allvars\_Expr}\ \text{arg} \\
& \quad \text{in}\ \text{fun\_allvars} \cup \text{arg\_allvars} \\
\ldots \\
\text{allvars\_Var} & : \text{Var} \rightarrow \mathbb{[}\text{String}] \\
\text{allvars\_Var}\ (\text{Ident}\ \text{name}) & = \mathbb{[}\text{name}] \\
\end{align*}
\]
Catamorphisms can be combined

- Several attributes: Several catamorphisms?
- Better: Write one catamorphism computing a tuple!
- Only one traversal of the tree, attributes can depend on each other.
Translating “free variables”

**SEM Expr**

| Let \( \text{lhs.freevars} = (@\text{expr.feevars} \cup @\text{decls.freevars}) \)
| \( - @\text{decls.defvars} \)

**SEM Decl**

| Decl \( \text{lhs.freevars} = @\text{expr.feevars} \) -- overriding USE
| \( \text{lhs.defvars} = @\text{var.feevars} \)

\[
\begin{align*}
\text{sem.Expr} & \quad :: \quad \text{Expr} \to [\text{String}] \\
\text{sem.Expr} (\text{Let decls expr}) &= \\
\text{let} \quad (\text{decls.defvars, decls.feevars}) &= \text{sem.Decls decls} \\
\quad \text{expr.feevars} &= \text{sem.Expr expr} \\
\text{in} \quad (\text{expr.feevars} \cup \text{decls.feevars}) \\
&- (\text{decls.feevars}) \\
\text{sem.Decl} & \quad :: \quad \text{Decl} \to ([\text{String}], [\text{String}]) \\
\text{sem.Decl} (\text{Decl var expr}) &= \\
\text{let} \quad \text{var.feevars} &= \text{sem.Var var} \\
\quad \text{expr.feevars} &= \text{sem.Expr expr} \\
\text{in} \quad (\text{var.feevars, expr.feevars})
\end{align*}
\]
Catamorphisms can compute functions

- Inherited attributes can be realised by computing functional values.
- In fact, a group of inherited and synthesised attributes is isomorphic to one synthesised attribute with a functional value.
- The final catamorphism for a type Type has type

\[
\text{sem\_Type} :: \text{Type} \rightarrow \text{Sem\_Type}
\]

where \text{Sem\_Type} is a type synonym for a functional type, mapping all inherited attributes to the synthesised attributes for Type:

\[
\text{type \ Sem\_Type} = \text{Inh}_1 \rightarrow \text{Inh}_2 \rightarrow \cdots \rightarrow \text{Inh}_m \\
\rightarrow (\text{Syn}_1, \text{Syn}_2, \ldots, \text{Syn}_n)
\]
Translating “substitution”

\[
\text{SEM } \text{Expr} \ [substenv : \{ \text{FiniteMap Var Expr} \} \\
\qquad | \qquad \text{substituted} : \text{SELF} \\
\qquad \text{freevars : [String]}] \\
\qquad | \qquad \text{Lam expr.substenv} = \text{delListFromFM } @\text{lhs.substenv } @\text{var.freevars} \\
\qquad | \qquad \text{Var } \text{lhs.substituted} = \text{case lookupFM }\ldots
\]

\text{type} \ \text{Sem.Expr} = \text{FiniteMap Var Expr } \rightarrow [\text{String}], \text{Expr} \\
\text{sem.Expr} :: \text{Expr } \rightarrow \text{Sem.Expr} \\
\text{sem.Expr} (\text{Lam var expr}) \text{lhs.substenv} = \\
\quad \text{let} (\text{var.freevars, var.substituted}) = \text{sem.Var var lhs.substenv} \\
\quad (\text{expr.freevars, expr.substituted}) = \text{sem.Var var (delListFromFM lhs.substenv var.freevars)} \\
\quad \text{in} \ \text{Lam var.substituted expr.substituted } \{ - \text{SELF default} - \} \\
\text{sem.Expr} (\text{Var var}) \text{lhs.substenv} = \\
\quad \text{let} (\text{var.freevars, var.substituted}) = \text{sem.Var var lhs.substenv} \\
\quad \text{in} \ \text{case lookupFM }\ldots
Implementation of UUAG

- Translates UUAG source files into a Haskell module.
- Normal Haskell code can occur in UUAG source files as well as in other modules.
- UUAG data types are translated into Haskell data types.
- Attribute definitions are translated into one catamorphism per data type, computing a function that maps the inherited to the synthesised attributes of the data type.
- The catamorphism generated for the root symbol is the entry point to the computation.
- UUAG copies the right-hand sides of rules almost literally and without interpretation.
- All Haskell constructs are available, system is lightweight.
- No type check on UUAG level; the generation process must be understood by the programmer.
Implementation of UUAG

- Translates UUAG source files into a Haskell module.
- Normal Haskell code can occur in UUAG source files as well as in other modules.
- UUAG data types are translated into Haskell data types.
- Attribute definitions are translated into one catamorphism per data type, computing a function that maps the inherited to the synthesised attributes of the data type.
- The catamorphism generated for the root symbol is the entry point to the computation.
- UUAG copies the right-hand sides of rules almost literally and without interpretation.
- All Haskell constructs are available, system is lightweight.
- No type check on UUAG level; the generation process must be understood by the programmer.
Implementation of UUAG

- Translates UUAG source files into a Haskell module.
- Normal Haskell code can occur in UUAG source files as well as in other modules.
- UUAG data types are translated into Haskell data types.
  - Attribute definitions are translated into one catamorphism per data type, computing a function that maps the inherited to the synthesised attributes of the data type.
  - The catamorphism generated for the root symbol is the entry point to the computation.
- UUAG copies the right-hand sides of rules almost literally and without interpretation.
- All Haskell constructs are available, system is lightweight.
- No type check on UUAG level; the generation process must be understood by the programmer.
Implementation of UUAG

- Translates UUAG source files into a Haskell module.
- Normal Haskell code can occur in UUAG source files as well as in other modules.
- UUAG data types are translated into Haskell data types.
- Attribute definitions are translated into one catamorphism per data type, computing a function that maps the inherited to the synthesised attributes of the data type.
- The catamorphism generated for the root symbol is the entry point to the computation.
- UUAG copies the right-hand sides of rules almost literally and without interpretation.
- All Haskell constructs are available, system is lightweight.
- No type check on UUAG level; the generation process must be understood by the programmer.
Implementation of UUAG

- Translates UUAG source files into a Haskell module.
- Normal Haskell code can occur in UUAG source files as well as in other modules.
- UUAG data types are translated into Haskell data types.
- Attribute definitions are translated into one catamorphism per data type, computing a function that maps the inherited to the synthesised attributes of the data type.
- The catamorphism generated for the root symbol is the entry point to the computation.
- UUAG copies the right-hand sides of rules almost literally and without interpretation.
- All Haskell constructs are available, system is lightweight.
- No type check on UUAG level; the generation process must be understood by the programmer.
Implementation of UUAG

- Translates UUAG source files into a Haskell module.
- Normal Haskell code can occur in UUAG source files as well as in other modules.
- UUAG data types are translated into Haskell data types.
- Attribute definitions are translated into one catamorphism per data type, computing a function that maps the inherited to the synthesised attributes of the data type.
- The catamorphism generated for the root symbol is the entry point to the computation.
- UUAG copies the right-hand sides of rules almost literally and without interpretation.

  - all Haskell constructs are available, system is lightweight
  - no type check on UUAG level; the generation process must be understood by the programmer
Implementation of UUAG

- Translates UUAG source files into a Haskell module.
- Normal Haskell code can occur in UUAG source files as well as in other modules.
- UUAG data types are translated into Haskell data types.
- Attribute definitions are translated into one catamorphism per data type, computing a function that maps the inherited to the synthesised attributes of the data type.
- The catamorphism generated for the root symbol is the entry point to the computation.
- UUAG copies the right-hand sides of rules almost literally and without interpretation.
- all Haskell constructs are available, system is lightweight
- no type check on UUAG level; the generation process must be understood by the programmer
Implementation of UUAG

- Translates UUAG source files into a Haskell module.
- Normal Haskell code can occur in UUAG source files as well as in other modules.
- UUAG data types are translated into Haskell data types.
- Attribute definitions are translated into one catamorphism per data type, computing a function that maps the inherited to the synthesised attributes of the data type.
- The catamorphism generated for the root symbol is the entry point to the computation.
- UUAG copies the right-hand sides of rules almost literally and without interpretation.
- all Haskell constructs are available, system is lightweight
- no type check on UUAG level; the generation process must be understood by the programmer
Haskell: lazy evaluation

- Function applications are reduced in “applicative order”: First the function, then (and **only if needed**) the arguments.
- Lazy boolean “or” function: \( \text{True} \lor \text{error} "\text{unreachable}" \)
- Lazy evaluation allows dealing with infinite data structures, as long as only a finite part is used in the end.

\[
\text{primes :: [Int]}
\]
\[
\text{primes = sieve [2 ..]}
\]
\[
\text{sieve :: [Int] \to [Int]}
\]
\[
\text{sieve (x : xs) = x : sieve [y | y \leftarrow xs, y \text{‘mod’ } x \neq 0]}
\]
\[
\text{take 100 primes}
\]

- As a consequence, the UUAG does not need to specify the order in which attributes are evaluated.
Haskell: lazy evaluation

- Function applications are reduced in “applicative order”: First the function, then (and **only if needed**) the arguments.
- Lazy boolean “or” function: $\text{True} \lor \text{error} \ "unreachable"$
- Lazy evaluation allows dealing with infinite data structures, as long as only a finite part is used in the end.

```haskell
primes :: [Int]
primes = sieve [2..]
sieve :: [Int] → [Int]
sieve (x : xs) = x : sieve [y | y ← xs, y `mod` x /= 0]
take 100 primes
```

- As a consequence, the UUAG does not need to specify the order in which attributes are evaluated.
Chained attributes

- Often, attributes should be both inherited and synthesised at the same time, traversing the whole tree, representing a current state.
- Such attributes are called **chained attributes**.
- They are nothing special, but there is syntactic sugar for them:

  ```
  ATTR * — Root [ | unique : Int | ]
  ```

  is short for

  ```
  ATTR * — Root [unique : Int | unique : Int]
  ```

- The default copy rules perform a depth-first top-down traversal from left to right.
Keeping an environment of type assumptions

ATTR * ─ Root [  | env : FiniteMap Var Type
                unique : Int
                | self : SELF ]

SEM Root
  └ Root expr.env = fmToList ["const", parseType "a -> b -> a"
        expr.unique = 0

SEM Expr
  └ Lam expr.unique = @lhs.unique + 1
      expr.env = addToFM @lhs.env
               (@var.self, tyVar @lhs.unique)
Depth-first traversal

**DATA** Root | Root Tree
**DATA** Tree | Leaf label : Int
| Node left : Leaf right : Leaf
**ATTR** Tree [ | counter : Int | dft : SELF]
**SEM** Root
| Root tree.counter = 0
**SEM** Tree
| Leaf lhs.counter = @lhs.counter + 1
  lhs.dft = Leaf @lhs.counter
Full copy rule

- For every node, the inputs are the inherited attributes of the left hand side, and the synthesized attributes of the children. Similarly, the outputs are the synthesized attributes of the left hand side, and the inherited attributes of the children.

- We define a partial order between attributes of the same name: left hand side attributes are smallest, then the children from left to right.

- When we must compute a synthesized **USE** or **SELF** attribute, we combine the results of the children or reconstruct the tree, respectively.

- Whenever we need an output, we first take it from a local attribute of the same name.

- If there’s no local attribute, we look for the largest smaller input attribute of the same name.
The copy rules we have used before are special instances of this general rule.

For chained attributes, the rule specifies exactly the depth-first traversal.
Breadth-first traversal

▪ A breadth-first traversal is not immediately covered by the copy rules.
▪ Nevertheless, it can be realised with only slightly more work (but making essential use of lazy evaluation!).
▪ Combinations of BF and DF traversal are often useful to implement scope of entities.
▪ Basic Idea: Provide a list with initial counter values for each level, return a list with final counter values for each level.
## Implementing BFT

<table>
<thead>
<tr>
<th>DATA Root</th>
<th>Root Tree</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATA Tree</td>
<td>Leaf label : Int</td>
</tr>
<tr>
<td></td>
<td>Node left : Leaf right : Leaf</td>
</tr>
<tr>
<td>ATTR Tree</td>
<td>levels : [Int]</td>
</tr>
</tbody>
</table>

**SEM Root**

| Root tree.levels = 0 : @tree.levels |

**SEM Tree**

| Node left.levels = tail @lhs.levels |
| lhs.levels = head @lhs.levels : tail (@right. · .levels) |
| Leaf loc.label = head @lhs.levels |
| lhs.levels = (@loc.label + 1) : tail @lhs.levels |
| lhs.bft = Leaf @loc.label |

- Note that this AG is circular.
Extending AGs

- As we have already seen, AGs can naturally be extended with new attributes. We simply add a new attribute definition and new semantic rules.

- We can, however, also extend the grammar, adding new datatypes or new constructors to datatypes(!). The AG system allows to group the rules in any way the programmer likes.

```
DATA Expr
  | Int  Int
  | Pair Expr Expr
```
As we have already seen, AGs can naturally be extended with new attributes. We simply add a new attribute definition and new semantic rules.

We can, however, also extend the grammar, adding new datatypes or new constructors to datatypes(!). The AG system allows to group the rules in any way the programmer likes.

```
DATA Expr
    | Int  Int
    | Pair Expr Expr
```
Conclusions

- Programming with UUAG is easy and fun.
- Application areas are compilers in the widest meaning of the word.
- Used in Utrecht to implement GH, Helium, Morrow, and EHC, all of which are of reasonable size.
- Available and stable.