The UU_AG System

Programming with Functions, Aspects, Attributes, and Catamorphisms

Andres Löh

Institute of Information and Computing Science
Utrecht University
e-mail: andres@cs.uu.nl

June 27, 2001

(http://www.cs.uu.nl/~andres/talk3.ps)
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 abstract syntax tree is best described by a grammar.
- A concrete value (program) is then a word of the language defined by that grammar.

\[
\text{Decimal} \quad \rightarrow \quad \text{Sign Digits} \\
\quad | \quad \text{Digits}
\]

- The rules in a grammar are called \textit{productions}. The right hand side of a rule is \textit{derivable} from the left hand side.
- In each production a \textit{nonterminal} is replaced by (\textit{terminals} and/or) other nonterminals.
- A word is in the language defined by the grammar if it is derivable from the \textbf{root symbol} (or root nonterminal) in a finite number of steps.
- For convenience, we will always name the root symbol \textit{Root}. 
An example grammar

The following grammar describes the abstract syntax of a very simple language:

\[
\begin{align*}
\text{Root} & \rightarrow \text{Exprs} \\
\text{Exprs} & \rightarrow \text{Expr} \text{ Exprs} \\
& \quad | \quad \varepsilon \\
\text{Expr} & \rightarrow \text{Term} \\
\text{Term} & \rightarrow \text{String} \\
& \quad | \quad \text{Term} \text{ Term}
\end{align*}
\]

- A program is a list of expressions.
- Each expression is a term.
- A term is either a string, or a concatenation of multiple strings.
Properties of Haskell I: Algebraic data types

• Haskell provides a powerful language construct to define own data types.
• Choice can be represented by introducing different constructors.
• Constructors may contain fields.
• It is possible to define type constructors by the introduction of type variables.
• It is possible to define recursive types.

```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]
```
Grammars correspond to datatypes

- Given this power, each nonterminal can be seen as a data type.
- The productions can be translated into definitions.
- Constructor names have to be invented.
- Abstraction is not needed, but recursion is.
The example grammar translated

\[
\begin{align*}
\text{Root} & \rightarrow \text{Exprs} \\
\text{Exprs} & \rightarrow \text{Expr} \; \text{Exprs} \\
& \mid \varepsilon \\
\text{Expr} & \rightarrow \text{Term} \\
\text{Term} & \rightarrow \text{String} \\
& \mid \text{Term} \; \text{Term}
\end{align*}
\]

\[
\begin{align*}
\text{DATA Root} & \mid \text{Root} \; \text{Exprs} \\
\text{DATA Exprs} & \mid \text{Cons} \; \text{hd} : \text{Expr} \; \text{tl} : \text{Exprs} \\
& \mid \text{Nil} \\
\text{DATA Expr} & \mid \text{Simple} \; \text{Term} \\
\text{DATA Term} & \mid \text{Single} \; \text{String} \\
& \mid \text{Concat} \; \text{left} : \text{Term} \; \text{right} : \text{Term}
\end{align*}
\]

- Data type definitions in UU_AG syntax are very similar (and, in fact, translated into) Haskell data type definitions.
- Fields may be given field names.
- (Contrary to Haskell, UU_AG constructor names do not have to be unique.)
An example program

```
Root
  /\      \
Cons  Exprs
  /   \
Simple  Expr
  /     \
Single  Term
       "$Haskell$"

Cons  Exprs
  /\    \
Simple  Expr
  /   \
Nil  Exprs
    /\    \
Concat  Term
  /   \
Single  Term
       "$AG$"

(Single  Term)

"s are cool!"
```
Computation follows structure I: Total length
Computation follows structure II: Maximum length

```
Root
  ↓
Cons
  ↓
Simple
  ↓
Single
  ↓
"Haskell"

Cons
  ↓
Simple
  ↓
Concat
  ↓
Single
  ↓
"AG"

Cons
  ↓
Simple
  ↓
Nil
  ↓
0
```

- Single
  - 7
  - "Haskell"

- Cons
  - 13

- Simple
  - 7
  - "AG"

- Nil
  - 0

- Concat
  - 13

- Single
  - 11
  - "s are cool!"

- Root
  - 13

- Cons
  - 13
  - 0

- Root
  - 13
  - 0
Computation follows structure III: Spaces?

The UU_AG System — Programming with Functions, Aspects, Attributes, and Catamorphisms
Computation follows structure IV: Value of last term
Computation follows structure — Observations

- Information is passed upwards.
- Constructors are replaced by operations.
- In many cases information is just copied unchanged.
Synthesised attributes

- In UU\textsubscript{AG}, computations are modelled by attributes.
- Each of the examples defines an attribute.
- Attributes that are computed in a bottom-up fashion are called synthesised attributes.

\[
\text{ATTR} \quad \text{Exprs} \quad \text{Expr} \quad \text{Term} \quad [\mid | \text{maxlen} : \text{Int}] \\
\text{SEM} \quad \text{Term} \\
\quad | \quad \text{Single lhs.maxlen} \quad = \quad \text{length string} \\
\quad | \quad \text{Concat lhs.maxlen} \quad = \quad \text{left.maxlen} + \text{right.maxlen} \\
\text{SEM} \quad \text{Expr} \\
\quad | \quad \text{Simple lhs.maxlen} \quad = \quad \text{term.maxlen} \\
\text{SEM} \quad \text{Exprs} \\
\quad | \quad \text{Cons lhs.maxlen} \quad = \quad \text{max hd.maxlen tl.maxlen} \\
\quad | \quad \text{Nil lhs.maxlen} \quad = \quad 0
\]
Synthesised attributes — continued

- Different attributes (and their semantics) can be defined separately, but can interact (be defined in terms of other attributes).
- The UU_AG system provides **copy rules** to eliminate trivial equations.

\[
\text{SEM } \text{Exprs} \mid\mid \text{isEmpty} : \text{Bool} \\
| \text{Cons } \text{lhs.isEmpty} = \text{False} \\
| \text{Nil } \text{lhs.isEmpty} = \text{True} \\
\]

\[
\text{ATTR } \text{Exprs} \text{ Expr Term} \mid\mid \text{lastval} : \text{String} \\
\text{SEM } \text{Term} \\
| \text{Single } \text{lhs.lastval} = \text{string} \\
| \text{Concat } \text{lhs.lastval} = \text{left.lastval} + \text{right.lastval} \\
\text{SEM } \text{Exprs} \\
| \text{Cons } \text{lhs.lastval} = \text{if } \text{tl.isEmpty} \text{ then } \text{hd.lastval} \text{ else } \text{tl.lastval} \\
| \text{Nil } \text{lhs.lastval} = \text{error "no term in program"} \\
\]
Distributing information

- Sometimes synthesised attributes depend on outside information.
- Examples: Options, parameters, results of other computations.
- In these cases it is not sufficient to pass information bottom-up. We need top-down attributes, too!
Example: Joining strings

```
Root
  " and "
  Cons
  " and "
  Simple
  Single
    "Haskell"
  Cons
  Cons
    " and "
    Single
      "AG"
    Simple
    Concat
      " and "
      Single
      "s are cool!"
    Nil
```
Example: Joining strings — continued

"Haskell and AGs are cool!" and "

"Haskell" and "AGs are cool!"

"Haskell" and "AGs are cool!"

"Haskell" and "AGs are cool!"

"Haskell" and "AGs are cool!"

"AG" and "s are cool!"

"AG" and "s are cool!"

"AG" and "s are cool!"
Inherited attributes

- In attribute grammars, top-down attributes are called **inherited attributes**.
- In UU_AG, inherited attributes can be defined with the help of the `ATTR` and `SEM` statements, just like synthesised attributes.
- Again, for the downward distribution of inherited attributes there are copy rules that save some typing.
- Attributes can be inherited and synthesised at the same time. They are then called **chained attributes**.

```
SEM Root
  | Root exprs.joinword = "\u2227 and \u2227"

SEM Exprs [joinsep : String || joinval : String]
  | Cons tl.joinsep = lhs.joinsep
  |   lhs.joinval = if tl.isEmpty
                  then hd.lastval
                  else hd.lastval + lhs.joinsep + tl.joinval

  | Nil lhs.joinval = ""
```
Properties of Haskell II: 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.

(+) :: Int \rightarrow (Int \rightarrow Int)
(+) 2 :: Int \rightarrow Int
(+) 2 3 :: Int

• Functions can be arguments of functions.

twice :: (a \rightarrow a) \rightarrow (a \rightarrow a)
twice f x = f (f x)
twice ((+) 17) 8 \equiv 42

map :: (a \rightarrow b) \rightarrow ([a] \rightarrow [b])
map f [] = []
map f (x : xs) = f x : map f xs
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 occurrences by recursive calls to the catamorphism
- Since Haskell provides algebraic data types, catamorphisms can be written easily in Haskell.
- Synthesised attributes can be translated into catamorphisms in a straight-forward way.
**Example translation**

\[
\text{maxlen\_Root} \quad :: \quad \text{Root} \rightarrow \text{Int} \\
\text{maxlen\_Root} \ (\text{Root exprs}) \quad = \quad \text{maxlen\_Exprs} \ \text{exprs} \\
\text{maxlen\_Exprs} \quad :: \quad \text{Exprs} \rightarrow \text{Int} \\
\text{maxlen\_Exprs} \ (\text{Cons} \ hd \ tl) \quad = \quad \text{let} \ \text{hd\_maxlen} = \ \text{maxlen\_Expr} \\
\phantom{\text{maxlen\_Exprs} \ (\text{Cons} \ hd \ tl) \quad =} \ \text{tl\_maxlen} = \ \text{maxlen\_Exprs} \\
\phantom{\text{maxlen\_Exprs} \ (\text{Cons} \ hd \ tl) \quad =} \ \text{in} \ \text{max} \ \text{hd\_maxlen} \ \text{tl\_maxlen} \\
\text{maxlen\_Exprs} \ \text{Nil} \quad = \quad 0 \\
\text{maxlen\_Expr} \quad :: \quad \text{Expr} \rightarrow \text{Int} \\
\text{maxlen\_Expr} \ (\text{Simple} \ \text{term}) \quad = \quad \text{maxlen\_Term} \ \text{term} \\
\text{maxlen\_Term} \quad :: \quad \text{Term} \rightarrow \text{Int} \\
\text{maxlen\_Term} \ (\text{Single} \ \text{string}) \quad = \quad \text{length} \ \text{string} \\
\text{maxlen\_Term} \ (\text{Concat} \ \text{left} \ \text{right}) \quad = \quad \text{let} \ \text{left\_maxlen} = \ \text{maxlen\_Term} \\
\phantom{\text{maxlen\_Term} \ (\text{Concat} \ \text{left} \ \text{right}) \quad =} \ \text{right\_maxlen} = \ \text{maxlen\_Term} \\
\phantom{\text{maxlen\_Term} \ (\text{Concat} \ \text{left} \ \text{right}) \quad =} \ \text{in} \ \text{left\_maxlen} + \ \text{right\_maxlen}
\]
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

\[
\begin{align*}
\text{SEM} \ E x p r s & \ [\mid \ isE m p t y : \ B o o l \ \ l a s t v a l : \ S t r i n g ] \\
| \ C o n s \ l h s . i s E m p t y & = \ True \\
\quad \ l h s . l a s t v a l & = \ \text{if} \ t l . i s E m p t y \ \text{then} \ h d . l a s t v a l \\
& \quad \quad \quad \quad \quad \ \text{else} \ t l . l a s t v a l \\
\text{sem}_\text{Exprs} & :: \ E x p r s \rightarrow (B o o l , \ S t r i n g ) \\
\text{sem}_\text{Exprs} \ (C o n s \ h d \ t l ) & = \ \text{let} \ (t l . i s E m p t y , t l . l a s t v a l ) = \text{sem}_\text{Exprs} \ t l \\
& \quad \quad \quad \quad \quad \quad h d . l a s t v a l = \text{sem}_\text{Expr} \ h d \\
& \quad \quad \quad \quad \quad \quad \text{in} \ (F a l s e \\
& \quad \quad \quad \quad \quad \quad , \ \text{if} \ t l . i s E m p t y \ \text{then} \ h d . l a s t v a l \\
& \quad \quad \quad \quad \quad \quad \quad \quad \quad \ \text{else} \ t l . l a s t v a l \\
& \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad )
\end{align*}
\]
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 inherited attributes get mapped to the synthesised attributes.
Catamorphisms can compute functions! — continued

\[
\text{SEM}\;\text{Exprs}\;[\text{joinsep} : \text{String} \ || \ \text{joinval} : \text{String}]
\]

\[
|\text{Cons}\;\text{tl}.\text{joinsep} \quad = \quad \text{lhs}.\text{joinsep} \\
\text{lhs}.\text{joinval} \quad = \quad \text{if}\;\text{tl}.\text{isEmpty} \\
\quad \text{then}\;\text{hd}.\text{lastval} \\
\quad \text{else}\;\text{hd}.\text{lastval} \;+\; \text{lhs}.\text{joinsep} \;+\; \text{tl}.\text{joinval}
\]

\[
\text{sem}_\text{Exprs} :: \quad \text{Exprs} \rightarrow \frac{\text{String} \rightarrow (\text{Bool}, \text{String}, \text{String})}{}
\]

\[
\text{sem}_\text{Exprs} \;\text{(Cons} \;\text{hd} \;\text{tl}) \\
\quad \text{lhs}_\text{joinsep} \quad = \quad \text{let} \; (\text{tl}_\text{isEmpty} \\
\quad \quad \quad , \; \text{tl}_\text{lastval} \\
\quad \quad \quad , \; \text{tl}_\text{joinval} \\
\quad \quad ) = \text{sem}_\text{Exprs} \;\text{tl} \;\text{lhs}_\text{joinsep} \\
\quad \quad \text{hd}_\text{lastval} = \text{sem}_\text{Expr} \;\text{hd} \\
\quad \text{in} \; (\text{False} \\
\quad \quad \text{if} \;\text{tl}_\text{isEmpty} \\
\quad \quad \quad \text{then} \;\text{hd}_\text{lastval} \\
\quad \quad \quad \text{else} \;\text{hd}_\text{lastval} \;+\; \text{lhs}_\text{joinsep} \;+\; \text{tl}_\text{joinval} \\
\quad)
\]
Implementation of UU_AG

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

- There is just one (very general) copy rule.
- Attributes are identified by name.
- If an explicit rule for a specific attribute is missing, it is copied from the “nearest” node (in the picture) that provides that attribute.
The copy rules for the distribution of inherited and the collection of synthesised attributes are special cases of the general copy rule.
Tree traversals made easy I: Preliminaries

\[
\begin{align*}
\text{DATA} & \quad \text{Root} & | & \quad \text{Root Tree} \\
\text{DATA} & \quad \text{Tree} & | & \quad \text{Empty} \\
& & | & \quad \text{Node left: Tree right: Tree}
\end{align*}
\]
Tree traversals made easy II: DFL

- The nodes should be uniquely labelled (in depth-first order).
- Useful for unique counters, building and changing environments corresponding to the order of the statements in the input code.

\[
\text{SEM } \text{Root} \\
\quad | \text{Root tree.label} = 0 \\
\text{SEM } \text{Tree [ [ label : Int ] ]} \\
\quad | \text{Node left.label} = \text{lhs.label} + 1
\]
Tree traversals made easy II: DFL example
Tree traversals made easy II: DFL example — continued
Properties of Haskell III: Lazy evaluation

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

\[
\begin{align*}
\text{primes} &:: [\text{Int}] \\
\text{primes} &= \text{sieve} [2..] \\
\text{sieve} &:: [\text{Int}] \rightarrow [\text{Int}] \\
\text{sieve} (x : xs) &= x : \text{sieve} [y | y \leftarrow xs, y \mod x \neq 0] \\
\text{take 100 primes}
\end{align*}
\]
Tree traversals made easy III: BFL

- 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 for scoping issues.
- Basic Idea: Provide a list with initial counter values for each level, return a list with final counter values for each level.

\[
\text{SEM } \text{Root} \\
\begin{align*}
| \text{Root } \text{tree.blabs} & = 0 : \text{tree.blabs} \\
\text{SEM } \text{Tree} & [ [ \text{labels} : [\text{Int}] ] ] \\
| \text{Node } \text{loc.blabs} & = \text{head } \text{lhs.blabs} \\
\text{left.blabs} & = \text{tail } \text{lhs.blabs} \\
\text{lhs.blabs} & = (\text{loc.blabs} + 1) : \text{right.blabs}
\end{align*}
\]
Tree traversals made easy III: BFL example
Extending the string example with variables

- Allow assignments to variables.
- Allow usage of variables.
- Variables should be visible globally.

```haskell
DATA Expr |
Assign var : String Expr

DATA Term |
Var var : String

ATTR Exprs Expr Term [vardist : Environment | varcollect : Environment ]
```

- We store mappings of variables to string literals in an environment.
- Environments are given here as an abstract data type.

```haskell
empty :: Environment
isDefined :: String -> Environment -> Bool
lookup :: String -> Environment -> Maybe String
add :: (String, String) -> Environment -> Environment
merge :: Environment -> Environment -> Environment
```
Extending the string example with variables — continued

SEM Root
| Root exprs.varcollect exprs.vardist = empty

SEM Expr
| Assign expr.varcollect = if isDefined var lhs.varcollect
  then error "non-unique variable name"
  else add (var, expr.lastval) lhs.varcollect

SEM Term
| Var lhs.lastval = case lookup var lhs.vardist of
  Nothing → error "unknown variable"
  Just x → x
Extending the string example with groups

- Allow a list of expressions to be grouped.
- Outer variables can be used in a group, but inner variables are local.
- An inner variable can “shadow” an outer variable of the same name.

\[
\begin{align*}
\text{DATA} & \quad \text{Expr} & | & \quad \text{Group Exprs} \\
\text{SEM} & \quad \text{Expr} & | & \quad \text{Group exprs.varcollect} = \text{empty} \\
& & | & \quad \text{lhs.varcollect} = \text{lhs.varcollect} \\
& & | & \quad \text{exprs.vardist} = \text{merge lhs.varcollect exprs.varcollect}
\end{align*}
\]
Aspects can be separated

The UU_AG system allows to freely mix two styles of programming:

- Attribute (i.e. aspect) oriented: Define the semantics of an attribute in one place.
- Data oriented: Define the attributes of a data type in one place.

The first one is usually difficult to realise in ordinary programming languages.
Work in progress

- Static analysis: circularity, dependencies, strictification
- Language independency
- Higher-order attributes
- Type checking
Acknowledgements

• The UU_AG system has originally been designed by S. Doaitse Swierstra and Pablo Azero.
• The current implementation has been developed by Arthur Baars and Andres Löh. Further development is coordinated by Arthur Baars. This version will soon be available via http://www.cs.uu.nl.