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

Fantasy syntax - I

```
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
```


Fantasy syntax - II

```
minInh = aspect
  | (Node l r) l.min   = @lhs.min
               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 π ...
- ... and show how to solve them
- ... and get soundness and completeness for free.

Extendible records

- Extendible records (Gaster and Jones)
- Rows are a special kind:
  - $\{\!\}$ ::: *row*
  - $\{\!\{\ l :: \_ \mid \_\ \}\!\}$ ::: $* \longrightarrow \mathit{row} \longrightarrow \mathit{row}$
  - Rec ::: $\mathit{row} \longrightarrow *$
- 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: $r \backslash a$
- Functions for extending records and selecting fields:
  - $(l = \_ \mid \_) :: r \backslash l \Rightarrow a \longrightarrow \mathrm{Rec}\ r \longrightarrow \mathrm{Rec}\ \{\!\{\ l :: a \mid r\ \}\!\}$
  - $(\_.l) :: r \backslash l \Rightarrow \mathrm{Rec}\ \{\!\{\ l :: a \mid r\ \}\!\} \longrightarrow a$

Attribute grammars

- Rows represent attribute grammar definitions.

  $| \mathit{Prod}\ t.\mathit{attr} = e$

- Labels contain information about:
  - production
  - attribute name
  - synthesized vs. inherited
- Functions similar to record extension define a single semantic rule.
- Separate wrapper around rows:
  - $\mathrm{Aspect} ::: \mathit{row} \longrightarrow *$

Built-in predicates

$$\begin{array}{lcl} \pi_G & ::= & r \text{ def syn } attr :: \tau \text{ on } nt \\ & | & r \text{ def inh } attr :: \tau \text{ on } nt \end{array}$$

Note that:

- r is a row defining an attribute
- τ is the type of the attribute
- nt is the non-terminal that is attributed

Predicates

π	$::=$	$r \backslash attr$	r lacks the field $attr$
	$\mid$	$r_1\ r_2$ partition r	r can be partitioned in r_1 and r_2
	$\mid$	knit r to $nt\ i\ s$	r defines a semantic function from i to s on the non-terminal nt

- Now we have to define predicate entailment...

Predicate entailment - lacks

$$\frac{}{P, \pi \Vdash \pi}$$

$$\frac{}{P \Vdash \{\!\}\backslash attr}$$

$$\frac{P \Vdash attr' \neq attr \quad P \Vdash r\backslash attr}{P \Vdash \{\!|\ attr' :: \tau \mid r\ |\!\}\backslash attr}$$

Predicate entailment - partition

$$\frac{}{P \Vdash \{\!\{\}\!\}\ r \text{ partition } r}$$

$$\frac{P \Vdash r_1 \backslash attr \quad P \Vdash r_2 \backslash attr \quad P \Vdash r_3 \backslash attr \quad P \Vdash r_1\ r_2 \text{ partition } r_3}{P \Vdash \{\!\{\ attr :: \tau \mid r_1\ \}\!\}\ r_2 \text{ partition } \{\!\{\ attr :: \tau \mid r_3\ \}\!\}}$$

Predicate entailment - knitting - I

$$\frac{}{\text{knit } \{\!\}\ \text{to } nt\ \{\!\}\ \{\!\}}$$

$$\frac{d \text{ def syn } a :: \tau \text{ on } nt \quad d\ r \text{ partition } ag \quad \text{knit } r \text{ to } nt\ i\ s}{\text{knit } ag \text{ to } nt\ i\ \{\!|\ a :: \tau \mid s\ |\!\}}$$

$$\frac{d \text{ def inh } a :: \tau \text{ on } nt \quad d\ r \text{ partition } ag \quad \text{knit } r \text{ to } nt\ i\ s}{\text{knit } ag \text{ to } nt\ \{\!|\ a :: \tau \mid i\ |\!\}\ s}$$

Predicate entailment - knitting - II

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

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

Predicate entailment - knitting - II

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

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

Eat the elephant a bite at a time!

Plug-and-play attribute grammars

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

> *plug* :: *r1* *r2* partition *r* $\Rightarrow$ Aspect *r1* $\rightarrow$ Aspect *r2* $\rightarrow$ Aspect *r*
>
> *knit* :: *ag* knit *nt* *i* *s* $\Rightarrow$ Aspect *ag* $\rightarrow$ *nt* $\rightarrow$ Rec *i* $\rightarrow$ Rec *s*

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