Generalising Tree Traversals to DAGs

Exploiting Sharing without the Pain

Patrick Bahr
Department of Computer Science
University of Copenhagen
paba@di.ku.dk

Emil Axelsson
Department of Computer Science and Engineering
Chalmers University of Technology
emax@chalmers.se

Abstract

We present a recursion scheme based on attribute grammars that can be transparently applied to trees and acyclic graphs. Our recursion scheme allows the programmer to implement a tree traversal and then apply it to compact graph representations of trees instead. The resulting graph traversals avoid recomputation of intermediate results for shared nodes – even if intermediate results are used in different contexts. Consequently, this approach leads to asymptotic speedup proportional to the compression provided by the graph representation. In general, however, this sharing of intermediate results is not sound. Therefore, we complement our implementation of the recursion scheme with a number of correspondence theorems that ensure soundness for various classes of traversals. We illustrate the practical applicability of the implementation as well as the complementing theory with a number of examples.

Categories and Subject Descriptors D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming; F.3.3 [*Logics and Meanings of Programs*]: Studies of Program Constructs—Program and Recursion Schemes

Keywords attribute grammars, sharing, graph traversal, Haskell

1. Introduction

Functional programming languages such as Haskell excel at manipulating tree-structured data. Using algebraic data types, we can define functions over trees in a natural way by means of pattern matching and recursion. As an example, we take the following definition of binary trees with integer leaves, and a function to find the set of leaves at and below a given depth in the tree:

$$
\begin{array}{l}
\mathbf{data}\ \mathit{IntTree} = \mathit{Leaf}\ \mathit{Int} \mid \mathit{Node}\ \mathit{IntTree}\ \mathit{IntTree} \\
\mathit{leavesBelow} :: \mathit{Int} \to \mathit{IntTree} \to \mathit{Set}\ \mathit{Int} \\
\mathit{leavesBelow}\ d\ (\mathit{Leaf}\ i) \\
\quad \mid d \leqslant 0 \qquad\qquad = \mathit{Set.singleton}\ i \\
\quad \mid \mathit{otherwise} \qquad = \mathit{Set.empty} \\
\mathit{leavesBelow}\ d\ (\mathit{Node}\ t_1\ t_2) = \\
\quad \mathit{leavesBelow}\ (d-1)\ t_1\ \cup\ \mathit{leavesBelow}\ (d-1)\ t_2
\end{array}
$$

One shortcoming of tree structures is that they are unable to represent sharing of common subtrees, which occur, for example, when a compiler substitutes a shared variable by its definition. The following tree has a shared node a that appears twice:

$$
\begin{array}{l}
t = \mathbf{let}\ a = \mathit{Node}\ (\mathit{Node}\ (\mathit{Leaf}\ 2)\ (\mathit{Leaf}\ 3))\ (\mathit{Leaf}\ 4) \\
\quad\ \ \mathbf{in}\ \ \mathit{Node}\ a\ a
\end{array}
$$

Unfortunately, a function like $\mathit{leavesBelow}$ is unable to observe this sharing, and thus needs to traverse the shared subtree in t twice.

In order to represent and take advantage of sharing, one could instead use a directed graph representation, such as the structured graphs of Oliveira and Cook [32]. However, such a change of representation would force us to express $\mathit{leavesBelow}$ by traversing the graph structure instead of by plain recursion over the Node constructors. If we are only interested in graphs as a compact representation of trees, this is quite a high price to pay. In an ideal world, one should be able to leave the definition of $\mathit{leavesBelow}$ as it is, and be able to run it on both trees and graphs.

Oliveira and Cook [32] define a fold operation for structured graphs which makes it possible to define structurally recursive functions as algebras that can be applied to both trees and graphs. However, $\mathit{leavesBelow}$ is a context-dependent function that passes the depth parameter down the recursive calls. Therefore, an implementation as a fold – namely by computing a function from context to result – would not be able to exploit the sharing present in the graph: intermediate results for shared nodes still have to be recomputed for each context in which they are used. Moreover, it is not possible to use folds to transform a graph without losing sharing.

This paper presents a method for running tree traversals on directed acyclic graphs (DAGs), taking full account of the sharing structure. The traversals are expressed as attribute grammars (AGs) using Bahr's representation of tree automata in Haskell [5]. The underlying DAG structure is completely transparent to the AGs, which means that the same AG can be run on both trees and DAGs. The main complication arises for algorithms that pass an accumulating parameter down the tree. In a DAG this may lead to a shared node receiving conflicting values for the accumulating parameter. Our approach is to resolve such conflicts using a separate user-provided function. For example, in $\mathit{leavesBelow}$, the resolution function for the depth parameter would be min, since we only need to consider the deepest occurrence of each shared subtree. As we will show, this simple insight extends to many tree traversals of practical relevance.

The paper makes the following contributions:

- We present an implementation of AGs in Haskell that allows us to write tree traversals such that they can be applied to compact DAG representations of trees as well.
- We extend AGs with rewrite functions to implement tree transformations that preserve sharing if applied to DAGs.

- We prove a number of general correspondence theorems that relate the semantics of AGs on trees to their semantics on corresponding DAG representations. These correspondence results allow us to prove the soundness of our approach for various classes of traversals.
- Our implementation and the accompanying theory covers an important class of algorithms, where an inherited attribute maintains a variable environment. This makes our method suitable for certain syntactic analyses, for instance in a compiler. We demonstrate this fact on a type inference implementation.

The rest of the paper is organised as follows: Section 2 presents embedded domain-specific languages, which are an important motivation for this work. Section 3 introduces recursion schemes based on AGs, and section 4 shows how to run AGs on DAGs. Section 5 gives the semantics and theoretical results for reasoning about AGs on trees and DAGs. Some proofs were elided or abridged to save space. The full proofs are presented in the accompanying technical report [6]. Likewise, the exact implementation of the recursion schemes is omitted. It is instead available in an accompanying repository: `https://github.com/emilaxelsson/ag-graph`.

2. Running Example

To illustrate the ideas in this paper, we will use the following simple expression language:

```
data Exp = LitB Bool                -- Boolean literal
         | LitI Int                 -- Integer literal
         | Eq Exp Exp               -- Equality
         | Add Exp Exp              -- Addition
         | If Exp Exp Exp           -- Condition
         | Var Name                 -- Variable
         | Iter Name Exp Exp Exp    -- Iteration

type Name = String
```

Most constructs in *Exp* have a straightforward meaning. For example, the following is a conditional expression that corresponds to the Haskell expression **if** $x \equiv 0$ **then** 1 **else** 2:

```
If (Eq (Var "x") (LitI 0)) (LitI 1) (LitI 2)
```

However, *Iter* requires some explanation. This is a looping construct that corresponds to the following Haskell function:

```
iter :: Int → s → (s → s) → s
iter 0 s b = s
iter n s b = iter (n − 1) (b s) b
```

The expression *iter n s b* applies the b function n times starting in state s. The corresponding expression *Iter* `"x"` $n\ s\ b$ (where $n, s, b :: \mathit{Exp}$) works in the same way. However, since we do not have functions in the *Exp* language, the first argument of *Iter* is a variable name, and this name is bound in the the body b. For example, the Haskell expression *iter* $5\ 1\ (\lambda s \rightarrow s + 2)$ is represented as

```
Iter "s" (LitI 5) (LitI 1) (Add (Var "s") (LitI 2))
```

2.1 Type Inference

A typical example of a function over expressions that has an interesting flow of information is simple type inference, defined in Figure 1. The first argument is the environment – a mapping from bound variables to their types. Most of the cases just check the types of the children and return the appropriate type. The environment is passed unchanged to the recursive calls, except in the *Iter* case, where the bound variable is added to the environment. The only case where the environment is used is in the *Var* case, where the type of the variable is obtained by looking it up in the environment.

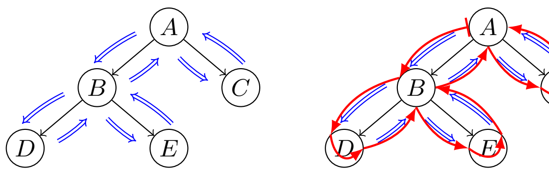
```
data Type = BoolType | IntType deriving (Eq)
type Env  = Map Name Type

typeInf :: Env → Exp → Maybe Type
typeInf env (LitB _)              = Just BoolType
typeInf env (LitI _)              = Just IntType
typeInf env (Eq a b)
   | Just ta        ← typeInf env a
   , Just tb        ← typeInf env b
   , ta ≡ tb                      = Just BoolType
typeInf env (Add a b)
   | Just IntType   ← typeInf env a
   , Just IntType   ← typeInf env b = Just IntType
typeInf env (If c t f)
   | Just BoolType ← typeInf env c
   , Just tt        ← typeInf env t
   , Just tf        ← typeInf env f
   , tt ≡ tf                      = Just tt
typeInf env (Var v)               = lookEnv v env
typeInf env (Iter v n i b)
   | Just IntType   ← typeInf env n
   , ti'@(Just ti)  ← typeInf env i
   , Just tb        ← typeInf (insertEnv v ti' env) b
   , ti ≡ tb                      = Just tb
typeInf _ _                       = Nothing

insertEnv :: Name → Maybe Type → Env → Env
insertEnv v Nothing env = env
insertEnv v (Just t) env = Map.insert v t env

lookEnv :: Name → Env → Maybe Type
lookEnv = Map.lookup
```

Figure 1: Type inference for example EDSL.

Note that *typeInf* has many similarities with *leavesBelow* from the introduction: It is defined using recursion over the tree constructors; it passes an accumulating parameter down the recursive calls; it synthesises a result from the results of the recursive calls. Naturally, it also has the same problems as *leavesBelow* when applied to an expression with shared sub-expressions: It will repeatedly infer types for shared sub-expressions each time they occur.

This issue can be resolved by adding a *let binding* construct to *Exp* in order to explicitly represent shared sub-expressions. The type inference algorithm can then be extended to make use of this sharing information. However, let bindings tend to get in the way of syntactic simplifications, which is why optimising compilers often try to inline let bindings in order to increase the opportunities for simplification. In general, it is not possible to inline all let bindings, as this can lead to unmanageably large ASTs. This leaves the compiler with the tricky problem of inlining enough to trigger the right simplifications, but not more than necessary so that the AST does not explode.

Ideally, one would like to program syntactic analyses and transformations without having to worry about sharing, especially if the sharing is only used to manage the size of the AST. The method proposed in this paper makes it possible to traverse expressions *as if all sharing was inlined*, yet one does not have to pay the price



(a) Attribute propagation. (b) Tree traversal.

Figure 2: Propagation of attribute values by an attribute grammar.

of duplicated sub-expressions, since the internal representation of expressions is an acyclic graph.

3. Attribute Grammars

In this section we describe the representation and implementation of attribute grammars in Haskell. The focus of our approach is put on a simple representation of this recursion scheme that at the same time allows us to easily move from tree-structured data to graph-structured data. To this end, we represent tree-structured data as fixed points of functors:

$$\mathbf{data}\ \mathit{Tree}\ f = \mathit{In}\ (f\ (\mathit{Tree}\ f))$$

For instance, to represent the type Exp, we define a corresponding functor ExpF below, which gives us the type $\mathit{Tree}\ \mathit{ExpF}$ isomorphic to Exp (modulo non-strictness):

$$\begin{aligned}\mathbf{data}\ \mathit{ExpF}\ a = {} & \mathit{LitB}\ \mathit{Bool} \mid \mathit{LitI}\ \mathit{Int} \mid \mathit{Var}\ \mathit{Name}\\ \mid {} & \mathit{Eq}\ a\ a \quad\ \mid \mathit{Add}\ a\ a \mid \mathit{If}\ a\ a\ a\\ \mid {} & \mathit{Iter}\ \mathit{Name}\ a\ a\ a\end{aligned}$$

Apart from requiring functors such as ExpF to be instances of $\mathit{Functor}$, we also require them to be instances of the $\mathit{Traversable}$ type class. This will keep the representation of our recursion scheme on trees simple and is indeed necessary in order to implement it on DAGs. Haskell is able to provide such instances automatically via its **deriving** clause.

An attribute grammar (AG) consists of a number of *attributes* and a collection of *semantic functions* that compute these attributes for each node of the tree. One typically distinguishes between *inherited attributes*, which are computed top-down, and *synthesised attributes*, which are computed bottom-up. For instance, if we were to express the type inference algorithm $\mathit{typeInf}$ as an AG, it would consist of an inherited attribute that is the environment and a synthesised attribute that is the inferred type.

Figure 2a illustrates the propagation of attribute values of an AG in a tree. The arrows facing upwards and downwards represent the propagation of synthesised and inherited attributes, respectively. Due to this propagation, the semantic functions that compute the attribute values for each node n have access to the attribute values in the corresponding neighbourhood of n. For example, to compute the inherited attribute value that is passed down from B to D, the semantic function may use the inherited attributes from A and the synthesised attributes from D and E. This scheme allows for complex interdependencies between attributes. Provided that there are no cyclic dependencies, a traversal through the tree that computes all attribute values of each node can be executed as illustrated in Figure 2b.

3.1 Synthesised Attributes

We defer the formal treatment of AGs until section 5 and focus on the implementation in Haskell for now. We start with the simpler case, namely synthesised attributes. The computation of synthesised attributes follows essentially the same structure as a fold, i.e. the following recursion scheme:

$$\mathbf{type}\ \mathit{Algebra}\ f\ c = f\ c \to c$$

$$\begin{aligned}&\mathit{fold} :: \mathit{Functor}\ f \Rightarrow \mathit{Algebra}\ f\ c \to \mathit{Tree}\ f \to c\\ &\mathit{fold}\ \mathit{alg}\ (\mathit{In}\ t) = \mathit{alg}\ (\mathit{fmap}\ (\mathit{fold}\ \mathit{alg})\ t)\end{aligned}$$

The algebra of a fold describes how the value of type c for a node in the tree is computed given that it has already been computed for its children.

AGs go beyond this recursion scheme: they allow us to use not only values of the attribute of type c being defined but also other attributes, which are computed by other semantic functions. To express that an attribute of type c is part of a larger collection of attributes, we use the following type class

$$\begin{aligned}&\mathbf{class}\ c \in \mathit{as}\ \mathbf{where}\\ &\quad \mathit{pr} :: \mathit{as} \to c\end{aligned}$$

Intuitively, $c \in \mathit{as}$ means that c is a component of as, and gives the corresponding projection function. We can give instance declarations accordingly, which gives us for example that $a \in (a, b)$ with the projection function $\mathit{pr} = \mathit{fst}$. Using closed type families [13], the type class $\in$ can be defined such that it works on arbitrarily nested product types, but disallows ambiguous instances such as $\mathit{Int} \in (\mathit{Int}, (\mathit{Bool}, \mathit{Int}))$ for which multiple projections exist. But there are also simpler implementations of $\in$ that only use type classes [5].

We can thus represent the semantic function for a synthesised attribute of type s as follows:

$$\mathbf{type}\ \mathit{Syn}\ f\ \mathit{as}\ s = (s \in \mathit{as}) \Rightarrow \mathit{as} \to f\ \mathit{as} \to s$$

To compute the attribute of type s we can draw from the complete set of attributes of type as at the current node as well as its children.

For example, the following excerpt gives one case for the synthesised type attribute of type inference (cf. the reference implementation in Figure 1):

$$\begin{aligned}&\mathit{typeInf}_S :: \mathit{Syn}\ \mathit{ExpF}\ \mathit{as}\ (\mathit{Maybe}\ \mathit{Type})\\ &\mathit{typeInf}_S\ \_\ (\mathit{Add}\ a\ b)\\ &\quad \mid \mathit{Just}\ \mathit{IntType} \leftarrow \mathit{pr}\ a\\ &\quad ,\ \mathit{Just}\ \mathit{IntType} \leftarrow \mathit{pr}\ b = \mathit{Just}\ \mathit{IntType}\\ &\ \ldots\end{aligned}$$

However, instead of the above Syn type, we shall use a more indirect representation, which will turn out to be beneficial for the representation of inherited attributes, and later for rewrite functions. It is based on the isomorphism below, which follows from the Yoneda Lemma for all functors f and types as, s:

$$(\forall a.(a \to \mathit{as}) \to (f\ a \to s)) \cong f\ \mathit{as} \to s$$

It allows us to define the type $\mathit{Syn}\ f\ \mathit{as}\ s$ alternatively as follows:

$$\forall\ a.(s \in \mathit{as}) \Rightarrow \mathit{as} \to (a \to \mathit{as}) \to f\ a \to s$$

We further transform this type by turning the first two arguments of type as and $a \to \mathit{as}$ into implicit parameters [29], which provides an interface closer to that of AG systems:

$$\begin{aligned}\mathbf{type}\ \mathit{Syn}\ f\ \mathit{as}\ s = \forall\ a.(&?\mathit{below} :: a \to \mathit{as}, ?\mathit{above} :: \mathit{as},\\ &s \in \mathit{as}) \Rightarrow f\ a \to s\end{aligned}$$

Combining the implicit parameters with projection gives us two convenient helper functions for writing semantic functions:

$$\begin{aligned}&\mathit{above} :: (?\mathit{above} :: \mathit{as}, i \in \mathit{as}) \Rightarrow i\\ &\mathit{above} = \mathit{pr}\ (?\mathit{above})\end{aligned}$$

$$\begin{aligned}&\mathit{below} :: (?\mathit{below} :: a \to \mathit{as}, s \in \mathit{as}) \Rightarrow a \to s\\ &\mathit{below}\ a = \mathit{pr}\ (?\mathit{below}\ a)\end{aligned}$$

The complete definition of the synthesised type attribute for type inference is given in Figure 3. The function $\mathit{typeInf}_I$ is the semantic function for the inherited environment attribute. It will

```
typeInf_S :: (Env ∈ as) ⇒ Syn ExpF as (Maybe Type)
typeInf_S (LitB _)                  = Just BoolType
typeInf_S (LitI _)                  = Just IntType
typeInf_S (Eq a b)
   | Just ta       ← typeOf a
   , Just tb       ← typeOf b
   , ta ≡ tb                        = Just BoolType
typeInf_S (Add a b)
   | Just IntType  ← typeOf a
   , Just IntType  ← typeOf b = Just IntType
typeInf_S (If c t f)
   | Just BoolType ← typeOf c
   , Just tt       ← typeOf t
   , Just tf       ← typeOf f
   , tt ≡ tf                        = Just tt
typeInf_S (Var v)                   = lookEnv v above
typeInf_S (Iter v n i b)
   | Just IntType  ← typeOf n
   , Just ti       ← typeOf i
   , Just tb       ← typeOf b
   , ti ≡ tb                        = Just tb
typeInf_S _                         = Nothing

typeInf_I :: (Maybe Type ∈ as) ⇒ Inh ExpF as Env
typeInf_I (Iter v n i b) = b ↦ insertEnv v ti above
                              where ti = typeOf i
typeInf_I _              = ∅
```

Figure 3: Semantic functions for synthesised and inherited attributes of type inference.

be explained in the following subsection. The code uses a convenient helper function for querying the synthesised type of a sub-expression:

```
typeOf :: (?below :: a → as, Maybe Type ∈ as) ⇒
        a → Maybe Type
typeOf = below
```

3.2 Inherited Attributes

The representation of semantic functions defining inherited attributes is slightly more complicated, which is to say that there is no representation that is both elegant and convenient to use. We need to represent a mapping that assigns attribute values to the children of a node. The most convenient way to represent such mappings in Haskell is in the form of a finite mapping provided by the type constructor *Map*.

Thus, given a node of type $f\ a$, where type a represents child positions of the node, we assign inherited attribute values of type i to each child node by providing a mapping of type $Map\ a\ i$. This gives us the following representation of semantic functions for inherited attributes:

```
type Inh f as i = ∀ a.(?below :: a → as, ?above :: as,
                         i ∈ as, Ord a) ⇒ f a → Map a i
```

Note that we have to add the constraint $Ord\ a$, since the operations to construct finite mappings require this.

The above type does not ensure that the returned mapping is complete, i.e. that each child is assigned a value. However, this situation provides the opportunity to allow so-called *copy rules*. Such copy rules are a common convenience feature in AG systems and state when inherited attributes are simply propagated to a child. In our case, we copy an inherited attribute value to a child if no assignment is made in the mapping of the semantic function.

To make it convenient to construct mappings as the result of semantic functions for inherited attributes, we define infix operators $\mapsto$ and &, which allow us to construct singleton mappings $x \mapsto y$ and take the union $m\ \&\ n$ of two mappings. Moreover, we use $\emptyset$ to denote the empty mapping.

The semantic function for the inherited environment attribute of type inference is given by $typeInf_I$ in Figure 3. The only interesting case is *Iter*, in which the local variable is inserted into the environment. The environment is only updated for the sub-expression b (because it only scopes over the body of the loop). Hence, the other sub-expressions (n and i) will get an unchanged environment by the abovementioned copy rule. Similarly, for all other constructs in the EDSL, the environment is copied unchanged.

3.3 Combining Semantic Functions to Attribute Grammars

Now that we have Haskell representations for semantic functions, we need combinators that allow us to combine them to form complete AGs.

At first, we define combinators that combine two semantic functions to obtain a semantic function that computes the attributes of both of them. For synthesised attributes, this construction is simple:

```
(⊗) :: Syn f as s₁ → Syn f as s₂ → Syn f as (s₁, s₂)
(sp ⊗ sq) t = (sp t, sq t)
```

The implementation for inherited attributes is more difficult as we have to honour the copy rule. That is, given two semantic functions i and j, where i assigns an attribute value for a given child node but j does not, the product of i and j must assign an attribute value consisting of the value given by i and a copy for the second attribute. We elide the details of the implementation and instead give only the type of the corresponding combinator:

```
(⊛) :: Functor f ⇒
        Inh f as i₁ → Inh f as i₂ → Inh f as (i₁, i₂)
```

Finally, a complete AG is given by a semantic function of type $Syn\ f\ (s,i)\ s$ and another one of type $Inh\ f\ (s,i)\ i$. That is, taken together the two semantic functions define the full attribute space (s,i). Moreover, we have to provide an initial value of the inherited attribute of type i in order to run the AG on an input tree of type $Tree\ f$. In general, the initial value of the inherited attributes does not have to be fixed but may depend on (some of) the synthesised attributes. These constraints are summarised in the type of the function that implements the run of an AG:

```
runAG :: Traversable f ⇒ Syn f (s, i) s →
            Inh f (s, i) i → (s → i) → Tree f → s
```

We are now able to define type inference as a run of the AG defined in Figure 3:

```
typeInf :: Env → Tree ExpF → Maybe Type
typeInf env = runAG typeInf_S typeInf_I (λ_ → env)
```

In this example, the initialisation function for the inherited attribute is simply a constant function that returns the environment. In the next section, we shall see an example that uses the full power of the initialisation function.

3.4 Example: Richard Bird's *repmin*

A classic example of a tree traversal with interesting information flow is Bird's *repmin* problem [8]. The problem is as follows: given a tree with integer leaves, compute a new tree of the same shape but where all leaves have been replaced by the minimal leaf in the original tree. Bird shows how this can be achieved by a single traversal in a lazy functional language.

To code *repmin* as an AG, we first define a functor corresponding to the tree type from section 1:

$$\textbf{data}\ \mathit{IntTreeF}\ a = \mathit{Leaf}\ \mathit{Int} \mid \mathit{Node}\ a\ a$$

Next, we introduce two attributes:

$$\begin{array}{l}\textbf{newtype}\ \mathit{Min}_S = \mathit{Min}_S\ \mathit{Int}\ \textbf{deriving}\ (\mathit{Eq}, \mathit{Ord})\\ \textbf{newtype}\ \mathit{Min}_I = \mathit{Min}_I\ \mathit{Int}\end{array}$$

Min_S is the synthesised attribute representing the smallest integer in a subtree, and Min_I is the inherited attribute which is going to be the smallest integer in the whole tree. We also define a convenience function for accessing the Min_I attribute:

$$\begin{array}{l}\mathit{globMin} :: (?\mathit{above} :: \mathit{as}, \mathit{Min}_I \in \mathit{as}) \Rightarrow \mathit{Int}\\ \mathit{globMin} = \textbf{let}\ \mathit{Min}_I\ i = \mathit{above}\ \textbf{in}\ i\end{array}$$

The semantic function for the Min_S attribute is as follows:

$$\begin{array}{l}\mathit{min}_S :: \mathit{Syn}\ \mathit{IntTreeF}\ \mathit{as}\ \mathit{Min}_S\\ \mathit{min}_S\ (\mathit{Leaf}\ i) \quad\ \ = \mathit{Min}_S\ i\\ \mathit{min}_S\ (\mathit{Node}\ a\ b) = \mathit{min}\ (\mathit{below}\ a)\ (\mathit{below}\ b)\end{array}$$

The Min_I attribute should be the same throughout the whole tree, so we define a function that just copies the inherited attribute:

$$\begin{array}{l}\mathit{min}_I :: \mathit{Inh}\ \mathit{IntTreeF}\ \mathit{as}\ \mathit{Min}_I\\ \mathit{min}_I\ \_ = \emptyset\end{array}$$

Finally, we need to be able to synthesise a new tree that depends on the globally smallest integer available from the Min_I attribute:

$$\begin{array}{l}\mathit{rep} :: (\mathit{Min}_I \in \mathit{as}) \Rightarrow \mathit{Syn}\ \mathit{IntTreeF}\ \mathit{as}\ (\mathit{Tree}\ \mathit{IntTreeF})\\ \mathit{rep}\ (\mathit{Leaf}\ i) \quad\ \ = \mathit{In}\ (\mathit{Leaf}\ \mathit{globMin})\\ \mathit{rep}\ (\mathit{Node}\ a\ b) = \mathit{In}\ (\mathit{Node}\ (\mathit{below}\ a)\ (\mathit{below}\ b))\end{array}$$

Now we have all the parts needed to define *repmin*:

$$\begin{array}{l}\mathit{repmin} :: \mathit{Tree}\ \mathit{IntTreeF} \rightarrow \mathit{Tree}\ \mathit{IntTreeF}\\ \mathit{repmin} = \mathit{snd} \circ \mathit{runAG}\ (\mathit{min}_S \otimes \mathit{rep})\ \mathit{min}_I\ \mathit{init}\\ \quad \textbf{where}\ \mathit{init}\ (\mathit{Min}_S\ i, \_) = \mathit{Min}_I\ i\end{array}$$

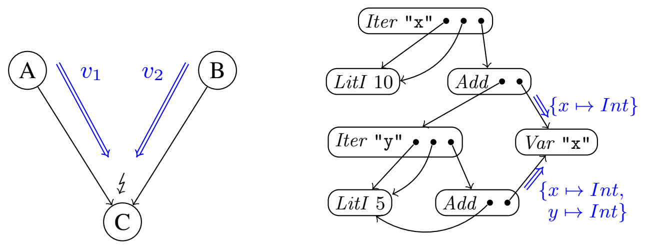
The *init* function uses the synthesised smallest integer as the initial inherited attribute value.

3.5 Informal Semantics

Instead of reproducing the implementation of runAG here, we shall informally describe the semantics of an AG and describe in which way runAG implements this semantics. The formal semantics is given in section 5.

The semantic functions of an AG describe how to compute the value of an attribute at a node n using the attributes in the "neighbourhood" of n. For synthesised attributes, this neighbourhood consists of n itself and its children, whereas for inherited attributes, it consists of n, its siblings, and its parent. Running the AG on a tree t amounts to computing, for each attribute a, the mapping $\rho_a : N \rightarrow D_a$ from the set of nodes of t to the set of values of a. In other words, the tree is decorated with the computed attribute values. We call the collection of all these mappings ρ_a a *run* of the AG on t. In general, there may not be a unique run (including no run at all), since there can be a cyclic dependency between the attributes. However, if there is no such cyclic dependency, runAG will effectively construct the unique run of the AG on the input tree, and return the product of all synthesised attribute values at the root of the tree.

Figure 2b illustrates how runAG may compute the run of a given AG by a traversal through the tree. Such a traversal is, however, not statically scheduled in advance but rather dynamically exploiting Haskell's lazy semantics.



(a) Abstract problem. (b) Example: type inference.

Figure 4: Confluence of inherited attributes.

4. Attribute Grammars on DAGs

Our goal is to apply algorithms intended to work on trees without or with only little change to DAGs such that we can exploit the sharing for performance gains. The key observation that allows us to do this is the fact that AGs are unaware of the underlying representation they are working on. Semantic functions simply compute attributes of a node using attribute values in the neighbourhood of the node. The informal semantics of AGs on trees given in section 3.5 is equally applicable to DAGs.

This straightforward translation of the semantics to DAGs, however, will rarely yield a well-defined run. The problem is that in the presence of sharing – i.e. there is a node with more than one incoming edge – the semantic function for an inherited attribute may overlap: it assigns potentially different values to the same attribute at the same node. Figure 4a illustrates the problem: the semantic function for the inherited attribute computes for each of the two nodes A and B the value v_1 resp. v_2 of the inherited attribute that should be passed down to the child node of A resp. B. However, A and B share the same child, C, which therefore receives both values for the inherited attribute.

The easiest way to deal with this situation is to traverse the sub-DAG reachable from C multiple times – once for each of the conflicting attribute values v_1 and v_2. This is what happens if we would implement traversals as folds in the style of Oliveira and Cook [32]. But our goal is to avoid such recomputation.

A simple special case is if we know that v_1 and v_2 are always the same. That happens, for example, if inherited attributes are only copied downwards as in the repmin example from section 3.4. However, for the type inference AG, this is clearly not the case. One example that shows the problem is the DAG in Figure 4b, where the shared variable "x" is used in two different environments.

Nonetheless, for type inference, as for many other AGs of interest, we can still extend the semantics to DAGs in a meaningful way by providing a commutative, associative operator $\oplus$ on inherited attributes that combines confluent attribute values. In the illustration in Figure 4a, the inherited attribute at C is then assigned the value $v_1 \oplus v_2$. For the type inference AG, a (provably) sensible choice for $\oplus$ is the intersection of environments (cf. section 4.3). In Figure 4b, intersecting the environments of the node Var "x" yields the environment $\{x \mapsto \mathit{Int}\}$.

This observation allows us to efficiently run AGs on DAGs. Our implementation provides a corresponding variant of runAG:

$$\begin{array}{l}\mathit{runAGDag} :: \mathit{Traversable}\ f \Rightarrow (i \rightarrow i \rightarrow i) \rightarrow\\ \quad \mathit{Syn}\ f\ (s,i)\ s \rightarrow \mathit{Inh}\ f\ (s,i)\ i \rightarrow (s \rightarrow i) \rightarrow \mathit{Dag}\ f \rightarrow s\end{array}$$

The interface differs in two points from runAG: (1) it takes DAGs as input and (2) it takes a binary operator of type $i \rightarrow i \rightarrow i$, which is to be used to combine confluent attributes as described above.

For instance, we may use the type inference AG to implement type inference on DAGs as follows:

$$
\begin{aligned}
&\mathit{typeInf}_G :: \mathit{Env} \to \mathit{Dag}\ \mathit{ExpF} \to \mathit{Maybe}\ \mathit{Type}\\
&\mathit{typeInf}_G\ \mathit{env} = \mathit{runAGDag}\ \mathit{intersection}\\
&\qquad\qquad\qquad\qquad \mathit{typeInf}_S\ \mathit{typeInf}_I\ (\lambda\_ \to \mathit{env})
\end{aligned}
$$

We will not go into the details of the implementation of *runAGDag*. Instead we refer to the informal semantics that we have given above as well as the formal semantics given in section 5

But we shall briefly explain how DAGs of type $\mathit{Dag}\ f$ are represented. We represent DAGs with explicit nodes and edges, with nodes represented by integers:

$$\mathbf{type}\ \mathit{Node} = \mathit{Int}$$

Edges are represented as finite mappings from *Node* into $f\ \mathit{Node}$. In this way, each node is mapped to all its children, but also its labelling. In addition, each node has a designated root node. This gives the following definition of *Dag* as a record type:

$$
\begin{aligned}
&\mathbf{data}\ \mathit{Dag}\ f = \mathit{Dag}\ \{\mathit{root}\ \ :: \mathit{Node},\\
&\qquad\qquad\qquad\quad \mathit{edges} :: \mathit{IntMap}\ (f\ \mathit{Node})\}
\end{aligned}
$$

Note that acyclicity is not explicitly encoded in this definition of DAGs. Instead, we rely on the combinators to construct such DAGs to ensure or check for acyclicity.

Following Gill [20], we provide a function that observes the implicit sharing of a tree of type $\mathit{Tree}\ f$ and turns it into a DAG of type $\mathit{Dag}\ f$:

$$\mathit{reifyDag} :: \mathit{Traversable}\ f \Rightarrow \mathit{Tree}\ f \to \mathit{IO}\ (\mathit{Dag}\ f)$$

As a final example, we turn the *repmin* function from section 3.4 into a function repmin_G that works on DAGs.

$$
\begin{aligned}
&\mathit{repmin}_G :: \mathit{Dag}\ \mathit{IntTreeF} \to \mathit{Tree}\ \mathit{IntTreeF}\\
&\mathit{repmin}_G = \mathit{snd} \circ \mathit{runAGDag}\ \mathit{const}\ (\mathit{min}_S \otimes \mathit{rep})\ \mathit{min}_I\ \mathit{init}\\
&\quad \mathbf{where}\ \mathit{init}\ (\mathit{Min}_S\ i, \_) = \mathit{Min}_I\ i
\end{aligned}
$$

The only additional definition we have to provide is the function to combine inherited attribute values, for which we choose *const*, i.e. we arbitrarily pick one of the values. The rationale behind this choice is that the value of inherited attribute – computed by min_I – is globally the same since it is copied. The formal justification for this choice is given in section 4.1 below.

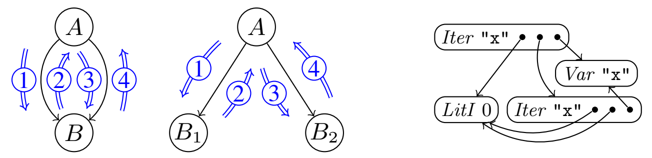
The type of repmin_G indicates that it is not quite the function we had hoped for: it returns a tree rather than a DAG. We defer addressing this issue until section 4.5.

4.1 Trees vs. DAGs

The most important feature of our approach is that we can express the semantics of an AG on DAGs in terms of the semantics on trees. This is achieved by two correspondence theorems that relate the semantics of AGs on DAGs to the semantics on trees. The theorems are discussed and proved in section 5. But we present them here informally and illustrate their applicability to the examples that we have seen so far.

To bridge the gap between the tree and the DAG semantics of AGs, we use the notion of *unravelling* (or *unsharing*) of a DAG g to a tree $\mathcal{U}(g)$, which is the uniquely determined tree $\mathcal{U}(g)$ that is bisimilar to g. Since we only consider finite acyclic graphs g, the unravelling $\mathcal{U}(g)$ is always a finite tree. The correspondence theorems relate the result of running an AG on a DAG g to the result of running it on the unravelling of g. The practical relevance of these theorems stems from the fact that *reifyDag* turns a tree t into a DAG g that unravels to t.

The first and simplest correspondence result is applicable to all so-called *copying* AGs, which are AGs that copy all their inherited attributes. That is, in concrete terms, the semantic function of each inherited attribute returns the empty mapping $\emptyset$. Such AGs are by no means trivial, since inherited attributes may still be initialised as a function on the synthesised attributes. The repmin AG is, for



(a) A DAG g. (b) Unravelling of g. (c) Example DAG.

Figure 5: Cyclic dependency in non-circular AG on DAGs.

example, copying. The following correspondence theorem is thus applicable to repmin:

Theorem 1 (sketch)**.** *Given a copying AG G, a binary operator* $\oplus$ *on inherited attributes with* $x \oplus y \in \{x, y\}$ *for all* x, y*, and a DAG g, we have that G terminates on* $\mathcal{U}(g)$ *with result r iff* $(G, \oplus)$ *terminates on g with result r.*

The above theorem is immediately applicable to the repmin AG. We obtain that repmin_G applied to a DAG g yields the same result as *repmin* applied to $\mathcal{U}(g)$. That is, we get the same result for $\mathit{repmin}\ t$ and $\mathit{fmap}\ \mathit{repmin}_G\ (\mathit{reifyDag}\ t)$.

Before we discuss the second correspondence theorem we have to consider the termination behaviour of AGs on trees vs. DAGs.

4.2 Termination of Attribute Grammars

While AGs are quite flexible in the interdependency between attributes they permit – which in general may lead to cyclic dependencies and thus non-termination – they come with a tool set to check for circular dependencies. Already when Knuth [25, 26] introduced AGs, he gave an algorithm to check for circular dependencies and proved that absent such circularity AGs terminate.

This result also applies to our AGs. And the example AGs we have considered this far are indeed non-circular – *runAG* will terminate for them (given any finite tree as input). Somewhat surprisingly this property does not carry over to acyclic graphs.

The essence of the phenomenon that causes this problem is illustrated in Figure 5. Figure 5a shows a simple DAG consisting of two nodes, and Figure 5b its unravelling to a tree. The double arrows illustrate the flow of information from a run of an AG. The numbers indicate the order in which the information flows: we first pass information from A to B_1 (via the inherited attribute) then from B_1 back to A (via the synthesised attribute) and then similarly to and from B_2. This is a common situation, which one e.g. finds in type inference. The underlying AG is non-circular, and the numbering indicates the order in which attributes are computed and then propagated.

However, in a DAG the two children of A may very well be shared, i.e. represented by a single node B. This causes a cyclic dependency, which can be observed in Figure 5a: information flow (2) can only occur after (1) and (3), as only then all the information coming to B has been collected. But (3) itself depends on (2).

Cyclic dependencies can easily occur with the type inference AG. In the DAG in Figure 5c the lower *Iter* loop computes the initial state of the upper *Iter* loop, and both loops use the variable `"x"` for the state. The variable node inherits two environments – one from each of the *Iter* nodes – which are resolved by intersection. Thus, the type of the variable depends on the environment from the upper loop, which depends on the type of the lower loop, which in turn depends on the type of the variable.

Semantically, the non-termination manifests itself in the lack of a unique run. While the type inference AG has a unique run on the unravelling of this DAG, there are exactly two distinct runs on the DAG itself: one in which the *Var* `"x"` node is given the synthesised attribute value *Nothing* and another one in which it is given the

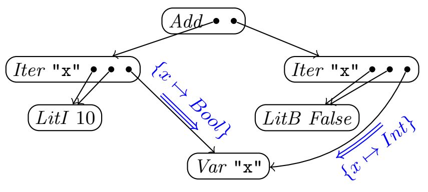


Figure 6: DAG that is not well-scoped.

value $Just\ IntType$. We discuss how to resolve this issue in the next section.

Note that this issue cannot occur for the repmin example. The repmin AG is non-circular and thus terminates on trees. By virtue of Theorem 1, it thus terminates on DAGs as well.

4.3 Correspondence by Monotonicity

Relating the semantics of the type inference AG on trees to its semantics on DAGs is much more difficult – even if the issue of termination is sorted out. In general, for these kinds of AGs, we do not have a simple equality relation as we have for copying AGs. In fact, it should be expected that type inference on a DAG is more restrictive than on its unravelling: a node that is shared in a DAG can only be assigned a single type, whereas its corresponding copies in the unravelling may have different types.

However, we can prove the following property: if the type inference AG infers a type t for a DAG g, then it infers the same type t for $\mathcal{U}(g)$. This soundness property follows immediately from a more general *monotonicity* correspondence theorem.

In order to apply this theorem, we have to find, for each attribute a, a quasi-order $\lesssim$ on the values of attribute a, such that each semantic function f is monotone w.r.t. these quasi-orders. That is, given two sets of inputs A and B, with B bigger than A, also the result of f applied to B is bigger than f applied to A. We say that an AG is monotone w.r.t. $\lesssim$, if each semantic function is. Moreover, we require the binary operator $\oplus$ on inherited attributes be *decreasing* w.r.t. the order $\lesssim$, i.e. $x \oplus y \lesssim x, y$.

Theorem 2 (sketch)**.** *Let G be a non-circular AG, $\oplus$ an associative, commutative operator on inherited attributes, and $\lesssim$ such that G is monotone and $\oplus$ is decreasing w.r.t. $\lesssim$. If $(G, \oplus)$ terminates on a DAG g with result r, then G terminates on $\mathcal{U}(g)$ with result r' such that $r \lesssim r'$.*

Note that due to the symmetry of Theorem 2, we also know that if $\oplus$ is increasing, i.e. $x, y \lesssim x \oplus y$, then we have that $r' \lesssim r$. We obtain this corollary by simply considering the inverse of $\lesssim$.

Let's see how the above theorem applies to the type inference AG. The order $\lesssim$ on Env is the usual order on partial mappings (i.e. the subset order on the DAG of partial mappings); $\lesssim$ on $Maybe\ Type$ is the least quasi-order with $Nothing \lesssim t$ for all t :: $Maybe\ Type$; and $\lesssim$ on $Set\ Name$ is the subset order. With these orders all semantic functions are monotone, and the operator $\oplus = intersection$ is decreasing. We thus get the soundness property by applying Theorem 2: if type inference on a DAG g returns r then it returns r' on $\mathcal{U}(g)$ with $r \lesssim r'$. In particular, if $r = Just\ t$ then also $r' = Just\ t$.

The DAG in Figure 6 is an example where $r = Nothing$ and $r' = Just\ t$. The problem is that the variable "x" is shared between two contexts in which it has different types. That is, intersecting the environments yields the empty environment. However, the above phenomenon as well as non-termination can only occur if the DAG is not *well-scoped* in the following sense: a DAG is well-scoped if no variable node is shared among different binders, or shared between a bound and free occurrence. This restriction rules out the DAG in Figure 6 as well as the one in Figure 5c.

Given this well-scopedness property, we can show that type inference on a well-scoped DAG g produces the same result as on its unravelling $\mathcal{U}(g)$ – provided it terminates. It only remains to be shown that whenever the result r on g is $Nothing$, then also the result r' on $\mathcal{U}(g)$ is $Nothing$. The full version of Theorem 2, as we will see in section 5.4, is much stronger than stated above: we have the relation $\lesssim$ between a run on a DAG and a run on its unravelling not only for the final results r and r' but for each node and each attribute. That means if we get a type t for a sub-DAG of g, then we also get the type t for the corresponding subtree of $\mathcal{U}(g)$. Consequently, if we got a type for $\mathcal{U}(g)$ but not for g itself, then the reason could not be a type mismatch. It can only be because a variable was not found in the environment. That, however, can never happen because of well-scopedness. Hence, also $r' = Nothing$.

We still have to deal with the issue of termination, though. Semantically, non-termination means that there may be either no run or multiple different runs on DAGs. While monotonicity does not prove termination in general, it can help us to at least establish the existence of runs:

Proposition 1 (sketch)**.** *Given G, $\oplus$, and $\lesssim$ as in Theorem 2 such that $\lesssim$ is well-founded on inherited attributes, then $(G, \oplus)$ has a run on any DAG.*

Proposition 1 immediately applies to the type inference AG. Thus it remains to be shown that runs are unique. As we have seen in Figure 5c, this is not true in general. However, restricted to well-scoped DAGs it is: if there were two distinct runs on a DAG g, then the runs can only differ on shared nodes, since runs on $\mathcal{U}(g)$ are unique. Moreover, the type attribute depends only on type attributes of child nodes, except in the case of variables. Hence, there must be a variable node to which the two runs assign different types. However, well-scopedness makes this impossible.

Thus, we can conclude that $typeInf_G$ on a well-scoped DAG g behaves as $typeInf$ on its unravelling $\mathcal{U}(g)$.

It is always possible to make a DAG well-scoped by means of alpha-renaming. However, note that renaming on a DAG may lead to duplication. For example, renaming one of the loops in Figure 6 would require introducing a new variable node. As a safe approximation, in particular when using $reifyDag$, making sure that all binders introduce distinct variable names guarantees that the DAG is well-scoped.[1]

Finally, it is important to note that monotonicity is not an intrinsic property of AGs, but depends on the choice of $\lesssim$.[2] In particular, we may choose one order $\lesssim$ for using Theorem 2 and another one for proving termination using Proposition 1.

4.4 Observing the Sharing

In this paper, we have only looked at AGs for which we want to get the same result when running on a DAG and running on its unravelling. That is, we have only cared about DAGs as a compact representation of trees, and we want to get the same result regardless of how the tree is represented.

However, there are cases where we actually want to give meaning to the sharing in the DAG. One such case is when estimating signal delays in digital circuits. The time it takes for an output of a logic gate to switch depends on how many other gates are connected to it – i.e. its *load*. A higher load leads to slower switching.

As a simple example, let us for a while assume that the $IntTreeF$ functor defined in section 3.4 represents digital circuits. $Leaf$ can represent inputs and $Node$ can be a NAND gate (any n-ary Boolean function can be computed by a network of NAND gates).

[1] See `Rename.hs` in the accompanying repository.

[2] For example, any AG is monotone w.r.t the full relation.

```
type Circuit = Dag IntTreeF
```

To implement delay analysis as an AG, we start by defining attributes for delay and load:

```
newtype Delay = Delay Int deriving (Eq, Ord, Num)
newtype Load  = Load  Int deriving (Eq, Ord, Num)
```

The delay attribute can be computed by summing the maximum input delay, some intrinsic gate delay and a load-dependent term:

```
gateDelay :: (Load ∈ as) ⇒ Syn IntTreeF as Delay
gateDelay (Leaf _)     = Delay 0
gateDelay (Node a b) = max (below a) (below b)
                              + Delay 10 + Delay l
  where Load l = above
```

In this simplified delay analysis, we interpret load as the number of connected gates, so the load attribute that is propagated down is 1 for both inputs:

```
gateLoad :: Inh IntTreeF as Load
gateLoad (Node a b) = a ↦ 1 & b ↦ 1
gateLoad _             = ∅
```

The delay analysis is completed by running the AG on a circuit DAG using $(+)$ as the resolution function:

```
delay :: Load → Circuit → Delay
delay l = runAGDag (+) gateDelay gateLoad (λ_ → l)
```

Note that the semantic function for the load attribute does not do any interesting computation. Instead, it is the resolution function that "counts" the number of connected gates for each node.

Since the above AG is monotone and $+$ is increasing w.r.t. the natural order on integers, Theorem 2 gives us the expected result that the delay of a circuit DAG is greater than or equal to the delay of its unravelling.

The circuit description system Wired [4] implements analyses on circuit DAGs using a generic traversal scheme and semantic functions similar to the ones above. It should be possible to give a more principled implementation of these analyses in terms of AGs using monotonicity as a proof principle.

4.5 Transforming and Constructing DAGs

The definition of $repmin_G$ from the beginning of section 4 uses $runAGDag$ to run the repmin AG on DAGs. While $repmin_G$ does take DAGs as input, it produces trees as output. The reason for this is that while the AG is oblivious to whether it runs on a DAG or a tree, it does explicitly construct a tree as its output.

However, there is no reason why it should do so. The only assumption that is made in constructing the synthesised tree attribute is that its values can be combined using the constructors of the underlying functor $IntTreeF$. However, this assumption is true for both the type $Tree\ \ IntTreeF$ and $Dag\ \ IntTreeF$. Indeed, by drawing ideas from macro tree transducers [7, 15] our AG recursion scheme can be generalised to preserve sharing in the result of an AG computation. That is, if applied to trees the AG constructs trees and if applied to a DAG the AG constructs DAGs in its attributes. An important property of this generalised recursion scheme, which we call *parametric AGs*, is that both Theorem 1 and Theorem 2 carry over to this generalisation.

For the sake of demonstration we shall only consider a simple instance of this generalised recursion scheme. For more details we refer the reader to the technical report [6]. The instance of parametric AGs that we consider consists of an ordinary AG together with a simple "rewrite" function, which is used to transform the input DAG. This intuition is encoded in the following type that can be seen as a specialisation of Syn:

```
type Rewrite f as g = ∀ a.(?below :: a → as, ?above :: as)
                                    ⇒ f a → g a
```

The semantic function rep, which defines the $repmin$ transformation, has to be modified only superficially to fit this type:

```
rep′ :: (Min_I ∈ as) ⇒ Rewrite IntTreeF as IntTreeF
rep′ (Leaf i)      = Leaf globMin
rep′ (Node a b) = Node a b
```

Note that the parametric polymorphism of the type $Rewrite$ allows us to instantiate the construction performed by rep' to both trees and DAGs. Apart from this polymorphism, functions of this type are no different from semantic functions for synthesised attributes. Therefore, we can extend the function $runAG$ such that it takes a rewrite function as an additional semantic function:

```
runRewrite :: (Traversable f, Functor g) ⇒
  Syn f (s, i) s → Inh f (s, i) i → Rewrite f (s, i) g →
  (s → i) → Tree f → Tree g
```

The definition of $repmin$ can thus be reformulated as follows:

```
repmin :: Tree IntTreeF → Tree IntTreeF
repmin = runRewrite min_S min_I rep′ init
  where init (Min_S i) = Min_I i
```

The corresponding variant for DAGs, not only takes DAGs as input but also returns DAGs:

```
runRewriteDag :: (Traversable f, Functor g) ⇒
  (i → i → i) → Syn f (s, i) s → Inh f (s, i) i →
  Rewrite f (s, i) g → (s → i) → Dag f → Dag g
```

The definition of $repmin_G$ is adjusted accordingly:

```
repmin_G :: Dag IntTreeF → Dag IntTreeF
repmin_G = runRewriteDag const min_S min_I rep′ init
  where init (Min_S i) = Min_I i
```

Now $repmin_G$ has the desired type – and the implementation of $runRewriteDag$ has the expected property that sharing of the input DAG is preserved. However, $repmin_G$ does not produce the same result for a DAG g as $repmin$ does for $\mathcal{U}(g)$. But it does produce a DAG that unravels to the result of $repmin$, i.e. both are equivalent modulo unravelling. This is an immediate consequence of the corresponding variant of Theorem 1 for AGs with rewrite functions [6]:

Theorem 3 (sketch). *Given a copying rewriting attribute grammar G, a binary operator $\oplus$ on inherited attributes with $x \oplus y \in \{x, y\}$ for all x, y, and a DAG g, we have that G terminates on $\mathcal{U}(g)$ with result t iff $(G, \oplus)$ terminates on g with result h such that $\mathcal{U}(h) = t$.*

The type of $Rewrite$ as given above is unnecessarily restrictive, since it requires that each constructor from the input functor f is replaced by a single constructor from the target functor g. In general, a rewrite function may produce arbitrary layers built from g. This generalisation can be expressed as follows, where $Free\ g$ is the free monad of g:

```
type Rewrite′ f as g = ∀ a.(?below :: a → as, ?above :: as)
                                      ⇒ f a → Free g a
```

5. Semantics

We present the semantics of AGs on trees and DAGs. To keep the presentation simple, we restrict ourselves to a set theoretic semantics. For a formal treatment of parametric AGs as discussed in section 4.5, we refer the reader to the technical report [6].

To give the semantics on DAGs, we have to restrict ourselves to functors that are representable by finitary containers [1]. In the

Haskell implementation, this assumption corresponds to the restriction to functors that are instances of the *Traversable* type class. Traversable functors (that satisfy the appropriate associated laws) are known to be exactly representable by finitary containers [9].

Definition 1. A *finitary container* F is a pair $(\mathsf{Sh}, \mathsf{ar})$ consisting of a set Sh of *shapes*, and an *arity* function $\mathsf{ar}\colon \mathsf{Sh} \to \mathbb{N}$. Each finitary container F gives rise to a functor $\mathsf{Ext}(F)\colon \mathsf{Set} \to \mathsf{Set}$, called the *extension* of F, that maps each set X to the set of (dependent) pairs $(s, \overline{x})$, where $s \in \mathsf{Sh}$ and $\overline{x} \in X^{\mathsf{ar}(s)}$. By abuse of notation we also write F for the functor $\mathsf{Ext}(F)$.

5.1 Trees and DAGs

Analogously to the way trees and DAGs are parametrised by a functor in our Haskell implementation, we parametrise the corresponding semantic notions by a finitary container. In the following, we use the shorthand notation $(s_i)_{i<l}$ for a tuple $(s_0, \dots, s_{l-1}) \in \Pi_{i<l} S_i$.

Definition 2. The set of trees $\mathsf{Tree}(F)$ over a finitary container F is the least fixed point of $\mathsf{Ext}(F)$. That is, each tree t is of the form $(s, (t_i)_{i<l})$ with $t_i \in \mathsf{Tree}(F)$ for all $i < l$. The set $\mathcal{P}(t)$ of *positions* of t is the least set of finite sequences over $\mathbb{N}$ such that $\langle\rangle \in \mathcal{P}(t)$ and if $p \in \mathcal{P}(t_j)$, then $\langle j\rangle \cdot p \in \mathcal{P}(s, (t_i)_{i<l})$. Given a position $p \in \mathcal{P}(t)$, we define the subtree $t|_p$ of t at p as follows: $t|_{\langle\rangle} = t$ and $(s, (t_i)_{i<l})|_{\langle j\rangle \cdot p} = t_j|_p$ for all $j < l$.

For DAGs, we use a representation similar to the Haskell implementation, viz. a mapping from nodes to their child nodes.

Definition 3. A *graph* $g = (N, E, r)$ over a finitary container F is given by a finite set N of nodes, an *edge* function $E\colon N \to F(N)$, and a root node $r \in N$. A graph g induces a *reachability* relation $\overset{g}{\to}$, which is the least transitive relation $\overset{g}{\to}$ such that $n \overset{g}{\to} n_j$, whenever $E(n) = (s, (n_i)_{i<l})$. We write $\overset{g}{\leftarrow}$ for the inverse of $\overset{g}{\to}$. A graph $g = (N, E, r)$ is called a *DAG* if (a) each node $n \in N$ is reachable from r, i.e. $r \overset{g}{\to} n$, and (b) g is *acyclic*, i.e. $\overset{g}{\leftarrow}$ is well-founded. The set of all DAGs over F is denoted $\mathsf{DAG}(F)$. Given a DAG $g = (N, E, r)$ and a node $n \in N$, the *sub-DAG* of g rooted in n, denoted $g|_n$, is the DAG (N', E', n), where $N' = \{m \in N | n \overset{g}{\to} m\}$ is the set of nodes reachable from n in g, and E' is the restriction of E to N'.

Note that as DAGs are finite, $\overset{g}{\to}$ is well-founded iff $\overset{g}{\leftarrow}$ is well-founded. Moreover, each tree $t \in \mathsf{Tree}(F)$ gives rise to a DAG $\mathcal{G}(t) = (\mathcal{P}(t), E, \langle\rangle) \in \mathsf{DAG}(F)$, where

$$E(p) = (s, (p \cdot \langle i\rangle)_{i<l}) \quad \text{if} \quad t|_p = (s, (t_i)_{i<l}).$$

Conversely, each DAG $g = (N, E, r)$ gives rise to a tree $\mathcal{U}(g)$, called the *unravelling* of g, as follows:

$$\mathcal{U}(g) = (s, (\mathcal{U}(g|_{n_i}))_{i<l}) \quad \text{if} \quad E(r) = (s, (n_i)_{i<l})$$

The mapping $\mathcal{U}(\cdot) : \mathsf{DAG}(F) \to \mathsf{Tree}(F)$ is well-defined by the principle of well-founded recursion with the well-founded relation $<$ given by: $g < h$ iff $g = h|_n$ with n a node in h that is not the root. Well-foundedness of $<$ follows from the well-foundedness of the reachability relation $\overset{g}{\leftarrow}$ for each DAG $g \in \mathsf{DAG}(F)$.

Similarly to positions in trees, we define *paths* in a DAG. Given a DAG $g = (N, V, r)$ and node $n \in N$, the set $\mathcal{P}_g(n)$ of paths to n in g is inductively defined as the least set with (a) $\langle\rangle \in \mathcal{P}_g(r)$, and (b) if $p \in \mathcal{P}_g(n)$ and $E(n) = (s, (n_i)_{i<l})$, then $p \cdot \langle i\rangle \in \mathcal{P}_g(n_i)$ for all $i < l$. The set of all paths in a DAG g, denoted $\mathcal{P}(g)$, is then simply the union $\bigcup_{n\in N} \mathcal{P}_g(n)$. This union is a disjoint union, i.e. for each path $p \in \mathcal{P}(g)$, there is a unique node $n \in N$ such that $p \in \mathcal{P}_g(n)$. We denote this unique node n as $g[p]$. We can observe the close relationship between paths and positions in the unravelling of DAGs: we have that $\mathcal{P}(g) = \mathcal{P}(\mathcal{U}(g))$.

5.2 Attribute Grammars and Their Semantics

In the following we will work with families $(D_a)_{a\in I}$ of sets and families $(f_a)_{a\in I}$ of functions $f_a\colon X \to D_a$ defined on them. To work with them conveniently, we make use of the notation D_A, with $A \subseteq I$, for the set $\Pi_{a\in A} D_a$ and f_A for the function of type $X \to D_A$ that maps each $x \in X$ to $(f_a(x))_{a\in A}$.

Definition 4. An *attribute grammar (AG)* $G = (S, I, D, \alpha, \delta)$ over a finitary container $F = (\mathsf{Sh}, \mathsf{ar})$ consists of:

- finite, disjoint sets S, I of *synthesised* resp. *inherited* attributes,
- a family $D = (D_a)_{a\in S\cup I}$ of *attribute domains*,
- a family $\alpha = (\alpha_a\colon D_S \to D_a)_{a\in I}$ of *initialisation functions*,
- a family $\delta = (\delta_a)_{a\in S\cup I}$ of *semantic functions*, where

$$\delta_a\colon F(D_S) \times D_I \to D_a \quad \text{if } a \in S$$
$$\delta_a\colon \Pi_{((s,\overline{d}),d)\in F(D_S)\times D_I} D_a^{\mathsf{ar}(s)} \quad \text{if } a \in I$$

In other words, δ_a maps each $((s, \overline{d}), d) \in F(D_S) \times D_I$ to some $e \in D_a$ if $a \in S$ and to some $\overline{e} \in D_a^{\mathsf{ar}(s)}$ if $a \in I$.

The semantics of an AG is defined in terms of runs on a tree or a DAG. A run is simply a decoration of all nodes in the tree/DAG with elements of the attribute domains that is consistent with the semantic and initialisation functions.

Definition 5. Let $G = (S, I, D, \delta, \alpha)$ be an AG on F and $t \in \mathsf{Tree}(F)$. A family $\rho = (\rho_a)_{a\in S\cup I}$ of mappings $\rho_a\colon \mathcal{P}(t) \to D_a$ is called a *run* of G on t if the following conditions are met:

- $\alpha_a(\rho_S(\langle\rangle)) = \rho_a(\langle\rangle)$ for all $a \in I$
- For each $p \in \mathcal{P}(t)$ with $t|_p = (s, (t_i)_{i<l})$, we have that

$$\delta_a((s, (\rho_S(p\cdot\langle i\rangle))_{i<l}), \rho_I(p)) = \begin{cases} \rho_a(p) & \text{if } a \in S \\ (\rho_a(p \cdot \langle i\rangle))_{i<l} & \text{if } a \in I \end{cases}$$

If there is a unique run ρ, we obtain the result $\rho_S(\langle\rangle) \in D_S$, which we denote by $[\![G]\!](t)$.

For the semantic function δ_a of an inherited attribute a, we use the notation $\delta_{a,j}$ for the function that returns the j-th component of the result of δ_a. For example, we can reformulate the condition on ρ_a from the above definition as follows:

$$\delta_{a,j}((s, (\rho_S(p \cdot \langle i\rangle))_{i<l}), \rho_I(p)) = \rho_a(p \cdot \langle j\rangle) \quad \text{for all } j < l$$

In general an AG may have multiple runs or no run at all. However, we can give sufficient conditions on AGs that ensure that a given AG has exactly one run on any tree. One such condition is that the semantic functions have no cyclic dependencies, which is known as *non-circularity* in the literature on AGs.

We will not go into the details of deciding non-circularity and instead refer to the algorithm of Knuth [25, 26]. An important consequence of non-circularity is that we can schedule the construction of the unique run of the AG on an input tree. In particular, given a tree $t \in \mathsf{Tree}(F)$ and AG $G = (S, I, D, \delta, \alpha)$ on F, there is a well-founded order $<$ on the set $(S \cup I) \times \mathcal{P}(t)$, which describes in which order the run of G on t can be constructed. In the following, when we say that an AG is non-circular, we assume that such a well-founded order exists for any input tree.

The definition of a run on DAGs is more difficult as a node in a DAG may have multiple parents, which leads to the situation depicted in Figure 4, where a node may receive several inherited attribute values. Our approach in this paper is to assume, for each inherited attribute a, a binary operator $\oplus_a$ that combines attribute values. In order to obtain well-defined notion of a run, we must in general assume that $\oplus_a$ is associative and commutative, i.e. it does not matter in which order inherited attributes are combined:

Definition 6. Let $G = (S, I, D, \alpha, \delta)$ be an AG on F, $\oplus = (\oplus_a : D_a \times D_a \to D_a)_{a \in I}$ a family of associative and commutative binary operators, and $g = (N, E, r) \in \mathsf{DAG}(F)$. A family $\rho = (\rho_a)_{a \in S \cup I}$ of mappings $\rho_a : N \to D_a$ is called a *run* of G modulo $\oplus$ on g if the following conditions are met:

- $\rho_a(r) = \alpha_a(\rho_S(r))$ for all $a \in I$
- For all $n \in N$ with $E(n) = (s, (n_i)_{i<l})$ and $a \in S$, we have
$$\rho_a(n) = \delta_a((s, (\rho_S(n_i))_{i<l}), \rho_I(n))$$
- For all $n \in N$ and $a \in I$, we have
$$\rho_a(n) = \bigoplus_{(m,j,s,(n_i)_{i<l}) \in M} \delta_{a,j}((s, (\rho_S(n_i))_{i<l}), \rho_I(m))$$
– where M is the set of all tuples $(m, j, s, (n_i)_{i<l})$ such that $E(m) = (s, (n_i)_{i<l})$ and $n_j = n$, and the sum is w.r.t. $\oplus_a$.

If there is a unique run ρ, we obtain the result $\rho_S(r) \in D_S$, which we denote by $(\!|G, \oplus|\!)(g)$.

Note that the definition of runs on DAGs generalises the definition of runs on trees in the sense that a run on a tree t is also a run on the corresponding DAG $\mathcal{G}(t)$ and vice versa.

In the following three sections, we shall formally state and prove the correspondence theorems that we used in section 4.

5.3 Copying Attribute Grammars

At first we consider the case of copying AGs, i.e. AGs whose semantic functions for all inherited attributes simply copy the value of the attribute from each node to all its child nodes:

Definition 7. An AG $G = (S, I, D, \alpha, \delta)$ over F is called *copying*, if $\delta_{a,j}((s, \overline{d}), (e_b)_{b \in I}) = e_a$ for all $a \in I$, $(s, \overline{d}) \in F(D_S)$, $j < \mathsf{ar}(s)$ and $(e_b)_{b \in I} \in D_I$. A family $(\oplus_a : D_a \times D_a \to D_a)_{a \in I}$ of binary operators is called *copying* if $d \oplus_a e \in \{d, e\}$ for all $a \in I$ and $d, e \in D_a$.

Given such a setting as described above, we can show that, for each run of an AG on a DAG g, we find an equivalent run of the AG on $\mathcal{U}(g)$, and vice versa. Equivalence of runs is defined as follows: given an AG $G = (S, I, D, \alpha, \delta)$ over F, we say that a run ρ of G on a DAG $g \in \mathsf{DAG}(F)$ and a run ρ' of G on $\mathcal{U}(g)$ are *equivalent* if $\rho'_a(p) = \rho_a(g[p])$ for all $a \in S \cup I$ and $p \in \mathcal{P}(g)$.

Theorem 1. *Given a copying AG $G = (S, I, D, \alpha, \delta)$ over F, a copying $\oplus = (\oplus_a : D_a \times D_a \to D_a)_{a \in I}$, and a DAG $g = (N, E, r) \in \mathsf{DAG}(F)$, we have that for each run of G modulo $\oplus$ on g there is an equivalent run of G on $\mathcal{U}(g)$, and vice versa.*

Proof sketch. Given a run ρ on g, we construct ρ' on $\mathcal{U}(g)$ by setting $\rho'_a(p) = \rho_a(g[p])$. Conversely, given a run ρ on $\mathcal{U}(g)$, we construct a run ρ' on g by setting $\rho'_a(n) = \rho_a(p)$ for some $p \in \mathcal{P}_g(n)$. This is well-defined since ρ_a is constant for $a \in I$, and for $a \in S$, we have $\rho_a(p) = \rho_a(q)$ whenever $\mathcal{U}(g)|_p = \mathcal{U}(g)|_q$. ☐

Corollary 1. *Given G, $\oplus$, and g as in Theorem 1 such that G is non-circular, we have that $(\!|G, \oplus|\!)(g) = [\![G]\!](\mathcal{U}(g))$.*

Note that for copying AGs we do not need $\oplus$ to be commutative and associative to obtain a well-defined semantics on DAGs – as long as $\oplus$ is copying, too.

5.4 Correspondence by Monotonicity

Next we show that if the attribute domains D_a of an AG G are quasi-ordered such that the semantic and initialisation functions are monotone and $\oplus_a$ are decreasing, then the result of any run of G on a DAG g is less than or equal to the result of the run of G on $\mathcal{U}(g)$. We start by making the preconditions of this theorem explicit:

Definition 8. A family of binary operators $(\oplus_a : D_a \times D_a \to D_a)_{a \in A}$ on a family of quasi-ordered sets $(D_a, \lesssim_a)_{a \in A}$ is called *decreasing* if $d_1 \oplus_a d_2 \lesssim d_1, d_2$ for all $a \in A$ and $d_1, d_2 \in D_a$. A function $f : S \to T$ between two quasi-ordered sets $(S, \lesssim_S)$ and $(T, \lesssim_T)$ is called *monotone* if $s_1 \lesssim_S s_2$ implies $f(s_1) \lesssim_T f(s_2)$ for all $s_1, s_2 \in S$. An AG $G = (S, I, D, \alpha, \delta)$ equipped with a quasi-order $\lesssim_a$ on D_a for each $a \in S \cup I$, is called *monotone* if each α_a and δ_a is monotone, where the orders on D_S, $F(D_S) \times D_I$ and D_S^n are defined pointwise according to $(\lesssim_a)_{a \in S \cup I}$. That is, e.g. $\lesssim_A$ on D_A is defined by $(d_a)_{a \in A} \lesssim_A (e_a)_{a \in A}$ iff $d_a \lesssim_a e_a$ for all $a \in A$, and $\lesssim$ on $F(D_S) \times D_I$ is defined by $((s, (d_i)_{i<k}), d) \lesssim ((t, (e_i)_{i<l}), e)$ iff $s = t$, $d_i \lesssim_S e_i$ for all $i < l$ and $d \lesssim_I e$.

Theorem 2. *Let $G = (S, I, D, \alpha, \delta)$ be a non-circular AG, $\oplus = (\oplus_a : D_a \times D_a \to D_a)_{a \in S \cup I}$ associative and commutative operators, and $(\lesssim_a)_{a \in S \cup I}$ quasi-orders such that G is monotone and $\oplus$ is decreasing w.r.t. $(\lesssim_a)_{a \in S \cup I}$. Given a run ρ of G modulo $\oplus$ on a DAG $g = (N, E, r)$ and the run ρ' of G on $\mathcal{U}(g)$, we have $\rho_a(g[p]) \lesssim_a \rho'_a(p)$ for all $a \in S \cup I$ and $p \in \mathcal{P}(g)$.*

Proof sketch. Since G is non-circular, there is a well-founded order $<$ on $(S \cup I) \times \mathcal{P}(\mathcal{U}(g))$ compatible with G. The above inequation can then be shown by well-founded induction using $<$. ☐

Corollary 2. *Given G, $\oplus$, $(\lesssim_a)_{a \in S \cup I}$, and g as in Theorem 2, and given that $(\!|G, \oplus|\!)(g)$ is defined, then $(\!|G, \oplus|\!)(g) \lesssim_S [\![G]\!](\mathcal{U}(g))$.*

Note that while we assume non-circularity of the AG – as in Corollary 1 – $(\!|G, \oplus|\!)(g)$ may not be defined – unlike in Corollary 1. Nonetheless, for the proof of Theorem 2 the assumption of non-circularity is essential since it is the basis of the induction argument. The issue of non-termination of AGs on DAGs was discussed in section 4.2 exemplified with the DAG depicted in Figure 6.

Nevertheless, in case the AG is monotone w.r.t. well-founded orders, we can at least prove the existence of runs on DAGs:

Proposition 1. *Given G, $\oplus$, and $(\lesssim_a)_{a \in S \cup I}$ as in Theorem 2 such that $\lesssim_a$ is well-founded for every $a \in I$, then, on any DAG there is a run of G modulo $\oplus$.*

6. Related Work

Graph Representations The immediate practical applicability of our recursion schemes is based on Gill's idea of turning the implicit sharing information in a Haskell expression into an explicit graph representation [20]; thus making sharing visible. The twist of our work is, however, that we provide recursion schemes that are – from the outside – oblivious to sharing but – under the hood – exploit the sharing information for efficiency.

Oliveira and Cook [32] introduced a purely functional representation of graphs, called *structured graphs*, using Chlipala's *parametric higher-order abstract syntax* [12]. The recursion scheme that Oliveira and Cook use is a fold generalised to (cyclic) graphs. For a number of specialised instances, e.g. *map* on binary trees and *fold* on streams, the authors provide laws for equational reasoning. Oliveira and Löh [33] generalised structured graphs to indexed data structures with particular focus on EDSLs. While AGs could be implemented as a fold on structured graphs, doing so would incur a performance penalty due to recomputation as soon as inherited attributes are used. Moreover, the indirect representation of sharing in structured graphs hinders a direct efficient implementation of AGs.

The Lightweight Modular Staging framework, by Rompf and Odersky [36], allows its internal graph representation to be traversed through a tree-like interface, and the implementation takes care of the administration of avoiding duplication in the generated code for shared nodes. However, as far as we can tell, there is no support for using the tree interface to write algorithms such as our type inference, which avoids duplicated computations when shared nodes are used in different contexts.

Buneman et al. [10] introduce a language UnQL for querying graph-structured data. Queries are based on structural recursion, which means that the user can view the data as a tree, regardless of the underlying representation (which can even be cyclic). The motivation behind UnQL is similar to ours; however, UnQL does not appear to support propagation and merging of accumulating parameters (inherited attributes) in recursive functions.

Tree and Graph Automata There is a strong relationship between tree automata and attribute grammars, where bottom-up acceptors correspond to synthesised attributes and top-down acceptors correspond to inherited attributes. The difference is that automata are used to characterise tree languages and devise decision procedures, i.e. the automaton itself is the object of interest rather than the results of its computations. Our notion of rewriting attribute grammars is derived from tree transducers [19], i.e. tree automata that characterise tree transformations, and our representation of them in Haskell is based on Hasuo et al. [21]. Our representation of AGs in Haskell is directly taken from Bahr's *modular tree automata* [5], which are in turn derived from representations of tree automata based on the work of Hasuo et al. [21].

While a number of generalisations of tree automata to graphs have been studied, a unified notion of graph automata remains elusive [35]. There are only specialised notions of graph automata for particular applications, and our notion of AGs on DAGs falls into this category as well. There are some automata models that come close to our approach. However, they either cause recomputation in case of conflicting top-down state (instead of providing a resolution operator $\oplus$) [18, 30], restrict themselves to bottom-up state propagation only [3, 11, 16], or assume that the in-degree of nodes is fixed for each node label [23, 34]. Either approach is too restrictive for the application we have demonstrated in this paper. Moreover, none of these automata models allow for interdependency between bottom-up and top-down state.

Kobayashi et al. [27] consider a much more general form of compact tree representations than just DAGs: programs that produce trees. The authors study and implement tree transducers on such compact tree representations. To this end, they consider generalised finite state transformations (GFSTs) [14], which subsume both bottom-up and top-down transducers. However, GFSTs only provide top-down state propagation. Bottom-up state propagation has to be encoded inefficiently and is restricted to finite state spaces.

Attribute Grammars Viera et al. [39] were the first to give an embedding of AGs in Haskell that allows the programmer to combine semantic functions to construct AGs in a modular fashion. They do not rely on a specific representation of trees as we do, but instead make heavy use of Template Haskell in order to derive the necessary infrastructure. As a result, their approach is applicable to a wider variety of data types. At the same time, however, this approach excludes transparent execution of thus defined AGs on graph structures.

The idea to utilise the structure of attributes that happen to be tree-structured – as our parametric AGs from section 4.5 do – also appears in the literature on AGs, albeit with a different motivation: so-called *higher-order attribute grammars* [40] permit the execution of the AG nested within those tree-structured attributes. By composing parametric AGs sequentially similarly to the composition of tree transducers [19], we can achieve the same effect.

Higher-order attribute grammars implicitly introduce sharing when duplicating higher-order attributes. Saraiva et al. [38] exploit this sharing for their implementation of incremental attribute evaluation. Their goal, however, is different from ours: the sharing structure makes equality tests, which are necessary for incremental evaluation, cheaper and increases cache hits.

Data Flow Analysis Despite the difference in their application, there is some similarity between our correspondence theorems for simple AGs and the soundness results for data flow analysis (DFA) [2]. In particular, variants of Theorem 2 also appear in the literature on DFA. In the context of DFA, these soundness results are formulated as follows: the maximum fixpoint (MFP) is bounded by the meet over all paths (MOP). The MFP roughly corresponds to the run of an AG on a DAG, whereas the individual paths in the MOP correspond to the run of an AG on a tree. However, there are a number of differences.

First of all we only consider acyclic graphs, whereas DFA typically considers cyclic graphs. As a consequence, there are stronger requirements for DFA, in particular, the ordering has to have finite height. Secondly, AGs perform bidirectional computations, whereas DFA typically only considers unidirectional problems, i.e. either forward or backwards analyses. There are DFA frameworks that do support bidirectional analyses, however, they come with additional restrictions, e.g. separability [24].

The differences become more pronounced if we consider the parametric AGs outlined in section 4.5, which allow us to implement sharing-preserving graph transformations. The closest analogue in the DFA literature is an approach that interleaves unidirectional DFA with transformation steps [28]. However, we are not aware of a DFA framework that combines bidirectional analyses with graph transformations.

7. Discussion and Future Work

We have presented a technique that allows us to represent trees as compact DAGs, yet process them as if they were trees. The distinguishing feature of our approach is that it avoids recomputation for shared nodes even in the case of interdependent bottom-up and top-down propagation of information. This approach is supported by complementing correspondence theorems to prove the soundness of the shift from trees to DAGs. In particular, correspondence by monotonicity (Theorem 2) provides a widely applicable proof principle since it is parametric in the quasi-order. We have presented three examples for which correspondence by monotonicity gives useful results: $\mathit{leavesBelow}$ (cf. [6]), $\mathit{typeInf}$ and $\mathit{gateDelay}$. The $\mathit{typeInf}$ algorithm follows a general pattern for simple syntax-directed analyses for which monotonicity gives strong correspondence properties. Another similar example, size inference, is given in the file `Size.hs` in the accompanying repository.

A difficult obstacle in this endeavour is ensuring termination of the resulting graph traversals. As we have shown, for some instances, such as type inference, termination can only be guaranteed if further assumptions are made on the structure of the input DAG. A priority for future work is to find more general principles that allow us to reason about termination on a higher level analogous to the correspondence theorems we presented. We already made some progress in this direction as Theorem 1, Theorem 3 and, to a limited degree, Proposition 1 allowed us to infer termination of graph traversals. A potential direction for improvement is a stricter notion of non-circularity that guarantees termination of AGs on DAGs. A simple approximation of this could be for example a coarser notion of dependency: if an attribute a depends on attribute b, then b may not depend on a. The resulting notion of non-circularity would for example prove that the AG corresponding to $\mathit{leavesBelow}$ from the introduction terminates on DAGs.

Another direction for future work is to extend the expressive power of our recursion scheme:

- Extend AGs with fixpoint iteration [17, 31, 37] to deal with cyclic graphs and to implement analyses based on abstract interpretation.
- Support a wider class of data types, e.g. mutually recursive data types and GADTs. Both should be possible using well-known techniques from the literature [22, 41].
- Support deep pattern matching in AGs. This can be done by extending the AG and $Rewrite$ type with a parameter that can partially uncover nested subtrees. Deep patterns would make it easier to express e.g. rewrite rules in a compiler.

The motivation behind this work is to make traversals over trees with sharing more efficient. In the accompanying technical report [6] we present results from our measurements on a set of benchmarks. In summary, the benchmarks show that running algorithms on DAGs is asymptotically more efficient than running on trees when the DAG has a lot of sharing. They also show that the overhead of running on DAGs when there is no sharing is less than $2\times$ for trees of size 2^{16}, both for AGs and the generalised AGs presented in section 4.5. To achieve this, our implementation works on a more efficient representation of DAGs that uses explicit pointers only if necessary, i.e. only for sharing. The essential idea is to replace f inside the definition of Dag with $Free\ f$, where $Free$ is the free monad construction. That is, we interleave the tree and the graph representation. As a consequence, the representation of DAGs without sharing is essentially a tree. This fact can be exploited by using the tree implementation of AGs for DAGs without sharing. The upshot of this implementation is that there is no overhead of running AGs on DAGs without sharing.

Acknowledgements The first author is funded by the Danish Council for Independent Research, Grant 12-132365. The second author is funded by the Swedish Foundation for Strategic Research, under grant RAWFP.

References

[1] M. Abbott, T. Altenkirch, and N. Ghani. Categories of containers. In *FoSSaCS*, 2003.

[2] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986. ISBN 0-201-10088-6.

[3] S. Anantharaman, P. Narendran, and M. Rusinowitch. Closure properties and decision problems of dag automata. *Inf. Process. Lett.*, 94(5): 231 – 240, 2005.

[4] E. Axelsson. *Functional Programming Enabling Flexible Hardware Design at Low Levels of Abstraction*. PhD thesis, Chalmers University of Technology, 2008.

[5] P. Bahr. Modular tree automata. In *MPC*, 2012.

[6] P. Bahr and E. Axelsson. Generalising Tree Traversals to DAGs: Exploiting Sharing without the Pain. Available from authors' web site, 2014. Technical report with full proofs.

[7] P. Bahr and L. E. Day. Programming macro tree transducers. In *WGP*, 2013.

[8] R. Bird. Using circular programs to eliminate multiple traversals of data. *Acta Inform.*, 21(3):239–250, 1984.

[9] R. Bird, J. Gibbons, S. Mehner, J. Voigtländer, and T. Schrijvers. Understanding idiomatic traversals backwards and forwards. In *Haskell*, 2013.

[10] P. Buneman, M. Fernandez, and D. Suciu. UnQL: a query language and algebra for semistructured data based on structural recursion. *The VLDB Journal*, 9(1):76–110, 2000.

[11] W. Charatonik. Automata on DAG representations of finite trees. Research report, Max-Planck-Institut für Informatik, March 1999.

[12] A. Chlipala. Parametric higher-order abstract syntax for mechanized semantics. In *ICFP*, 2008.

[13] R. A. Eisenberg, D. Vytiniotis, S. Peyton Jones, and S. Weirich. Closed type families with overlapping equations. In *POPL*, 2014.

[14] J. Engelfriet. Bottom-up and top-down tree transformations — a comparison. *Mathematical systems theory*, 9(2):198–231, 1975.

[15] J. Engelfriet and H. Vogler. Macro tree transducers. *J. Comput. System Sci.*, 31(1):71–146, 1985.

[16] B. Fila and S. Anantharaman. Running tree automata on trees and/or dags. Technical report, LIFO, 2006.

[17] J. Fokker and S. D. Swierstra. Abstract interpretation of functional programs using an attribute grammar system. In *LDTA*, 2009.

[18] A. Fujiyoshi. Recognition of directed acyclic graphs by spanning tree automata. *Theor. Comput. Sci.*, 411(38–39):3493 – 3506, 2010.

[19] Z. Fülöp and H. Vogler. *Syntax-Directed Semantics: Formal Models Based on Tree Transducers*. Springer-Verlag New York, Inc., 1998.

[20] A. Gill. Type-safe observable sharing in Haskell. In *Haskell*, 2009.

[21] I. Hasuo, B. Jacobs, and T. Uustalu. Categorical views on computations on trees (extended abstract). In *ICALP*, 2007.

[22] P. Johann and N. Ghani. Foundations for structured programming with GADTs. In *POPL*, 2008.

[23] T. Kamimura and G. Slutzki. Transductions of dags and trees. *Math. Syst. Theory*, 15(1):225–249, 1981.

[24] U. P. Khedker and D. M. Dhamdhere. A generalized theory of bit vector data flow analysis. *ACM Trans. Program. Lang. Syst.*, 16(5): 1472–1511, 1994.

[25] D. Knuth. Semantics of context-free languages: Correction. *Math. Syst. Theory*, 5(2):95–96, 1971.

[26] D. E. Knuth. Semantics of context-free languages. *Theory Comput. Syst.*, 2(2):127–145, 1968.

[27] N. Kobayashi, K. Matsuda, A. Shinohara, and K. Yaguchi. Functional programs as compressed data. *Higher-Order and Symbolic Computation*, pages 1–46, 2013.

[28] S. Lerner, D. Grove, and C. Chambers. Composing dataflow analyses and transformations. In *POPL*, pages 270–282, 2002.

[29] J. R. Lewis, J. Launchbury, E. Meijer, and M. B. Shields. Implicit parameters: dynamic scoping with static types. In *POPL*, 2000.

[30] M. Lohrey and S. Maneth. Tree automata and XPath on compressed trees. In *CIAA*, 2006.

[31] A. Middelkoop. *Inference with Attribute Grammars*. PhD thesis, Universiteit Utrecht, Feb. 2012.

[32] B. C. Oliveira and W. R. Cook. Functional programming with structured graphs. In *ICFP*, 2012.

[33] B. C. d. S. Oliveira and A. Löh. Abstract syntax graphs for domain specific languages. In *PEPM*, 2013.

[34] D. Quernheim and K. Knight. Dagger: A toolkit for automata on directed acyclic graphs. In *FSMNLP*, 2012.

[35] J.-C. Raoult. Problem #70: Design a notion of automata for graphs, 2005. URL `http://rtaloop.mancoosi.univ-paris-diderot.fr/problems/70.html`. The RTA list of open problems.

[36] T. Rompf and M. Odersky. Lightweight modular staging: A pragmatic approach to runtime code generation and compiled DSLs. In *GPCE*, 2010.

[37] M. Rosendahl. Abstract interpretation using attribute grammars. In *WAGA*, 1990.

[38] J. Saraiva, D. Swierstra, and M. Kuiper. Functional incremental attribute evaluation. In *Compiler Construction*, 2000.

[39] M. Viera, S. D. Swierstra, and W. Swierstra. Attribute grammars fly first-class. In *ICFP*, 2009.

[40] H. H. Vogt, S. D. Swierstra, and M. F. Kuiper. Higher order attribute grammars. In *PLDI*, 1989.

[41] A. R. Yakushev, S. Holdermans, A. Löh, and J. Jeuring. Generic programming with fixed points for mutually recursive datatypes. In *ICFP*, 2009.