Generalising Tree Traversals to DAGs

Exploiting Sharing without the Pain

Patrick Bahr\textsuperscript{1} Emil Axelsson\textsuperscript{2}

\textsuperscript{1}University of Copenhagen
 paba@diku.dk

\textsuperscript{2}Chalmers University of Technology
 emax@chalmers.se

PEPM 2015
Motivation

Goal
Do stuff on acyclic graphs, but pretend they are only trees.
Motivation

Goal
Do stuff on acyclic graphs, but pretend they are only trees.

Primary Application
Abstract Syntax Graphs/Trees:
- type inference
- program analyses
- program transformations
- ...
The Idea

Γ ⊩ ρ : τ

unravels to

Attribute Grammar

Why?
▶ It's more difficult to get a traversal on graphs right.
▶ But: it's more efficient to traverse the graph.
The Idea

$\Gamma \vdash \rho : \tau$

unravels to Attribute Grammar

Why?

▶ It's more difficult to get a traversal on graphs right.
▶ But: it's more efficient to traverse the graph.
The Idea

Γ ⊨ ρ : τ
The Idea

\[ \Gamma \vdash \rho : \tau \]

- Why?
  - It's more difficult to get a traversal on graphs right.
  - But: it's more efficient to traverse the graph.
The Idea

\[ \Gamma \vdash \rho : \tau \]

Why?

- It's more difficult to get a traversal on graphs right.
- But: it's more efficient to traverse the graph.

Result
The Idea

\[ \Gamma \vdash p : \tau \]

unravels to

Attribute Grammar

Why?

▶ It's more difficult to get a traversal on graphs right.
▶ But: it's more efficient to traverse the graph.
The Idea

\[ \Gamma \vdash \rho : \tau \]

unravels to

Why?

▶ It's more difficult to get a traversal on graphs right.
▶ But: it's more efficient to traverse the graph.
The Idea

\[ \Gamma \vdash p : \tau \]

Attribute Grammar

unravels to

Result

Why?

- It's more difficult to get a traversal on graphs right.
- But: it's more efficient to traverse the graph.
The Idea

Attribute Grammar

Result

unravels to

Why?

▶ It’s more difficult to get a traversal on graphs right.
▶ But: it’s more efficient to traverse the graph.
What’s the Catch?

It doesn’t work in general.

But: it does work for many cases.

Our Contribution

▶ Identify classes of AGs for which this approach works.
▶ Prototype implementation in Haskell.
▶ Case studies and benchmarks.
What’s the Catch?

It doesn’t work
What’s the Catch?

It doesn’t work in general.
What’s the Catch?

It doesn’t work in general.

But: it does work for many cases.
What’s the Catch?

It doesn’t work in general.

**But:** it does work for many cases.

Our Contribution

- Identify classes of AGs for which this approach works.
- Prototype implementation in Haskell.
- Case studies and benchmarks.
A Toy Example

**data** IntTree = Leaf Int
               □ Node IntTree IntTree

leavesBelow :: Int → IntTree → Set Int

leavesBelow d (Leaf i) =
                       □ d ≤ 0 = Set.singleton i
                       □ otherwise = Set.empty

leavesBelow d (Node t₁ t₂) =
                            leavesBelow (d − 1) t₁
                            ∪ leavesBelow (d − 1) t₂
A Toy Example

\textbf{data} \textit{IntTree} = \textit{Leaf} \textit{Int} \\
\quad \mid \textit{Node} \textit{IntTree} \textit{IntTree}

\textit{leavesBelow} :: \textit{Int} \to \textit{IntTree} \to \textit{Set} \textit{Int}

\textit{leavesBelow} \ d \ (\textit{Leaf} \ i) \\
\quad \mid \ d \leq 0 \quad = \textit{Set} \textit{.singleton} \ i \\
\quad \mid \text{otherwise} \quad = \textit{Set} \textit{.empty}

\textit{leavesBelow} \ d \ (\textit{Node} \ t_1 \ t_2) = \\
\quad \textit{leavesBelow} \ (d - 1) \ t_1 \\
\quad \cup \textit{leavesBelow} \ (d - 1) \ t_2
A Toy Example

\textbf{data} \textit{IntTree} = \textit{Leaf} \textit{Int}
\hspace{1em} | \hspace{1em} \textit{Node} \textit{IntTree} \textit{IntTree}

\textit{leavesBelow} :: \textit{Int} \to \textit{IntTree} \to \textit{Set} \textit{Int}
\textit{leavesBelow} \ d \ (\textit{Leaf} \ i)
\hspace{1em} | \hspace{1em} d \leq 0 \ = \ \textit{Set} \cdot \textit{singleton} \ i
\hspace{1em} | \hspace{1em} \text{otherwise} \ = \ \textit{Set} \cdot \textit{empty}
\textit{leavesBelow} \ d \ (\textit{Node} \ t_1 \ t_2) =
\hspace{1em} \textit{leavesBelow} \ (d - 1) \ t_1
\hspace{1em} \cup \hspace{1em} \textit{leavesBelow} \ (d - 1) \ t_2
Traversals on Graphs

For which traversals is this correct?
Traversals on Graphs

For which traversals is this correct?
For which traversals is this correct?
For which traversals is this correct?
Traversals on Graphs

For which traversals is this correct?
But before that, let’s implement it!

\[
\textbf{data } \text{IntTree} \quad = \quad \text{Leaf } \text{Int} \\
\quad \mid \quad \text{Node } \text{IntTree } \text{IntTree}
\]
But before that, let’s implement it!

```haskell
data IntTree a = Leaf Int
               | Node a a
```

leavesBelow :: Int → Tree IntTreeF → Set Int
leavesBelow = runAG leavesBelow

leavesBelowG :: Int → Dag IntTreeF → Set Int
leavesBelowG = runAGDag min leavesBelow

⊕ 7
But before that, let’s implement it!

```
data IntTreeF a = Leaf Int
              | Node a a
```
But before that, let’s implement it!

```haskell
data IntTreeF a = Leaf Int
  | Node a a

leavesBelow :: Int → Tree IntTreeF → Set Int
leavesBelow = runAG leavesBelowₕ leavesBelow₁
```
But before that, let’s implement it!

```haskell
data IntTreeF a = Leaf Int
                | Node a a

leavesBelow :: Int → Tree IntTreeF → Set Int
leavesBelow = runAG leavesBelowS leavesBelow₁

leavesBelow_G :: Int → Dag IntTreeF → Set Int
leavesBelow_G = runAGDag min leavesBelowS leavesBelow₁
```

⊕
Implementing the semantic functions

\[ \text{leavesBelow}_I :: \text{Inh IntTreeF atts Int} \]
\[ \text{leavesBelow}_I (\text{Leaf } i) = \emptyset \]
\[ \text{leavesBelow}_I (\text{Node } t_1 t_2) = t_1 \mapsto d \& t_2 \mapsto d \]
\[ \text{where } d = \text{above} - 1 \]
Implementing the semantic functions

\[
\text{leavesBelow}_I :: \text{Inh IntTreeF atts Int}
\]
\[
\text{leavesBelow}_I (\text{Leaf } i) = \emptyset
\]
\[
\text{leavesBelow}_I (\text{Node } t_1 \, t_2) = t_1 \mapsto d \& t_2 \mapsto d
\]
  \text{where } d = \text{above} - 1

\[
\text{leavesBelow}_S :: (\text{Int} \in \text{atts}) \Rightarrow \text{Syn IntTreeF atts (Set Int)}
\]
\[
\text{leavesBelow}_S (\text{Leaf } i)
\]
  \mid (\text{above} :: \text{Int}) \leq 0 = \text{Set.singleton } i
\]
  \mid \text{otherwise} = \text{Set.empty}
\[
\text{leavesBelow}_S (\text{Node } t_1 \, t_2) = \text{below } t_1 \cup \text{below } t_2
\]
Correctness

unravels to

Result
Correctness

unravels to

Result

Attribute Grammar
Correctness

unravels to

Result

Attribute Grammar & ⊕
Correctness

unravels to

the same
Attribute Grammar

merge operator

Result

Attribute Grammar
Correspondence Theorems

Theorem (Monotone AGs)

Let

1. $G$ be a non-circular AG,
2. $\oplus$ an assoc., comm. operator on inherited attributes, and
3. $\preceq$ such that $G$ is monotone and $\oplus$ is decreasing w.r.t. $\preceq$.

If $(G, \oplus)$ terminates on a DAG $g$ with result $r$,
then $G$ terminates on $\mathcal{U}(g)$ with result $r'$ such that $r \preceq r'$. 
Correspondence Theorems

Theorem (Monotone AGs)

Let

1. \( G \) be a non-circular AG,
2. \( \oplus \) an assoc., comm. operator on inherited attributes, and
3. \( \preceq \) such that \( G \) is monotone and \( \oplus \) is decreasing w.r.t. \( \preceq \).

If \((G, \oplus)\) terminates on a DAG \( g \) with result \( r \), then \( G \) terminates on \( \mathcal{U}(g) \) with result \( r' \) such that \( r \preceq r' \).
Correspondence Theorems

Theorem (Monotone AGs)

Let

(1) $G$ be a non-circular AG,
(2) $\oplus$ an assoc., comm. operator on inherited attributes, and
(3) $\preceq$ such that $G$ is monotone and $\oplus$ is decreasing w.r.t. $\preceq$.

If $(G, \oplus)$ terminates on a DAG $g$ with result $r$, then $G$ terminates on $U(g)$ with result $r'$ such that $r \preceq r'$.

Example

For the $\text{leavesBelow}$ AG, define $\preceq$ as follows:

- on $\text{Int}$: $x \preceq y \iff x \leq y$
- on $\text{Set Int}$: $S \preceq T \iff S \supseteq T$
Correspondence Theorems

Theorem (Monotone AGs)

Let

1. $G$ be a non-circular AG,
2. $\oplus$ an assoc., comm. operator on inherited attributes, and
3. $\preceq$ such that $G$ is monotone and $\oplus$ is decreasing w.r.t. $\preceq$.

If $(G, \oplus)$ terminates on a DAG $g$ with result $r$, then $G$ terminates on $U(g)$ with result $r'$ such that $r \preceq r'$.

Example

For the $\text{leavesBelow}$ AG, define $\preceq$ as follows:

- on $\text{Int}$: $x \preceq y \iff x \leq y$
- on $\text{Set Int}$: $S \preceq T \iff S \supseteq T$

$\implies \text{leavesBelow}_G d g \supseteq \text{leavesBelow}_d (U(g))$
Correspondence Theorems

Theorem (Monotone AGs)

Let

(1) \( G \) be a non-circular AG,
(2) \( \oplus \) an assoc., comm. operator on inherited attributes, and
(3) \( \preceq \) such that \( G \) is monotone and \( \oplus \) is decreasing w.r.t. \( \preceq \).

If \( (G, \oplus) \) terminates on a DAG \( g \) with result \( r \),
then \( G \) terminates on \( \mathcal{U}(g) \) with result \( r' \) such that \( r \preceq r' \).

Example

For the \texttt{leavesBelow} AG, define \( \preceq \) as follows:

- on \texttt{Int}: \( x \preceq y \iff x \leq y \)
- on \texttt{Set Int}: \( S \preceq T \iff S \supseteq T \)

\[ \implies \text{leavesBelow}_G d \ g \supseteq \text{leavesBelow}_d (\mathcal{U}(g)) \]
for \( \text{leavesBelow}_G d \ g \subseteq \text{leavesBelow}_d (\mathcal{U}(g)) \) see paper
Termination

- We know: non-circular AGs terminate on any tree.
- But: non-circular AGs may diverge on DAGs.
Termination

- We know: non-circular AGs terminate on any tree.
- But: non-circular AGs may diverge on DAGs.

Example

\[
\begin{align*}
\text{Example} & \\
\text{Termination} & \\
\text{We know: non-circular AGs terminate on any tree.} & \\
\text{But: non-circular AGs may diverge on DAGs.} & \\
\end{align*}
\]
Termination

- We know: non-circular AGs terminate on any tree.
- But: non-circular AGs may diverge on DAGs.

Example

Theorem (termination)

Let $G$, $\oplus$, and $\lesssim$ be as before.

If $\lesssim$ is well-founded on inherited attributes, then $(G, \oplus)$ terminates on any DAG.
Correspondence Theorem for Copying AGs

Copying AGs

- inherited attributes are just propagated, not changed
- Example: Bird’s repmin problem.
Correspondence Theorem for Copying AGs

Copying AGs

- inherited attributes are just propagated, not changed
- Example: Bird’s repmin problem.

Theorem (copying AGs)

Let

1. \(G\) be a copying, non-circular AG, and
2. \(x \oplus y \in \{x, y\}\) for all \(x, y\).

Then

(i) \((G, \oplus)\) terminates on any DAG, and
(ii) \(\|G, \oplus\| (g) = \|G\| (U (g))\).
Graph Transformations

- Our framework generalises to tree/DAG-transformations
- Idea: attributes may contain trees/DAGs.
Graph Transformations

- Our framework generalises to tree/DAG-transformations
- **Idea:** attributes may contain trees/DAGs.
Graph Transformations

- Our framework generalises to tree/DAG-transformations
- **Idea:** attributes may contain trees/DAGs.
Graph Transformations

- Our framework generalises to tree/DAG-transformations
- Idea: attributes may contain trees/DAGs.
Example: Bird’s Repmin Problem

**newtype** $\text{Min}_S = \text{Min}_S \text{Int}$;  
**newtype** $\text{Min}_I = \text{Min}_I \text{Int}$

\[
\begin{align*}
\text{min}_S :: \text{Syn} \text{IntTreeF} \text{ atts} \text{Min}_S \\
\text{min}_S (\text{Leaf} \ i) & = \text{Min}_S i \\
\text{min}_S (\text{Node} \ a \ b) & = \text{min} \ (\text{below} \ a) \ (\text{below} \ b)
\end{align*}
\]

\[
\begin{align*}
\text{min}_I :: \text{Inh} \text{IntTreeF} \text{ atts} \text{Min}_I \\
\text{min}_I _\emptyset = \emptyset
\end{align*}
\]

\[
\begin{align*}
\text{rep} :: (\text{Min}_I \in \text{atts}) \Rightarrow \text{Rewrite} \text{IntTreeF} \text{ atts} \text{IntTreeF} \\
\text{rep} (\text{Leaf} \ i) & = \text{let} \ \text{Min}_I i' = \text{above} \\
& \quad \text{in} \ \text{Leaf} \ i' \\
\text{rep} (\text{Node} \ a \ b) & = \text{Node} \ a \ b
\end{align*}
\]
Example: Bird’s Repmin Problem

\textbf{newtype} \( \text{Min}_S = \text{Min}_S \text{Int} \);

\textbf{newtype} \( \text{Min}_I = \text{Min}_I \text{Int} \)

\( \text{min}_S :: \text{Syn} \text{IntTreeF atts Min}_S \)
\( \text{min}_S (\text{Leaf} \ i) = \text{Min}_S i \)
\( \text{min}_S (\text{Node} \ a \ b) = \text{min} (\text{below} \ a) (\text{below} \ b) \)

\( \text{rep} :: (\text{Min}_I \in \text{atts}) \Rightarrow \text{Rewrite} \text{IntTreeF atts IntTreeF} \)
\( \text{rep} (\text{Leaf} \ i) = \text{let} \ \text{Min}_I i' = \text{above} \)
\quad \text{in} \ \text{Leaf} \ i' \)
\( \text{rep} (\text{Node} \ a \ b) = \text{Node} \ a \ b \)

\( \text{repmin} :: \text{Tree} \text{IntTreeF} \rightarrow \text{Tree} \text{IntTreeF} \)
\( \text{repmin} = \text{runRewrite} \text{min}_S \text{min}_I \text{rep init} \)
\quad \text{where} \ \text{init} (\text{Min}_S i) = \text{Min}_I i \)
Example: Bird’s Repmin Problem

\textbf{newtype} \( \text{Min}_S = \text{Min}_S \text{Int} \); \hspace{1cm} \textbf{newtype} \( \text{Min}_I = \text{Min}_I \text{Int} \)

\( \min_S :: \text{Syn} \text{IntTreeF} \text{ atts} \text{Min}_S \)
\( \min_S (\text{Leaf } i) = \text{Min}_S i \)
\( \min_S (\text{Node } a \ b) = \min (\text{below } a) (\text{below } b) \)

\( \min_I :: \text{Inh} \text{IntTreeF} \text{ atts} \text{Min}_I \)
\( \min_I _- = \emptyset \)

\( \text{rep} :: (\text{Min}_I \in \text{atts}) \Rightarrow \text{Rewrite} \text{IntTreeF} \text{ atts} \text{IntTreeF} \)
\( \text{rep} (\text{Leaf } i) = \text{let} \ \text{Min}_I i' = \text{above} \)
\hspace{1cm} \text{in} \ \text{Leaf } i' \)
\( \text{rep} (\text{Node } a \ b) = \text{Node } a \ b \)

\( \text{repmin} :: \text{Tree} \text{IntTreeF} \to \text{Tree} \text{IntTreeF} \)
\( \text{repmin} = \text{runRewrite} \min_S \min_I \text{ rep } \text{init} \)
\hspace{1cm} \text{where} \ \text{init } (\text{Min}_S i) = \text{Min}_I i \)

\( \text{repmin}_G :: \text{Dag} \text{IntTreeF} \to \text{Dag} \text{IntTreeF} \)
\( \text{repmin}_G = \text{runRewriteDag} \text{const} \min_S \min_I \text{ rep } \text{init} \)
\hspace{1cm} \text{where} \ \text{init } (\text{Min}_S i) = \text{Min}_I i \)
Summary

Our Contributions

▶ Haskell library to run AGs on DAGs
▶ Correspondence & termination theorems to prove correctness
Summary

Our Contributions

▶ Haskell library to run AGs on DAGs
▶ Correspondence & termination theorems to prove correctness

More in the paper

▶ Examples: type inference; circuits
▶ full theory & proofs
▶ parametric AGs (→ tech report)
▶ Benchmarks (→ tech report)
Conclusion

Future and Ongoing Work

- AGs with fixpoint iteration \( \leadsto \) cyclic graphs
- mutually recursive data types and GADTs
- deep pattern matching in AGs
- corresponding notion of non-circularity for AGs on DAGs

Implementation


- Haskell library source code
- more examples
- benchmarks

Try the compositional datatypes library

```bash
$ cabal install compdata-dags
```
Conclusion

Future and Ongoing Work

- AGs with fixpoint iteration \(\leadsto\) cyclic graphs
- mutually recursive data types and GADTs
- deep pattern matching in AGs
- corresponding notion of non-circularity for AGs on DAGs

Implementation

Available from http://j.mp/AG-DAG.

- Haskell library source code
- more examples
- benchmarks
Conclusion

Future and Ongoing Work

- AGs with fixpoint iteration $\leadsto$ cyclic graphs
- mutually recursive data types and GADTs
- deep pattern matching in AGs
- corresponding notion of non-circularity for AGs on DAGs

Implementation


- Haskell library source code
- more examples
- benchmarks

Try the compositional datatypes library

> cabal install compdata-dags
Generalising Tree Traversals to DAGs

Exploiting Sharing without the Pain

Patrick Bahr\textsuperscript{1}  Emil Axelsson\textsuperscript{2}

\textsuperscript{1}University of Copenhagen
paba@diku.dk

\textsuperscript{2}Chalmers University of Technology
emax@chalmers.se

Source Code Repository

http://j.mp/AG-DAG

Haskell Library

> cabal install compdata-dags