Embedding Attribute Grammars with Functional Zippers

João Saraiva*

(Pedro Martins, JP Fernandes, Eric Van Wyk, Tony Sloane, Marcos Viera, Alberto Pardo)

Spring Festival Workshop on Programming Language, 2015
National Institute of Informatics (NII), Japan

*HASLab/INESC TEC and Department of Informatics
University of Minho
Portugal
http://www.di.uminho.pt/~jas
“Indeed, we are convinced that the style of programming with attribute grammars helps the programmer to construct better functional programs. Thus, the question that arises immediately is whether it would be possible to incorporate the elegant style of attribute grammar writing directly within a functional programming language. ... We hope that the continuing developments on functional languages and attribute grammars will make it possible to have functional programming languages supporting the attribute grammar style of programming.”

*Purely Functional Implementation of Attribute Grammars*

PhD Thesis - Conclusions, December 1999

João Saraiva
Goal of this talk:

To show how to write efficient, complex, multiple tree traversal algorithms as pure Haskell programs.

By, viewing/structuring them as attribute grammars!
Attribute Grammars

Why Attribute Grammars?

While, in the beginning, AG-based systems were used mainly to specify and derive efficient (batch) compilers for formal languages,

Nowadays, AG-based systems are powerful tools to construct:

- Programming Environments
- Haskell compilers (UU)
- Type Systems
- Complex Pretty Printing Algorithms, etc
Attribute Grammars

Why Attribute Grammars?

AGs are not only effectively used to specify such language-based tools, but also to express powerful (lazy) functional programs.

5
Attribute Grammars

Indeed, they are a convenient formalism to express (multiple traversal) algorithms:

1: The AG writer does not concern himself in breaking-up his (elegant) algorithm into traversal functions and to glue such functions.

2: They statically detect circularities: Indeed, termination of programs is statically guaranteed (for “pure” HAGs).

=> Semantic functions are not considered/analyzed by standard AG techniques...
But, in HAG inductive function can be replaced by higher-order attributes. Thus, semantic functions are redundant.

3: They provide a Component-based Style of Programming for AGs.

=> Components can be glued within the formalism

=> Induced circularities can be statically detected
The RepMin Problem (Bird 84): consider the problem of transforming a tree, for example $T_1$

$$T_1 = (\text{Fork (Tip 3)} \ (\text{Fork (Tip 2)} \ (\text{Tip 4})))$$

into a second tree, identical in shape to the original one, but with all the tip values replaced by the minimum tip value.

$$T_2 = (\text{Fork (Tip 2)} \ (\text{Fork (Tip 2)} \ (\text{Tip 2})))$$
The Straightforward Functional Solution:

\[
\begin{align*}
\text{data } \text{Prog} & \equiv \text{Root} \quad \text{Tree} \\
\text{data } \text{Tree} & \equiv \text{Fork} \quad \text{Tree} \quad \text{Tree} \\
& \quad | \quad \text{Tip} \quad \text{Int} \\
\text{tmin} & \equiv \text{Tree} \to \text{Int} \\
\text{tmin} (\text{Tip } n) & = n \\
\text{tmin} (\text{Fork } l \ r) & = \text{min} (\text{tmin } l) (\text{tmin } r) \\
\text{replace} & \equiv \text{Tree} \to \text{Int} \to \text{Tree} \\
\text{replace} (\text{Tip } n) \ mt & = \text{Tip } mt \\
\text{replace} (\text{Fork } l \ r) \ mt & = \text{Fork} (\text{replace } l \ mt) (\text{replace } r \ mt) \\
\text{transform} & \equiv \text{Prog} \to \text{Tree} \\
\text{transform} (\text{Root } t) & = \text{replace } t (\text{tmin } t)
\end{align*}
\]
Bird's repmin circular lazy program:

\[
\text{repmin} :: Tree \rightarrow \text{Int} \rightarrow (Tree, \text{Int}) \\
\text{repmin} (\text{Tip } n) \ m = (\text{Tip } m, n) \\
\text{repmin} (\text{Fork } l \ r) \ m = (\text{Fork } t1 \ t2, \text{min } m1 \ m2) \\
\text{where} \ (t1, m1) = \text{repmin } l \ m \\
(t2, m2) = \text{repmin } r \ m
\]

\[
\text{transform} :: Prog \rightarrow \text{Tree} \\
\text{transform} (\text{Root } t) = nt \\
\text{where} \ (nt, m) = \text{repmin } t \ m
\]

A single function call to \textit{repmin}: a single traversal.
Strict versus Lazy Solution

- **Strict:**
  - A Modular approach...
  - Efficient
  - *but, no modularity!* (later in the talk)

- **Lazy**
  - No scheduling of traversals (single traversal)
  - No gluing data structures (single traversal)
  - *but, no modularity!*


**The Straightforward Catamorphic Solution:**

Probably, Zhenjiang Hu and Jeremy Gibbons prefer to express repmin as follows:

\[
\begin{align*}
\text{tmin} & \quad = \quad \text{foldBTree} \ (id, \text{min}) \\
\text{replace} \ m & \quad = \quad \text{foldBTree} \ (\ x \rightarrow \ \text{Tip} \ m, \text{Fork})
\end{align*}
\]

\[
\begin{align*}
\text{transform} & \quad :: \quad \text{Prog} \rightarrow \text{Tree} \\
\text{transform} \ (\text{Root} \ t) & \quad = \quad \text{replace} \ t \ (\text{tmin} \ t)
\end{align*}
\]

or, as a circular fold/unfold.

(but, with the same modularity problems...)

Gluing Data Structures

Multiple tree traversal programs use intermediate data structures to glue traversals!

\textit{repmin}: the original tree is the “gluing” data structure.

Actually, the gluing tree is “\textit{smaller}” than the original tree.

We can make it explicit...
RepMin with an Intermediate Gluing Structure:

\[
\begin{align*}
\text{data } STree & = \text{SFork } STree \ STree \\
& \quad | \quad \text{STip}
\end{align*}
\]

\[
\begin{align*}
tminst & : : Tree \to (STree, \text{Int}) \\
tminst (\text{Tip } n) & = (\text{STip}, n) \\
tminst (\text{Fork } l \ r) & = (\text{SFork } nl \ nr, \text{min } ml \ mr) \\
\text{where } (nl, ml) & = tminst l \\
(nr, mr) & = tminst r
\end{align*}
\]

\[
\begin{align*}
\text{transform} & : : Tree \to Tree \\
\text{transform} \ \text{tree} & = \text{streplace } t \ m \\
\text{where } (t, m) & = tminst \ \text{tree}
\end{align*}
\]

\[
\begin{align*}
\text{streplace} & : : STree \to \text{Int} \to Tree \\
\text{streplace } (\text{STip}) \ m & = \text{Tip } m \\
\text{streplace } (\text{SFork } l \ r) \ m & = \text{Fork } (\text{streplace } l \ m) \ (\text{streplace } r \ m)
\end{align*}
\]
Repmin: The Data Flow:

These dependencies force additional tree traversals!
Repmin: The Data Flow:

Computing the minimum:

Fork:

Tip:

Tree
Repmin: The Data Flow:

Passing down the minimum and constructing the new tree:
Brief Introduction to AGs: The RepMin AG

The Context-Free Grammar:

\[
\begin{align*}
    \text{Root} & : \text{Prog} \rightarrow \text{Tree} \\
    \text{Tip} & : \text{Tree} \rightarrow \text{Int} \\
    \text{Fork} & : \text{Tree} \rightarrow \text{Tree} \text{ Tree}
\end{align*}
\]

The attributes:

- \textit{m}: synthesizes the minimum value of the original tree
- \textit{mt}: distributes downwards the minimum value (\textit{i.e.,} passes context information down in the tree).
- \textit{nt}: synthesizes the new tree
The RepMin Attribute Grammar

To capture the term-like nature of CFGs we prefer the following notation for (abstract) grammars:

\[
\begin{align*}
Prog & = \text{Root} \quad Tree \\
Tree & = \text{Tip} \quad Int \\
& \quad | \quad \text{Fork} \quad Tree \quad Tree
\end{align*}
\]

Roughly speaking, non-terminal symbols define tree type constructors and productions define value type constructors. We will return to this subject later on.
The RepMin Attribute Grammar

Attribute grammars are modular: They can be decomposed into different fragments (or aspects) describing a particular semantic domain of the language.

Fragment 1: Computing the minimum.
The RepMin Attribute Grammar

Fragment 2: Passing down the min and constructing the new tree.
The RepMin Attribute Grammar

*Fragment 3:* The root production.

\[
\text{Prog} \quad <\uparrow nt : \text{Tree} > \\
\text{Prog} = \text{Root \ Tree} \\
\text{Tree.mt} = \text{Tree.m} \\
\text{Prog.nt} = \text{Tree.nt}
\]

These dependencies force additional tree traversals!
Attribute Grammars & Functional Programming

Strictification of circular programs
JP Fernandes, J Saraiva, D Seidel, J Voigtländer
PEPM 2011

A shortcut fusion rule for circular program calculation
JP Fernandes, A Pardo, J Saraiva
Haskell 2007
Gluing Data Structures

Multiple tree traversal programs use intermediate data structures to glue traversals!

repmin: needs a smaller gluing data structure.

Let us consider a more realistic example: the gluing tree is “bigger” than the original one.
The Block Language

- The concrete Syntax:

  \[
  [ \text{use } x \text{ ; use } y \text{ ; decl } x \text{ ;}
  
  [ \text{decl } y \text{ ; use } y \text{ ; use } w ] \text{ ;}
  
  \text{decl } y \text{ ; decl } x
  
  ]
  \]

- The semantic rules:

  - Use of declared names;
  - Use before declaration;
  - No duplicated declarations inside a block;
  - We want to produce a list of errors which follows the sequential structure of the program.
The Block Language

- The concrete Syntax:

  [ use x ; use y ; decl x ;
    [ decl y ; use y ; use w ] ;
    decl y ; decl x
  ]

- The semantic rules:

  - Use of declared names;
  - Use before declaration;
  - No duplicated declarations inside a block;
  - We want to produce a list of errors which follows the sequential structure of the program: $[w,x]$. 
The Block Language

Because Block does not force a declare-before-use discipline, this naturally leads to the following algorithm:

- First, we collect the declarations

- After that, we are in a position to detect invalid uses of variables.
The Block Language

The algorithm:

1st Traversal
- Collect the list of definitions
- Detect duplicated definitions
(usage the collected definitions)

2nd Traversal
- Use the list of definitions as the global environment
- Detect use of non defined names
- Combine “both” errors

The straightforward implementation of this algorithm is not so simple.

Intermediate values (the errors) have to be passed from the first to the second traversal.
Block attribute grammar

The Declaration Aspect: In order to detect invalid uses of identifiers we have to construct the environment:

\[ \text{env} = [x, y] \]

\[ [\text{use } y ; \text{decl } x ; \text{decl } x ; \text{decl } y ] \]

Thus, we have two possibilities:
The Block Language

Inner blocks inherit the environment of their outer ones.

\[
\begin{align*}
dcli &= [] \\
decl &= \{(x,0),(y,0)\} \\
\text{sentence} &= \begin{array}{c}
\text{use } x \text{ ; use } y \text{ ; decl } x \text{ ;} \\
dcli &= \{(x,0),(y,0)\} \\
decl &= \{(y,1),(x,0),(y,0)\} \\
&\quad \begin{array}{c}
\text{decl } y \text{ ; use } y \text{ ; use } w \text{ ;} \\
\text{decl } y \text{ ; decl } x
\end{array}
\end{array}
\end{align*}
\]

To distinguish declarations at different levels we associate the level to the declared identifier.
This simple algorithm is difficult to implement within the imperative or functional paradigm:

\[
\text{declarations} :: Block \rightarrow Env \rightarrow Env \\
\text{invalidUses} :: Block \rightarrow Env \rightarrow Errors
\]

But, we compute duplicated declarations while building the environment...
This simple algorithm is difficult to implement within the imperative or functional paradigm:

\[
\text{declarations} :: \text{Block} \rightarrow \text{Env} \rightarrow (\text{Env}, \text{Errors})
\]

\[
\text{invalidUses} :: \text{Block} \rightarrow \text{Env} \rightarrow \text{Errors} \rightarrow \text{Errors}
\]

Hum... the errors will not follow the sequential structure of the program....
Inter Traversal Dependencies

In this simple example there are intermediate values that have to be passed from one traversal to following ones.
Intermediate value (corresponding to errors detected) have to be passed from the first to the second traversal.

- Thus gluing data types have to be defined.

\[
declarations :: \ Block \rightarrow \ Env \rightarrow (Block_2, \ Env)
\]

\[
invalidUses :: Block_2 \rightarrow \ Env \rightarrow Errors
\]

(in an imperative setting such values are stored in the original tree as side effects)
The Block (Strict) Program

Block Data Types:

\[
\begin{align*}
\text{data } \text{Prog} &= \text{RootP} \ \text{Its} \\
\text{data } \text{Its} &= \text{NilIts} \\
&| \quad \text{ConsIts IIts} \\
\text{data } \text{It} &= \text{Use} \ \text{String} \\
&| \quad \text{Decl} \ \text{String} \\
&| \quad \text{Block} \ \text{Its}
\end{align*}
\]

Gluing Data Types:

\[
\begin{align*}
\text{data } \text{Its}^2 &= \text{NilIts}^2 \\
&| \quad \text{ConsIts}^2 \ \text{It}^2 \ \text{Its}^2 \\
\text{data } \text{It}^2 &= \text{Use}^2 \ \text{String} \\
&| \quad \text{Decl}^2 \ \text{Errors} \quad -- \text{duplicated names :: } [\text{String}] \\
&| \quad \text{Block}^2 \ \text{Int} \ \text{Its} \quad -- \text{level, original inner block}
\end{align*}
\]
The Block (Strict) Program

The Block Evaluator

\[
\text{visit} \text{Prog} (\text{RootP it}) = \text{errors}
\]
where \((\text{its}^2, \text{dclo}_2) = \text{declsIts} [] 0\)
\[
\text{errors} = \text{usesIts it}^2 \text{ dclo}_2
\]

\[
\text{declsIt} (\text{Block it}) \text{ dcli lev} = (\text{Block}^2 \text{ lev}_2 \text{ it}, \text{dclo})
\]
where \(\text{lev}_2 = \text{lev} + 1\)

\[
\text{usesIt} (\text{Block}^2 \text{ lev it}) \text{ env} = \text{errors}_2
\]
where \((\text{its}^2, \text{dclo}_2) = \text{declsIts it env lev}\)
\[
\text{errors}_2 = \text{usesIts it}^2 \text{ dclo}_2
\]

\(\text{lev}\) is computed in first visit and used in the second one
We should **not** write such programs!

- Moreover, it is difficult to schedule the traversals only in the second traversal of the outer block, we visit inner blocks for the first time!

**Solution:** Circular Lazy Programs...

**Hum... no modularity!**
Solution: To express these algorithms as attribute grammars directly in Haskell.

What we need: A mechanism to traverse trees

Functional Zippers!
Functional Zipper

- Invented by Grard Huet in 1997

- Aggregate Data Structure

- Convenient for writing programs that traverse the structure arbitrarily
Navigating on trees with Zippers
Navigating on trees with Zippers

\[
\text{let } z = \text{toZipper } \text{tree}
\]
Navigating on trees with Zippers

let z = toZipper tree
let dz = z.$2
Navigating on trees with Zippers

let z = toZipper tree
let dz = z.$2
let ddz = dz.$1
Navigating on trees with Zippers

let z = toZipper tree
let dz = z.$2
let ddz = dz.$1
let uddz = parent ddz
Zipper-based Repmin

Computing the minimum:

$$m :: \text{Zipper Root} \rightarrow \text{Int}$$

$$m \text{ tree} = \text{case constructor tree of}$$

"Tip" $\rightarrow$ lexeme tree

"Fork" $\rightarrow$ min (m (tree.$1$)) (m (tree.$2$))
Zipper-based Repmin

Passing down the minimum:

\[ \text{mt} :: \text{Zipper Root} \to \text{Int} \]
\[ \text{mt tree} = \text{case constructor t of} \]
\[ "\text{Root}" \to m \text{ tree.$1} \]
\[ "\text{Tip}" \to \text{mt (parent tree)} \]
\[ "\text{Fork}" \to \text{mt (parent tree)} \]
Zipper-based Repmin

Constructing the new tree:

\[
nt : \text{ Zipper Root} \rightarrow \text{ Tree}
\]
\[
nt \text{ tree } = \text{ case constructor tree of}
\]
  
  "Root" \rightarrow nt (tree.$1)

  "Tip" \rightarrow Tip (mt tree)

  "Fork" \rightarrow Fork (nt (tree.$1)) (nt (tree.$2))
Zipper-based Repmin

transform :: Root -> Tree
transform tree = nt (toZipper tree)
Zipper-based Block

The abstract structure of Block language can be described by the following context-free grammar:

\[
P = R \quad Its
\]

\[
Its = ConsIts \quad It \quad Its \\
| \quad NilIts
\]

\[
It = Decl \quad Name \\
| \quad Use \quad Name \\
| \quad Block \quad Its
\]
Block attribute grammar

The attributes:

- \textit{dcli,dclo}: Attributes that define the accumulation of declarations.

- \textit{lev}: Attribute that assigns the level to a declaration.

- \textit{env}: Attribute that defines the total environment of each block.

- \textit{errors}: Attribute that defines the list of errors.
Accumulating the list of definitions:
Accumulating the list of definitions:

\[
dclo \text{ tree} = \text{case (constructor tree) of}
\]
\[
"\text{ConsIts}" \rightarrow \text{dclo (tree.$2)}
\]
\[
"\text{NilIts}" \rightarrow \text{dcli tree}
\]
\[
"\text{Use}\" \rightarrow \text{dcli tree}
\]
\[
"\text{Decl}\" \rightarrow (\text{lexeme tree,level tree}) : (\text{dcli tree})
\]
\[
"\text{Block}\" \rightarrow \text{dcli tree}
\]
Accumulating the list of definitions:

**Inherited attributes:** Different productions have different rules. To know the context of a production, we need to navigate to the parent first.

dcli tree = case (constructor tree) of
  "NilIts" -> case (constructor (parent tree)) of
    "ConsIts" -> dclo ((parent tree).$1)
    "Block"   -> env (parent tree)
    "Root"    -> []
Accumulating the list of definitions:

The complete definition of \textit{dcli}.

dcli \text{tree} = \text{case (constructor tree) of}
    \begin{align*}
    \textquote{NilIts} & \rightarrow \text{case (constructor (parent tree)) of} \\
    \textquote{ConsIts} & \rightarrow \text{dclo ((parent tree).$1)} \\
    \textquote{Block} & \rightarrow \text{env (parent tree)} \\
    \textquote{Root} & \rightarrow [] \\
    \textquote{ConsIts} & \rightarrow \text{case (constructor (parent tree)) of} \\
    \textquote{ConsIts} & \rightarrow \text{dclo ((parent tree).$1)} \\
    \textquote{Block} & \rightarrow \text{env (parent tree)} \\
    \textquote{Root} & \rightarrow [] \\
    \textquote{Block} & \rightarrow \text{dcli (parent tree)} \\
    \textquote{Use} & \rightarrow \text{dcli (parent tree)} \\
    \textquote{Decl} & \rightarrow \text{dcli (parent tree)} \\
    \textquote{Root} & \rightarrow []
    \end{align*}
Distributing the environment:

**Root:** $P$

**NilIts:** $IIts$

**ConsIts:**

**Decl:** $Name$

**Use:** $Name$

**Block:** $IIts$
Distributing the environment:

\[
\text{env } z = \text{case } (\text{constructor } z) \text{ of } \\
\quad "\text{NilIts}" \rightarrow \text{case } (\text{constructor } (\text{parent } z)) \text{ of } \\
\qquad "\text{Block}" \rightarrow \text{dclo } z \\
\qquad "\text{ConsIts}" \rightarrow \text{env } (\text{parent } z) \\
\qquad "\text{Block}" \rightarrow \text{dclo } z \\
\quad "\text{ConsIts}" \rightarrow \text{case } (\text{constructor } (\text{parent } z)) \text{ of } \\
\qquad "\text{Block}" \rightarrow \text{dclo } z \\
\qquad "\text{ConsIts}" \rightarrow \text{env } (\text{parent } z) \\
\qquad "\text{Root}" \rightarrow \text{dclo } z \\
\quad "\text{Block}" \rightarrow \text{env } (\text{parent } z) \\
\quad "\text{Use}" \rightarrow \text{env } (\text{parent } z) \\
\quad "\text{Decl}" \rightarrow \text{env } (\text{parent } z) \\
\quad "\text{Root}" \rightarrow \text{dclo } z
\]

55
Computing the Level:

level :: Zipper Root -> Int
level z = case (constructor z) of
  "Root"    -> 0
  "NilIts"  -> case (constructor $ parent z) of
    "Block"  -> (level (parent z)) + 1
    "ConsIts"  -> level (parent z)
    "Root"    -> 0
  "ConsIts" -> case (constructor (parent z)) of
    "Block"  -> (level (parent z)) + 1
    "ConsIts" -> level (parent z)
    "Root"    -> 0
  "Block"   -> level (parent z)
  "Use"     -> level (parent z)
  "Decl"    -> level (parent z)
Computing the list of errors:

\[ \text{Root: } P \quad \text{NILIts: } Its \quad \text{CONSIts: } Its \]

\[ \text{DECL: lev dcli error} \quad \text{USE: env Its errors} \quad \text{BLOCK: Its} \]

\( \text{mNBin, mBin: simple lookups on lists} \)

57
Computing the list of errors:

\[
\text{errs tree} = \text{case (constructor tree) of}
\]

- "Root" → \text{errs (tree.$1)}
- "NilIts" → []
- "ConsIts" → \((\text{errs (tree.$1)}) ++ \text{errs (tree.$2)})\)
- "Use" → \text{mBIn (lexeme tree) (env tree)}
- "Decl" → \text{mNBI}n (lexeme tree,lev tree) (dcli tree)
- "Block" → \text{errs (tree.$1)}
Zipper-based Block

We have shown the complete Haskell program!

\[
\text{semantics} :: P \rightarrow [\text{String}]
\]
\[
\text{semantics} \ p = \text{errs} \ (\text{toZipper} \ p)
\]

And, we can execute it with our example program:

\[
p = \begin{array}{c}
\ [ \ \text{use} \ x \ ; \ \text{use} \ y \ ; \ \text{decl} \ x \ ; \\
[ \ \text{decl} \ y \ ; \ \text{use} \ y \ ; \ \text{use} \ w \ ] \\
\text{decl} \ y \ ; \ \text{decl} \ x
\end{array}
\]

\[
\text{semantics} \ p = ["w","x"]
\]
Zipper-based Embedding of Attribute Grammars

We just presented a pure Haskell program that:

- is modular;
- does not use gluing data structures;
- where we did not schedule traversals;
- was implemented in a simple mechanism and library; (Data.Generics.Zippers + 50 lines of our code!)
- that does not make essential use of lazy evaluation (nor, on advanced Haskell mechanisms);
- and that can model all modern AG extensions! (paper submitted to SCP)
Embedded Attribute Grammars

Vieira and Swierstra\textsuperscript{2} (ICFP’09) provide a proper embedding of AGs:

\textbf{Attribute Grammars Fly First-Class}

\textit{How to do Aspect Oriented Programming in Haskell}

\begin{tabular}{lll}
Marcos Viera & S. Doaitse Swierstra & Wouter Swierstra \\
Instituto de Computación  & Department of Computer Science  & Chalmers University of Technology \\
Universidad de la República  & Utrecht University  & Göteborg, Sweden \\
Montevideo, Uruguay  & Utrecht, The Netherlands  & wouter@chalmers.se \\
mviera@fing.edu.uy  & doaitse@cs.uu.nl
\end{tabular}

using \textbf{all} advanced mechanisms of Haskell, and relying on a \textbf{circular lazy program} to execute the AG.
But, zippers do *not* provide a proper embedding of AGs!

Emails with Doaitse Swierstra

Subject: AGs Fly Executive Class!

Attach:

Zipper-based Attribute Grammars and their Extensions*

Pedro Martins¹, João Paulo Fernandes¹,², and João Saraiva¹

¹ High-Assurance Software Laboratory (HASLAB/INESC TEC), Universidade do Minho, Portugal
But, zippers do *not* provide a proper embedding of AGs!

Emails with Doaitse Swierstra: One hour later!

Subject: RE: AGs Fly Executive Class!

The title of the email is nice, but this is not a proper embedding of AGs: you are recalculating attribute values.
But, zippers do *not* provide a proper embedding of AGs!

Doaitse was quick and correct...

\[
\text{nt tree = case constructor tree of } \\
\quad \text{"Root" -> nt (tree.$1)} \\
\quad \text{"Tip" -> Tip (mt tree) -- ???} \\
\quad \text{"Fork" -> Fork (nt (tree.$1)) (nt (tree.$2))}
\]

Repmin profiling:

(Hum... PhD supervisors are always right :-)
But, zippers do *not* provide a proper embedding of AGs!

Emails with Doaitse Swierstra: (10 min later!)

Subject: RE: AGs Fly Executive Class!

You are right! But, that is an optimization issue that we can solve using memoization.
Memoized Zippers

Pedro Martins visited Vieira and Pardo on October 2014

- we memoize attribute values in the tree to avoid attribute recalculation.
Memoized Repmin

We mimic the imperative approach: computed attribute values are stored in a memo tree.

data MemoTree = MemoRoot MemoTree MemoTable
  | MemoFork MemoTree MemoTree MemoTable
  | MemoLeaf Int MemoTable
  deriving (Typeable, Data)

type MemoTable = [(String, Dynamic)]

We need functions to manipulate memo tables:

lookupAttr :: Typeable res => String -> MemoTable -> Maybe res
memo :: Typeable res => String -> MemoAG res -> Dir -> MemoAG res
Because, we need to memoize attribute values in the tree while traversing it: the tree is in the state monad.

type MemoAG = State (Zipper MemoTree)

And, we write our AGs in a monadic style (again, very similar to the style of AG programming):

locmin = memo "Locmin" $ do
    constr <- constructor
    case constr of
        "Root" -> locmin (Child 1)
        "Leaf" -> do ag <- get
                   return $ lexeme_Leaf ag
        "Fork" -> do left <- locmin (Child 1)
                   right <- locmin (Child 2)
                   return $ min left right
globmin = memo "Globmin" $ do
    constr <- constructor
    case constr of
        "Root" -> locmin Local
        "Leaf" -> globmin Parent
        "Fork" -> globmin Parent

replace = memo "Replace" $ do
    constr <- constructor
    case constr of
        "Root" -> replace (Child 1)
        "Leaf" -> do mini <- globmin Local
                   return $ Leaf mini
        "Fork" -> do left <- replace (Child 1)
right <- replace (Child 2)
return $ Fork left right

semantics :: Tree -> Tree
semantics t = let ag = toZipper $ buildMemoTree t
            in fst $ runState (replace Local) ag
Memoized Zippers: Benchmarking
Memoized Attribute Grammars:

Efficient Zipper-based Tree Traversals.

(to be submitted to ICFP’15)

Functional Pearl: Memoized Attribute Grammars

Efficient zipper-based tree traversals

Pedro Martins
HASLab/INESC TEC and Univ. do Minho, Portugal
prmartins@di.uminho.pt

Marcos Viera and Alberto Pardo
Universidad de la Republica, Uruguay
{pardo, mviera}@ping.edu.uy

João Fernandes and João Saraiva
HASLab/INESC TEC and Univ. do Minho & ESTGF, Portugal
{jpaulo, jas}@di.uminho.pt

(it is about time to send another email...
Memoized Zipper-based AGs: Energy Efficiency?

We have developed a bridge from Haskell to intel RAPL library.

**Zipper-based Repmin:** RepMin (sum): 1952

- Package energy after: 21.442520J consumed
- on-core GPU (if avail) after: 0.277084J consumed

**Memoized zipper-based Repmin:** RepMin (sum): 1952

- Package energy after: 0.289246J consumed
- on-core GPU (if avail) after: 0.007507J consumed
Attribute Grammar Extensions

Moreover, zipper-based attribute grammars support all modern extensions, namely:

- Higher order Attribute Grammars (tree is not fixed during evaluation)
- Reference Attribute Grammar (graph algorithms)
- Circular Attribute Grammars (fix point computations)
- Bidirectional Attribute Grammars (Pedro Martins’ talk)
Conclusions

“Indeed, we are convinced that the style of programming with attribute grammars helps the programmer to construct better functional programs. Thus, the question that arises immediately is whether it would be possible to incorporate the elegant style of attribute grammar writing directly within a functional programming language. ... We hope that the continuing developments on functional languages and attribute grammars will make it possible to have functional programming languages supporting the attribute grammar style of programming.”

• We did answer the question! Thus, there is hope...
Embedding Attribute Grammars with Functional Zippers

João Saraiva*

(Pedro Martins, JP Fernandes, Eric Van Wyk, Tony Sloane, Marcos Viera, Alberto Pardo)

Spring Festival Workshop on Programming Language, 2015
National Institute of Informatics (NII), Japan

*HASLab/INESC TEC and Department of Informatics
University of Minho
Portugal
http://www.di.uminho.pt/~jas