Attribute Grammars in Haskell with UUAG

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

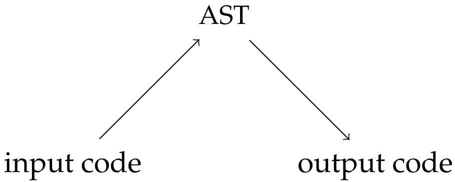
andres@cs.uu.nl

10 February 2005

Universiteit Utrecht

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.



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.

```
Expr → Var          -- variable
     | Expr Expr    -- application
     | Var Expr     -- lambda abstraction
```

- The rules in a grammar are called **productions**. The right hand side of a rule is **derivable** from the left hand side.
- The symbols on the left hand side are called **nonterminals**.
- A word is in the language defined by the grammar if it is derivable from the **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     -- λ
      | Decls Expr   -- let
Decls → Decl Decls
      | ε
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**.

```
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**.

```
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**.

```
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**.

```
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)
```

Haskell: Algebraic datatypes (contd.)

- There is a builtin list type with special syntax.

```
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

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

```
data  Root  = Root  Expr
data  Expr  = Var   Var
            | App   Expr Expr
            | Lam   Var Expr
            | Let   Decls Expr
data  Decls = Cons  Decls Decls
            | Nil     {- ε -}
data  Decl  = Decl  Var Expr
data  Var   = Ident String
```

The example grammar translated

Root	$\longrightarrow$	Expr
Expr	$\longrightarrow$	Var
	$\mid$	Expr Expr
	$\mid$	Var Expr
	$\mid$	Decls Expr
Decls	$\longrightarrow$	Decl Decls
	$\mid$	ε
Decl	$\longrightarrow$	Var Expr
Var	$\longrightarrow$	String

DATA Root	$\mid$	*Root*	Expr
DATA Expr	$\mid$	*Var*	Var
	$\mid$	*App*	*fun* : Expr *arg* : Expr
	$\mid$	*Lam*	Var Expr
	$\mid$	*Let*	Decls Expr
DATA Decls	$\mid$	*Cons*	*hd* : Decls *tl* : Decls
	$\mid$	*Nil*	{- ε -}
DATA Decl	$\mid$	*Decl*	Var Expr
DATA Var	$\mid$	*Ident*	*name* : String

The example grammar translated

$$\begin{array}{lcl}
\mathsf{Root} & \longrightarrow & \mathsf{Expr} \\
\mathsf{Expr} & \longrightarrow & \mathsf{Var} \\
 & \mid & \mathsf{Expr}\ \mathsf{Expr} \\
 & \mid & \mathsf{Var}\ \mathsf{Expr} \\
 & \mid & \mathsf{Decls}\ \mathsf{Expr} \\
\mathsf{Decls} & \longrightarrow & \mathsf{Decl}\ \mathsf{Decls} \\
 & \mid & \varepsilon \\
\mathsf{Decl} & \longrightarrow & \mathsf{Var}\ \mathsf{Expr} \\
\mathsf{Var} & \longrightarrow & \mathsf{String}
\end{array}$$

$$\begin{array}{llll}
\mathbf{DATA}\ \mathsf{Root} & \mid & \mathit{Root} & \mathsf{Expr} \\
\mathbf{DATA}\ \mathsf{Expr} & \mid & \mathit{Var} & \mathsf{Var} \\
 & \mid & \mathit{App} & \mathit{fun} : \mathsf{Expr}\ \mathit{arg} : \mathsf{Expr} \\
 & \mid & \mathit{Lam} & \mathsf{Var}\ \mathsf{Expr} \\
 & \mid & \mathit{Let} & \mathsf{Decls}\ \mathsf{Expr} \\
\mathbf{TYPE}\ \ \mathsf{Decls} = [\mathsf{Decl}] & & & \\
\mathbf{DATA}\ \mathsf{Decl} & \mid & \mathit{Decl} & \mathsf{Var}\ \mathsf{Expr} \\
\mathbf{DATA}\ \mathsf{Var} & \mid & \mathit{Ident} & \mathit{name} : \mathsf{String}
\end{array}$$

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
```

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
```

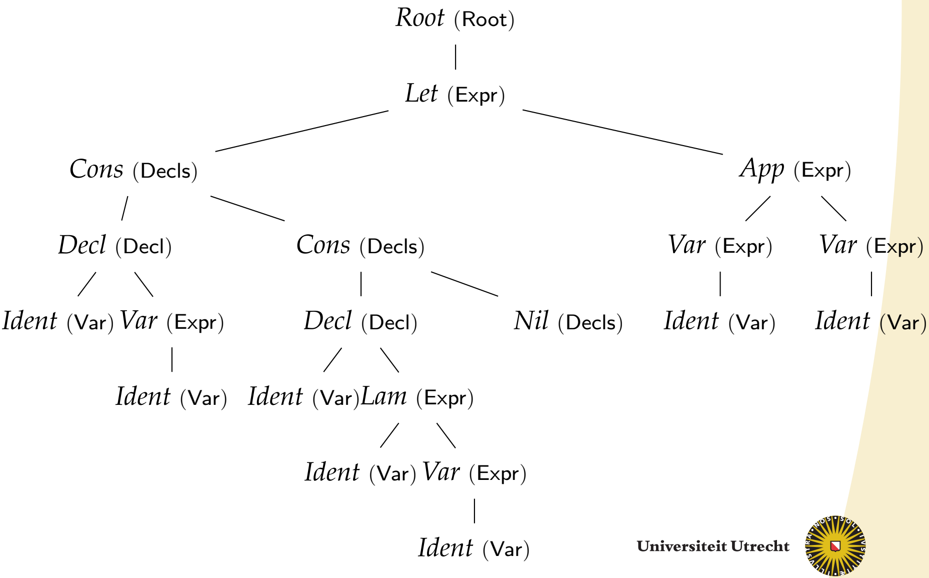
An example value

```
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)

App (Expr)

Decl (Decl)

Cons (Decls)

Var (Expr)

Var (Expr)

Ident (Var) *Var* (Expr)

Decl (Decl)

Nil (Decls)

Ident (Var)

Ident (Var)

Ident (Var)

Ident (Var) *Lam* (Expr)

Ident (Var) *Var* (Expr)

Ident (Var)

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 ∪ @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
   [ | | allvars : { [String] } ]
SEM Root
   | Root  lhs.allvars = @expr.allvars
SEM Expr
   | Var   lhs.allvars = @var.allvars
   | 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
  [ | | 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
   [ | | allvars : {[String]} USE {∪} {[]}]
```

```
SEM Var
   | Ident lhs.allvars = [@name]
```

Synthesised attribute computation in UUAG

ATTR Root Expr Decls Decl Var
  $[\ |\ |\ \mathit{allvars} : \{[\mathsf{String}]\}$ **USE** $\{\cup\}\ \{[\,]\}]$

SEM Var
  $|\ \mathit{Ident}$ **lhs**.$\mathit{allvars} = [@\mathit{name}]$

Synthesised attribute computation in UUAG

ATTR $*$
$[\ |\ |\ \mathit{allvars} : \{[\mathsf{String}]\}\ \mathbf{USE}\ \{\cup\}\ \{[\,]\}]$

SEM Var
$|\ \mathit{Ident}\ \mathbf{lhs}.\mathit{allvars} = [@\mathit{name}]$

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

> **SET** All = Root Expr Decls Decl Var
>
> *
>
> -- implicitly defined All, contains all **DATA** types in scope
>
> **SET** D = Decls Decl
>
> All − D
>
> -- set difference
>
> Root → Var
>
> -- all nonterminals on paths from Root to Var, excluding Root

- Such sets can be used as arguments to **ATTR** and **SEM**.

Combining computations

- Attributes can (mutually) depend on each other.

```
ATTR *
  [ | | freevars : {[String]} USE {∪} {[]}]
ATTR D
  [ | | defvars : {[String]} USE {++} {[]}]
SEM Var
  | Ident lhs.freevars = [@name]
SEM Expr
  | Lam   lhs.freevars = @expr.freevars − @var.freevars
  | Let   lhs.freevars = (@expr.freevars ∪ @decls.freevars)
                           − @decls.defvars
SEM Decl
  | Decl  lhs.freevars = @expr.freevars    -- overriding USE
          lhs.defvars  = @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)
  [substenv : {FiniteMap Var Expr} | | ]
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

```
ATTR Root (Root → Expr)
  [substenv : {FiniteMap Var Expr} | | ]
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

```
ATTR Root (Root → Expr)
  [substenv : {FiniteMap Var Expr} | | ]


SEM Expr


  | Lam  expr.substenv  = delListFromFM @lhs.substenv @var.freevars
  | Let  loc.substenv   = 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., the 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.

```
ATTR * − Root
   [ | | substituted : SELF ]
ATTR Root
   [ | | substituted : Expr ]
ATTR Expr
    | Var lhs.substituted = case lookupFM @lhs.substenv
                                                   @var.substituted of
                                    Just expr  → expr
                                    Nothing    → 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.

Haskell: higher-order functions

- 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.

$$
\begin{aligned}
&(+) && :: \mathit{Int} \rightarrow (\mathit{Int} \rightarrow \mathit{Int}) \\
&(+)\ 2 && :: \mathit{Int} \rightarrow \mathit{Int} \\
&(+)\ 2\ 3 && :: \mathit{Int}
\end{aligned}
$$

- Functions can be arguments of functions.

$$
\begin{aligned}
&\mathit{twice} && :: (a \rightarrow a) \rightarrow (a \rightarrow a) \\
&\mathit{twice}\ f\ x && = f\ (f\ x) \\
&\mathit{twice}\ ((+)\ 17)\ 8 == 42 && \\
&\mathit{map} && :: (a \rightarrow b) \rightarrow ([a] \rightarrow [b]) \\
&\mathit{map}\ f\ [\,] && = [\,] \\
&\mathit{map}\ f\ (x : \mathit{xs}) && = f\ x : \mathit{map}\ f\ \mathit{xs}
\end{aligned}
$$

Catamorphisms

- 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 occurences by recursive calls to the catamorphism
- Since Haskell provides algebraic data types, catamorphisms can be written easily in Haskell.
- Sythesised attributes can be translated into “catamorphic form” in a straight-forward way.

Example translation

```
allvars_Root                   :: Root → [String]
allvars_Root (Root expr)       = allvars_Expr expr

allvars_Expr                   :: Expr → [String]
allvars_Expr (Var var)         = allvars_Var var
allvars_Expr (App fun arg)     = let fun_allvars = allvars_Expr fun
                                     arg_allvars = allvars_Expr arg
                                 in  fun_allvars ∪ arg_allvars
...
allvars_Var                    :: Var → [String]
allvars_Var (Ident name)       = [name]
```

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   lhs.freevars = (@expr.freevars ∪ @decls.freevars)
                            − @decls.defvars

SEM Decl
  | Decl lhs.freevars = @expr.freevars    -- overriding USE
         lhs.defvars  = @var.freevars
```

```
sem_Expr                      :: Expr → [String]
sem_Expr (Let decls expr) =
  let (decls_defvars, decls_freevars) = sem_Decls decls
      expr_freevars                   = sem_Expr expr
  in  (expr_freevars ∪ decls_freevars)
        − (decls_freevars)
sem_Decl                      :: Decl → ([String], [String])
sem_Decl (Decl var expr)  =
  let var_freevars                    = sem_Var var
      expr_freevars                   = sem_Expr expr
  in  (var_freevars, expr_freevars)
```

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

  > $\mathit{sem\_Type} :: \mathsf{Type} \rightarrow \mathsf{Sem\_Type}$

  where Sem_Type is a type synonym for a functional type, mapping all inherited attributes to the synthesised attributes for Type:

  > $$\begin{aligned}\textbf{type}\ \mathsf{Sem\_Type} = \mathsf{Inh}_1 \rightarrow \mathsf{Inh}_2 \rightarrow \cdots \rightarrow \mathsf{Inh}_m \\ \rightarrow (\mathsf{Syn}_1, \mathsf{Syn}_2, \ldots, \mathsf{Syn}_n)\end{aligned}$$

Translating "substitution"

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

```
type Sem_Expr = FiniteMap Var Expr → [String], Expr
sem_Expr :: Expr → Sem_Expr
sem_Expr (Lam var expr) lhs_substenv =
   let (var_freevars, var_substituted)
            = sem_Var var lhs_substenv
       (expr_freevars, expr_substituted)
            = sem_Var var (delListFromFM lhs_substenv var_freevars)
   in  Lam var_substituted expr_substituted  {- SELF default -}
sem_Expr (Var var) lhs_substenv =
   let (var_freevars, var_substituted)
            = sem_Var var lhs_substenv
   in  case lookupFM . . .
```

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: $\mathit{True} \vee \mathit{error}$ `"unreachable"`
- Lazy evaluation allows dealing with infinite data structures, as long as only a finite part is used in the end.

$$\begin{aligned}
&\mathit{primes} && :: [\mathit{Int}] \\
&\mathit{primes} && = \mathit{sieve}\ [2\,..] \\
&\mathit{sieve} && :: [\mathit{Int}] \rightarrow [\mathit{Int}] \\
&\mathit{sieve}\ (x : \mathit{xs}) && = x : \mathit{sieve}\ [y \mid y \leftarrow \mathit{xs}, y\ \text{‘}\mathit{mod}\text{‘}\ x \not\equiv 0] \\
&\mathit{take}\ 100\ \mathit{primes} &&
\end{aligned}$$

- 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: $\mathit{True} \lor \mathit{error}$ `"unreachable"`
- Lazy evaluation allows dealing with infinite data structures, as long as only a finite part is used in the end.

$$\begin{aligned}
&\mathit{primes} &&:: [\mathit{Int}]\\
&\mathit{primes} &&= \mathit{sieve}\ [2\mathinner{..}]\\
&\mathit{sieve} &&:: [\mathit{Int}] \rightarrow [\mathit{Int}]\\
&\mathit{sieve}\ (x : \mathit{xs}) &&= x : \mathit{sieve}\ [y \mid y \leftarrow \mathit{xs}, y\ `\mathit{mod}`\ x \not= 0]
\end{aligned}$$

$$\mathit{take}\ 100\ \mathit{primes}$$

- 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.

Full copy rule (contd.)

- 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

```
DATA Root | Root Tree
DATA Tree | Leaf  label : Int
          | Node left : Leaf right : Leaf
ATTR Tree [ | levels : [Int] | bft : SELF]
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
```

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
```

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.