AG Combinators (Fighting TREX)

Doaitse Swierstra, Pablo Azero
Informatica Instituut
Utrecht University
Contents of the talk

• Motivation: why aspect-oriented

• Successive refinement of the RepMin problem

• Further work

• Insights about type systems
Introduction: the RepMin problem

Given a Tree...

data Root = Root Tree

data Tree = Node Tree Tree
  | Leaf Int

example = Root (Node (Node (Node (Leaf 5) (Leaf 4))
                 (Node (Leaf 2) (Leaf 3))
                 )
               (Node (Node (Leaf 9) (Leaf 8))
                 (Node (Leaf 7) (Leaf 6))
                 )
               )


... compute a Tree with the minimum element in the leaves
First version: a traditional RepMin

eval_Root (Root tree ) =
   let ( ) = (eval_Tree tree )
   in ( )

eval_Tree (Node left right) =
   let ( ) = (eval_Tree left )
   ( ) = (eval_Tree right)
   in ( )
eval_Tree (Leaf i) =

• recursive walk over the structure
First version: a traditional RepMin

\[
\text{eval\_Root} \quad (\text{Root tree}) \\
= \quad \text{let} \ (\text{smin}) = (\text{eval\_Tree} \quad \text{tree}) \quad \text{in} \ (\quad) \\
\text{eval\_Tree} \quad (\text{Node left right}) \\
= \quad \text{let} \ (\text{lmin}) = (\text{eval\_Tree} \quad \text{left}) \\
\quad (\text{rmin}) = (\text{eval\_Tree} \quad \text{right}) \quad \text{in} \ (\text{lmin} \ `\min` \ text{rmin}) \\
\text{eval\_Tree} \quad (\text{Leaf i}) = \quad (\text{i})
\]

• synthesizing an attribute: the minimum
First version: a traditional RepMin

```haskell
eval_Root (Root tree) = let (smin, sres) = (eval_Tree tree) in ( )

eval_Tree (Node left right) = let (lmin, lres) = (eval_Tree left) (rmin, rres) = (eval_Tree right) in (lmin `min` rmin, Node lres rres)

eval_Tree (Leaf i) = (i, Leaf)
```

- synthesizing an attribute: the minimum
- synthesizing the shape tree
First version: a traditional RepMin

```haskell
eval_Root (Root tree) = let (smin, sres) = (eval_Tree tree) smin in ()

eval_Tree (Node left right) = \imin -> let (lmin, lres) = (eval_Tree undefined left) \imin (rmin, rres) = (eval_Tree undefined right) \imin in (lmin `min` rmin, Node lres rres)

eval_Tree (Leaf i) = \imin -> (i, Leaf imin)
```

- synthesizing an attribute: the minimum
- synthesizing the shape tree
- inheriting an attribute: the minimum
First version: a traditional RepMin

\[
\text{eval\_Root} \ (\text{Root tree}) = \\lambda () \rightarrow \text{let } (smin, sres) = (\text{eval\_Tree undefined tree}) \text{ smin} \\
\text{in } (sres = sres)
\]

\[
\text{eval\_Tree} \ (\text{Node left right}) = \\lambda \text{imin} \rightarrow \text{let } (lmin, lres) = (\text{eval\_Tree undefined left}) \text{ imin} \\
(lmin `\text{min}` rmin, Node lres rres)
\]

\[
\text{eval\_Tree} \ (\text{Leaf i}) = \\lambda \text{imin} \rightarrow (i, \text{Leaf imin})
\]

• output value in Rec (sres::Tree)
• the root will require also an inherited attribute
• dummy parameters for the grammar (subsequent versions)
Observations

• Data type declarations model the underlying context-free syntax.
• Semantic functions are defined “horizontally” (by production rule).
• Consequence of the above point: modifications require to scan the complete grammar.
• Lazy evaluation replaces scheduling of the attribute computations.
What do we want to achieve

Compositionality of attribute computations:

root: sem_Root
tree: sem_Node
    sem_Leaf

vs

imin : {root, node, leaf}
smin: {root, node, leaf}
sres : {root, node, leaf}
Traditional AG systems

... can achieve this goal but only at a syntactic level:

DATA Root | Root Tree
DATA Tree
  | Node left, right : Tree
  | Leaf Int

SEM Tree [-> smin: Int]
  | Leaf LHS.smin = int
  | Node LHS.smin = "left_smin `min` righth_smin"

SEM Root [->sres: Tree] | Root LHS.sres = "tree_sres"
SEM Tree [->sres: Tree]
  | Leaf LHS.res = "Leaf lhs_imin"
  | Node LHS.res = "Node left_res right_res"

ATTR Tree [imin: Int <-]
SEM Root | Root tree . minval = tree_smin
Recent Haskell extensions...

... introduce the possibility of embedding such compositionality in a library!

First solution...

... was conceived by Oege de Moor et al.
Second version: a record based solution

The idea:
• attribute computations: functions from input to output attributes
• semantic functions in the first version depend on the shape of the production rule and the attribute computations

```haskell
eval_Tree left right imin
  = let (lmin, lres) = left imin
      (rmin, rres) = right imin
      in (lmin `min` rmin, Node lres rres)
```
Second version

eval_Root \( g \) (Root tree)
=  \( (#\text{root } g ) \) (eval_Tree \( g \) tree )

eval_Tree \( g \) (Node left right)
=  \( (#\text{node } g ) \) (eval_Tree \( g \) left )
  (eval_Tree \( g \) right)

eval_Tree \( g \) (Leaf i)
=  \( (#\text{leaf } g \ i) \)

g () = ( root = rootf, node = nodef, leaf = leaff )

• A global \( g \) containing the attribute computations is passed down the structure
• \( g \) is an extensible record (TREX extension in Hugs)
Second version

eval_Root g (Root tree      )
  = knit1 (#root g  ) (eval_Tree g tree )
eval_Tree g (Node left right)
  = knit2 (#node g  ) (eval_Tree g left )
          (eval_Tree g right)
eval_Tree g (Leaf i         )
  = knit0 (#leaf g i)

\[ g () = ( \text{root} = \text{rootf}, \text{node} = \text{nodef}, \text{leaf} = \text{leaff}) \]

the knit functions generalize the attribute flow over the tree; they only depend on the shape of the production rule
knit0 f = \ pi -> let def = pi
    po = f def
  in (#po po)

knit1 f c = \ pi -> let def = (co,pi)
    co = c (#ci cipo)
    cipo = f def
  in (#po cipo)

knit2 f l r = \ pi -> let def = (lo,ro,pi)
    lo = l (#li liripo)
    ro = r (#ri liripo)
    liripo = f def
  in (#po liripo)
Second version: attribute comput.

\[
\text{rootf}(\text{co}, \pi) = (\text{ci} = (\text{imin} = (#\text{smin} \text{co})), \text{po} = (\text{sres} = (#\text{sres} \text{co})))
\]

\[
\text{type Tree_Inh} = \text{Rec} (\text{imin} :: \text{Int})
\]
\[
\text{type Tree_Syn} = \text{Rec} (\text{smin} :: \text{Int}, \text{sres} :: \text{Tree})
\]

\[
\text{nodef} :: \text{Rec} (\text{li} :: \text{Tree_Inh}, \text{ri} :: \text{Tree_Inh}, \text{po} :: \text{Tree_Syn})
\]

\[
\text{nodef} \ (\text{lo}, \text{ro}, \pi) = (\text{li} = (\text{imin} = #\text{imin} \pi), \text{ri} = (\text{imin} = #\text{imin} \pi), \text{po} = (\text{sres} = \text{Node} (#\text{sres} \text{lo}) (#\text{sres} \text{ro}), \text{smin} = (#\text{smin} \text{lo}) \text{`min`} (#\text{smin} \text{ro}))
\]

\[
\text{leaff} :: \text{Int} \to \text{Tree_Inh} \to \text{Rec} (\text{po} :: \text{Tree_Syn})
\]

\[
\text{leaff} \ i \ (\pi) = (\text{po} = (\text{sres} = \text{Leaf} (#\text{imin} \pi), \text{smin} = i)
\]
Third version: slicing into aspects

-- computing the minimum
nodef_smin_po (lo, ro, pi) (po=v|r)
  = (po = ( smin = (#smin lo) \text{\textbackslash '}min\text{\textbackslash '} (#smin ro)| v)| r)

leaff_smin_po i ( pi) (po=v|r)
  = (po = ( smin = i | v)| r)

• to reorganize in aspects we depart from the equations on the last slide by
• slicing them into single elements ...
• and since we want to be able to recover the original functions we make them into a function
Third version: slicing into aspects

-- computing the shape tree
rootf_sres_po (co, pi) (po=vlr)
= (po = (sres = #sres co l v) r)
nodef_sres_po (lo, ro, pi) (po=vlr)
= (po = (sres = Node (#sres lo) (#sres ro) l v) r)
leaff_sres_po i (pi) (po=vlr)
= (po = (sres = Leaf (#imin pi) l v) r)

-- distributing the minimum
rootf_imin_ci (co, pi) (ci=vlr)
= (ci = (imin = #smin co l v) r)
nodef_imin_li (lo, ro, pi) (li=vlr)
= (li = (imin = #imin pi l v) r)
nodef_imin_ri (lo, ro, pi) (ri=vlr)
= (ri = (imin = #imin pi l v) r)
Third version: bringing the pieces together

\[
\begin{align*}
\text{rootf}() & = \text{rootf\_sres\_po} `\text{ext}` \text{rootf\_imin\_ci} `\text{ext}` \ldots \\
\text{nodef}() & = \text{nodef\_sres\_po} `\text{ext}` \text{nodef\_smin\_po} `\text{ext}` \text{nodef\_imin\_li} `\text{ext}` \text{nodef\_imin\_ri} `\text{ext}` \ldots \\
\text{leaff}() i & = \text{leaff\_sres\_po} i `\text{ext}` \text{leaff\_smin\_po} i `\text{ext}` \ldots
\end{align*}
\]

```
-- and its basic machinery
f `\text{ext}\` g = \ inputs -> f inputs (g inputs)
```

\text{ext} composes single elements into the original definition
Third version: bringing the pieces together

\[
\text{rootf}() = \text{rootf}_{\text{sres}}_{\text{po}} `\text{ext}` \text{rootf}_{\text{imin}}_{\text{ci}} `\text{ext}` \text{new}_{\text{Rootf}}
\]

\[
\text{nodef}() = \text{nodef}_{\text{sres}}_{\text{po}} `\text{ext}` \text{nodef}_{\text{smin}}_{\text{po}} `\text{ext}` \text{nodef}_{\text{imin}}_{\text{li}} `\text{ext}` \text{nodef}_{\text{imin}}_{\text{ri}} `\text{ext}` \text{new}_{\text{Nodef}}
\]

\[
\text{leaff}() i = \text{leaff}_{\text{sres}}_{\text{po}} i `\text{ext}` \text{leaff}_{\text{smin}}_{\text{po}} i `\text{ext}` \text{new}_{\text{Leaff}} i
\]

-- ending the composed sequence

\[
\text{new}_{\text{Rootf}} = \text{const} (\text{ci} = \text{EmptyRec}, \text{po} = \text{EmptyRec})
\]

\[
\text{new}_{\text{Nodef}} = \text{const} (\text{li} = \text{EmptyRec}, \text{ri} = \text{EmptyRec}, \text{po} = \text{EmptyRec})
\]

\[
\text{new}_{\text{Leaff}} i = \text{const} (\text{po} = \text{EmptyRec})
\]
Fourth version: more abstractions for attribute definitions

We may extend the library with combinators to describe attribute definitions
Using products instead of records allows us to define more combinators, but first we redefine the knits:

\[
\begin{align*}
\text{knit0 } f \ &= \ \lambda \ pi \rightarrow \ \text{let} \ \text{def} \ = \ pi \\
& \quad \ \text{po} \ = \ f \ \text{def} \\
& \quad \ \text{in} \ \text{po} \\
\text{knit1 } f \ c \ &= \ \lambda \ pi \rightarrow \ \text{let} \ \text{def} \ = \ (c, pi) \\
& \quad \ \text{co} \ = \ c \ \text{ci} \\
& \quad \ (\text{ci, po}) \ = \ f \ \text{def} \\
& \quad \ \text{in} \ \text{po} \\
\text{knit2 } f \ l \ r \ &= \ \lambda \ pi \rightarrow \ \text{let} \ \text{def} \ = \ (l, r, pi) \\
& \quad \ \text{lo} \ = \ l \ (li) \\
& \quad \ \text{ro} \ = \ r \ (ri) \\
& \quad \ (li, ri, po) \ = \ f \ \text{def} \\
& \quad \ \text{in} \ \text{po}
\end{align*}
\]
Fourth version: new combinators

syndef₀ = \ f p -> f p

syndef₁ = \ f (in₁, p) -> (in₁, f p)
def₁₁₁ = \ f (in₁, p) -> (f in₁, p)

def₂₁₁ = \ f (in₁, in₂, p) -> (f in₁, in₂, p)

def₂₁₂ = \ f (in₁, in₂, p) -> (in₁, f in₂, p)

-- computing the minimum ...

nodef_smin_po (lo, ro, pi)
  = syndef₂ (\v -> (smin = (#smin lo) `min` (#smin ro)| v))
leaff_smin_po i (pi)
  = syndef₀ (\v -> (smin = i | v))
-- ... the shape tree ...

rootf_sres_po (co, pi)  
  = syndef1 (v -> (sres = #sres co | v))
nodef_sres_po (lo, ro, pi)  
  = syndef2 (v -> (sres = Node (#sres lo) (#sres ro) | v))
leaff_sres_po i (pi)  
  = syndef0 (v -> (sres = Leaf (#imin pi) | v))

-- ... and distributing the minimum

rootf_imin_ci (co, pi)  
  = def_1_1 (v -> (imin = #smin co | v))
nodef_imin_li (lo, ro, pi)  
  = def_2_1 (v -> (imin = #imin pi | v))
nodef_imin_ri (lo, ro, pi)  
  = def_2_2 (v -> (imin = #imin pi | v))
Fifth version: more genericity?

Regularity in the knit functions:

\[
\text{knit}_i \ f \ \text{ch_fn} \\
= \ \pi \to \ \text{let} \ (\text{ch_arg}, \ po) = f \ (\text{kn}_i \ \text{ch_fn} \ \text{ch_arg}, \ pi) \\
\text{in} \ \ po
\]

\[
\text{kn}_i \ \sim (f_1, \ \ldots (f_n, \ ()), \ \sim (a_1, \ \ldots (a_n, \ ()), \ ) \\
= \ (f_1 \ a_1, \ \ldots (f_n \ a_n, \ ()), \ )
\]

Is it possible to write \text{kn}_i in Haskell?
class Knit a b c | a -> b c where
  kn :: a -> b -> c

instance Knit (a -> b) a b where
  kn f a = f a

instance Knit d e f => Knit (a -> b, d) (a, e) (b, f) where
  kn ~(a2b, d) ~(a, e) = (a2b a, kn d e)

knit f ch_fn
  = \ pi -> let (ch_arg, locals, po)
                = f (kn ch_fn ch_arg, locals, pi) new_Empty
        in po
And the consequences are:

-- redefinition of attribute definition combinators

syndef = \ f (x, l, p) -> (x, l, f p)
locdef = \ f (x, l, p) -> (x, f l, p)
def_1_1 = \ f (in1, l, p) -> (f in1, l, p)
def_2_1 = \ f ((in1, in2), l, p) -> ((f in1, in2), l, p)
def_2_2 = \ f ((in1, in2), l, p) -> ((in1, f in2), l, p)

-- tree walk

eval_Root (Root t)
  = knit (rootf ()) (eval_Tree t)
eval_Tree (Node left right)
  = knit (nodef ()) (eval_Tree left, eval_Tree right)
eval_Tree (Leaf i)
  = knit (leaff ()) (\EmptyRec -> i)
-- and the redefinition of attribute equations

rootf_imin_ci ~(co, loc, pi)
    = def_1_1 (\v -> (imin = #smin co | v))
nodef_imin_li ~((lo,ro), loc, pi)
    = def_2_1 (\v -> (imin = #imin pi | v))
nodef_imin_ri ~((lo,ro), loc, pi)
    = def_2_2 (\v -> (imin = #imin pi | v))

rootf_sres_po ~(co, loc, pi)
    = syndef (\v -> (sres = #sres co | v))
nodef_sres_po ~((lo,ro), loc, pi)
    = syndef (\v -> (sres = Node (#sres lo) (#sres ro) | v))
leaff_sres_po ~(i, loc, pi)
    = syndef (\v -> (sres = Leaf (#imin pi) | v))
class Knit a b c d | a -> b c d where --
   kn :: a -> b -> (c, d)

instance Knit (a -> b) a b (Rec EmptyRow) where
   kn f a = (f a, EmptyRec)

instance Knit d e f g
   => Knit (a -> b, d) (a, e) (b, f) (Rec EmptyRow, g) where
   kn ~(a2b, d) ~(a, e) = let (f, emptyrest) = kn d e
                         in  ((a2b a, f), (EmptyRec, emptyrest))

knit f ch_fn =
 \ pi -> let (ch_arg,loc_args, po) = f (ch_res, loc_args, pi )
                (empties, EmptyRec, EmptyRec)
             (ch_res, empties) = kn ch_fn ch_arg
         in  po
We should like to be able to “iterate” over common fields in records:

```
instance Apply r r' => Apply (?name::a -> blr) (?name:a l\ r) where
   apply (?name=f l \ r) (?name= a l \ r') = (?name = f a l apply r r')
```

This extension is currently being implemented in the completely attribute grammar based Utrecht Haskell Compiler.