Plug-and-play attribute grammars

Wouter Swierstra
wouter@swierstra.net

Universiteit Utrecht

August 21, 2004
Beyond the UUAG

The UUAG works nicely, but is far from perfect.

- Typing deferred to Haskell compiler
- Occassional tweaking of generated code
- Fixed functionality
- Limited abstractions
The next step...

- First-class attribute grammars (Oege de Moor, Kevin Backhouse, Doaitse Swierstra)
- Fighting TREX (Doaitse Swierstra and Pablo Azero)
- Template Haskell wizardry
- Embedded attribute grammars
data Tree = Leaf Int | Node Tree Tree

data Root = Root Tree

valSyn f = aspect
| (Leaf i)      lhs.val = i
| (Node l r)   lhs.val = f @l.val @r.val

compMaxAG = valSyn max
compMax    = knit compMaxAG
minInh = aspect
    | (Node l r) l.min = @lhs.min
    | (Node l r) r.min = @lhs.min
    | (Root t) t.min = @t.val

resSyn = aspect
    | (Leaf i) lhs.res = Leaf @lhs.min
    | (Node l r) lhs.res = Node @l.res @r.res

repMinAG = valSyn min ‘plug‘ minInh ‘plug‘ resSyn
repMin = knit repMinAG
Goals

We want to:

▶ define a semantic rule - that refers to other attributes.
▶ plug aspects together.
▶ generate semantic functions.

Taking into account that:

▶ all attribute definition are well-typed
▶ no missing attribute definitions
▶ no undefined attributes
▶ no multiple attribute definitions
The theory of qualified types

- A general framework for type systems
- Generalization of Haskell’s type classes
- Qualified types have the form \( \pi \Rightarrow \tau \)
- You get to introduce predicates \( \pi \ldots \)
- \( \ldots \) and show how to solve them
- \( \ldots \) and get soundness and completeness for free.
Extendible records

- Extendible records (Gaster and Jones)
- Rows are a special kind:
  - \{\} :: row
  - \{ l :: _ | _ \} :: * \to row \to row
  - Rec :: row \to *
- Rows have their own unification rules:
  - \{ l_1 :: \tau_1, l_2 :: \tau_2 \} = \{ l_2 :: \tau_2, l_1 :: \tau_1 \}
- A special predicate to describe when a label is not present: \( r \setminus a \)
- Functions for extending records and selecting fields:
  - \((l = _ | _) :: r \setminus l \Rightarrow a \to \text{Rec } r \to \text{Rec } \{ l :: a | r \}\)
  - \((_.l) :: r \setminus l \Rightarrow \text{Rec } \{ l :: a | r \} \to a\)
Attribute grammars

- Rows represent attribute grammar definitions.

  | Prod t.attr = e

- Labels contain information about:
  - production
  - attribute name
  - synthesized vs. inherited

- Functions similar to record extension define a single semantic rule.

- Separate wrapper around rows:
  - Aspect ::: row → *
Built-in predicates

\[ \pi_G ::= r \text{ def syn } \text{attr} :: \tau \text{ on } nt \]
\[ | \quad r \text{ def inh } \text{attr} :: \tau \text{ on } nt \]

Note that:

- \( r \) is a row defining an attribute
- \( \tau \) is the type of the attribute
- \( nt \) is the non-terminal that is attributed
Predicates

\[ \pi ::= r \setminus \text{attr} \quad r \text{ lacks the field attr} \]
\[ \quad | \quad r_1 \; r_2 \text{ partition } r \quad r \text{ can be partitioned in } r_1 \text{ and } r_2 \]
\[ \quad | \quad \text{knit } r \text{ to } nt \; i \; s \quad r \text{ defines a semantic function from } i \text{ to } s \text{ on the non-terminal } nt \]

- Now we have to define predicate entailment...
Predicate entailment - lacks

\[ P, \pi \vdash \pi \]

\[ P \vdash \{ || \} \ \text{attr} \]

\[ P \vdash \text{attr}' \neq \text{attr} \quad P \vdash r \ \text{\textbackslash attr} \]

\[ P \vdash \{ \text{attr}' :: \tau \mid r \} \ \text{\textbackslash attr} \]
Predicate entailment - partition

\[
P \models \{\} \ r \ \text{partition} \ r
\]

\[
P \models r_1 \backslash \text{attr} \quad P \models r_2 \backslash \text{attr} \quad P \models r_3 \backslash \text{attr} \quad P \models r_1 \ r_2 \ \text{partition} \ r_3
\]

\[
P \models \{ \text{attr} :: \tau \mid r_1 \} \ r_2 \ \text{partition} \ \{ \text{attr} :: \tau \mid r_3 \}
\]
Predicate entailment - knitting - 1

\[
\text{knit } \{ \} \text{ to } nt \{ \} \{ \}
\]

\[
d \text{ def syn } a :: \tau \text{ on } nt \quad d \text{ r partition } ag \quad \text{knit } r \text{ to } nt \quad i \quad s
\]

\[
\text{knit } ag \text{ to } nt \quad i \quad \{ \quad a :: \tau \quad | \quad s \quad \}
\]

\[
d \text{ def inh } a :: \tau \text{ on } nt \quad d \text{ r partition } ag \quad \text{knit } r \text{ to } nt \quad i \quad s
\]

\[
\text{knit } ag \text{ to } nt \quad \{ \quad a :: \tau \quad | \quad i \quad \} \quad s
\]
Eat the elephant a bite at a time!
Predicate entailment - knitting - II

\[ d \text{ def } \text{syn} a :: \tau \text{ on } nt' \quad d \text{ r partition } ag \quad \text{knit } r \text{ to } nt \quad is \]
\[ \text{knit } ag \text{ to } nt \quad is \]

\[ d \text{ def } \text{inh} a :: \tau \text{ on } nt' \quad d \text{ r partition } ag \quad \text{knit } r \text{ to } nt \quad is \]
\[ \text{knit } ag \text{ to } nt \quad is \]

Eat the elephant a bite at a time!
Plug-and-play attribute grammars

- It is now easy to type *plug* and *knit*!

\[
\begin{align*}
\text{plug} &::= r_1 \ r_2 \ \text{partition} \ r \Rightarrow \text{Aspect} \ r_1 \rightarrow \text{Aspect} \ r_2 \rightarrow \text{Aspect} \ r \\
\text{knit} &::= ag \ \text{knit} \ nt \ i \ s \Rightarrow \text{Aspect} \ ag \rightarrow nt \rightarrow \text{Rec} \ i \rightarrow \text{Rec} \ s
\end{align*}
\]
What I haven’t talked about

- Undefined attributes
- How to define attributes
- Attributing polymorphic data-structures
- Predicate improvement
- Quality of error messages
- Compilation
- Defining copy-rules and other extensions
- Not everything is first-class
Conclusions

- Embedded attribute grammars are within grasp!
- Work in progress:
  - Currently being implemented in EH compiler
  - Paper for Science of Computer Programming
- Future work:
  - Still some issues to smooth out
  - Include copy rules and other UUAG features
  - Relation with dependent types and generic programming