Science of Computer Programming ••• (••••) •••–•••

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Generalising tree traversals and tree transformations to DAGs: Exploiting sharing without the pain

Patrick Bahr [a,*], Emil Axelsson [b]

[a] IT University of Copenhagen, Denmark
[b] Department of Computer Science and Engineering, Chalmers University of Technology, Sweden

ARTICLE INFO

Article history:
Received 30 June 2015
Received in revised form 25 January 2016
Accepted 15 March 2016
Available online xxxx

Keywords:
Attribute grammars
Sharing
Graph traversal
Graph transformation
Directed acyclic graphs

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 or a tree transformation and then apply it to compact graph representations of trees instead. The resulting graph traversal or graph transformation avoids 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.

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:

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

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 t has a shared node a that

* Corresponding author.
E-mail addresses: paba@itu.dk (P. Bahr), emax@chalmers.se (E. Axelsson).

http://dx.doi.org/10.1016/j.scico.2016.03.006
0167-6423/

appears twice:

$$t = \textbf{let } a = \mathit{Node}\ (\mathit{Node}\ (\mathit{Leaf}\ 2)\ (\mathit{Leaf}\ 3))\ (\mathit{Leaf}\ 4)\ \textbf{in } \mathit{Node}\ a\ a$$

Unfortunately, a function like *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 [54]. However, such a change of representation would force us to express *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 *leavesBelow* as it is, and be able to run it on both trees and graphs.

Oliveira and Cook [54] 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, *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 [6]. 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 *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 practically relevant computations over trees including program analyses and program transformations.

The paper makes the following contributions:

- We present an implementation of AGs in Haskell, which allows us to write tree traversals such that they can be applied to compact DAG representations of trees as well.
- We generalise AGs to parametric AGs in order to implement complex tree transformations that preserve sharing if applied to DAGs.
- We prove a number of general correspondence theorems that relate the semantics of (parametric) 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 and manipulations, for instance in a compiler. We demonstrate this fact by implementing type inference and a size-based simplifier for a simple functional language.

The rest of the paper is organised as follows: Section 2 presents the running example – a simple expression language and a type inference algorithm for it. 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 comparing the semantics of AGs on trees with the corresponding semantics of AGs on DAGs. Section 6 introduces a generalisation of AGs called *parametric attribute grammars* with which we can transparently express transformations of trees and graphs. Section 7 presents an extended example of our technique – a simplifier for a simple functional language. Section 8 gives the implementation of AGs on DAGs, and evaluates the performance of different implementations. The theory for this paper is developed in sections 5 and 6.3. Readers that are not interested in the details of the theory may safely skip these sections as the theorems that are developed there are reproduced in simplified form in the rest of the paper. Some proofs were elided or abridged to save space. The full proofs are presented in the accompanying technical report [10]. Likewise, the paper does not give the exact implementation of all recursion schemes. The missing parts are available in an accompanying repository [8].

This paper extends and improves a previous paper that appeared in the proceedings of PEPM 2015 [9]. In particular, the present paper includes additional examples to illustrate the application of attribute grammars on DAGs; it introduces the more general notion of parametric attribute grammars, which unifies the theory; it presents a new implementation of our framework based on a hybrid representation of DAGs; and it provides benchmark results that illustrate the benefit of our technique.

2. Running example

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

Please cite this article in press as: P. Bahr, E. Axelsson, Generalising tree traversals and tree transformations to DAGs: Exploiting sharing without the pain, Sci. Comput. Program. (2016), http://dx.doi.org/10.1016/j.scico.2016.03.006

```
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 $n, s, b :: Exp$) 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 body *b*. For example, the Haskell expression *iter* 5 1 $(\lambda s \to s + 2)$ 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 Fig. 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.

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 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:

```
data Tree f = In (f (Tree f))
```

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

```
data ExpF a = LitB Bool | LitI Int | Var Name | Eq a a | Add a a | If a a a | Iter Name a a a
```

Please cite this article in press as: P. Bahr, E. Axelsson, Generalising tree traversals and tree transformations to DAGs: Exploiting sharing without the pain, Sci. Comput. Program. (2016), http://dx.doi.org/10.1016/j.scico.2016.03.006

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

typeInf :: Env Type → 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 a → Env a → Env a
insertEnv = Map.insert

lookEnv :: Name → Env a → Maybe a
lookEnv v = join.Map.lookup v
```

Fig. 1. Type inference for example EDSL.

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

Fig. 2. Propagation of attribute values by an attribute grammar.

Apart from requiring functors such as *ExpF* to be instances of *Functor*, we also require them to be instances of the *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 *typeInf* as an AG, it would consist of an inherited attribute that is the environment and a synthesised attribute that is the inferred type.

Fig. 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 Fig. 2b.

Please cite this article in press as: P. Bahr, E. Axelsson, Generalising tree traversals and tree transformations to DAGs: Exploiting sharing without the pain, Sci. Comput. Program. (2016), http://dx.doi.org/10.1016/j.scico.2016.03.006

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:

```
type Algebra f c = f c → c
fold :: Functor f ⇒ Algebra f c → Tree f → c
fold alg (In t) = alg (fmap (fold alg) t)
```

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:

```
class c ∈ as where
  pr :: as → c
```

Intuitively, $c \in as$ means that c is a component of as, and it provides the corresponding projection function. We can give instance declarations accordingly, which gives us for example that $a \in (a, b)$ with the projection function defined as $pr = \lambda(x, y) \to x$. Using closed type families [22], the type class $\in$ can be defined such that it works on arbitrarily nested product types, but disallows ambiguous instances such as $Int \in (Int, (Bool, Int))$ for which multiple projections exist. But there are also simpler implementations of $\in$ that only use type classes [6].

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

```
type Syn f as s = (s ∈ as) ⇒ as → f as → 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. Moreover, we can assume that as at least contains s.

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

```
typeInf_S :: Syn ExpF as (Maybe Type)
typeInf_S _ (Add a b)
  | Just IntType ← pr a
  , Just IntType ← pr b = Just IntType

...
```

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 parametric AGs. It is based on the isomorphism below, which follows from the Yoneda Lemma for all functors f and types as, s:

$$(\forall c.(c \to as) \to (f\ c \to s)) \cong f\ as \to s$$

It allows us to define the type $Syn\ f\ as\ s$ alternatively like this:

$$\forall\ c.(s \in as) \Rightarrow as \to (c \to as) \to f\ c \to s$$

This representation corresponds to Mendler-style folds [65].

The benefit of this Mendler-style representation is that it provides an *extensible* interface for the abstract type c of child nodes. For now, this interface only has one operation, namely a function of type $c \to as$ that returns the attribute values associated to a given child node. Later we shall extend this interface so that we can also assign inherited attribute values to child nodes or use a child node to construct a tree or a DAG.

We further transform the above type by turning the first two arguments of type as and $c \to as$ into implicit parameters [46], which provides an interface closer to that of AG systems:

```
type Syn f as s = ∀ c.(?below :: c → as, ?above :: as, s ∈ as) ⇒ f c → s
```

The implicit parameters *?below* and *?above* provide access to the attribute values at the children and the current node, respectively. Combining the implicit parameters with projection gives us two convenient helper functions for writing semantic functions:

Please cite this article in press as: P. Bahr, E. Axelsson, Generalising tree traversals and tree transformations to DAGs: Exploiting sharing without the pain, Sci. Comput. Program. (2016), http://dx.doi.org/10.1016/j.scico.2016.03.006

```
typeInf_S :: (Env Type ∈ 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 Type)
typeInf_I (Iter v n i b) = b ↦ insertEnv v ti above
                           where ti = typeOf i
typeInf_I _              = ∅
```

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

```
above :: (?above :: as, i ∈ as) ⇒ i
above = pr (?above)

below :: (?below :: a → as, s ∈ as) ⇒ a → s
below a = pr (?below a)
```

These functions pick out a specific attribute from the compound type *as* of all attributes. Typically, *above* is used to access an inherited attribute (propagated from the parent) and *below* to access a synthesised attribute (propagated from a child). But it is not unusual to use *above* for synthesised attributes (propagated to the parent) and *below* for inherited attributes (propagated to the children).

The complete definition of the synthesised type attribute for type inference is given in Fig. 3. The function $typeInf_I$ is the semantic function for the inherited environment attribute. It will be explained in the following subsection. The code uses a convenient helper function for querying the synthesised type of a sub-expression:

```
typeOf :: (?below :: c → as, Maybe Type ∈ as) ⇒ c → 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. Concretely, given a node of type $f\ c$, where type c represents child positions of the node, we assign inherited attribute values of type i to each such child position. This can be achieved, for example, by a finite mapping of type $Map\ c\ i$. This would give us the following representation of semantic functions for inherited attributes:

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

However, instead of choosing a concrete representation of the mapping of child positions to attribute values, such as *Map*, we rather want to give an abstract interface. This will also enable us to provide an efficient implementations of inherited attributes tailored to the specific use cases. In our definition of *Inh* below, $m\ i$ is an abstract type that represents finite

Please cite this article in press as: P. Bahr, E. Axelsson, Generalising tree traversals and tree transformations to DAGs: Exploiting sharing without the pain, Sci. Comput. Program. (2016), http://dx.doi.org/10.1016/j.scico.2016.03.006

```
class Traversable m ⇒ Mapping m k | m → k where
  -- operators to construct mappings
(&)  :: m v → m v → m v
(↦) :: k → v → m v
∅    :: m v
  -- methods for the internal implementation
prodMapWith    :: (v1 → v2 → v) → v1 → v2 → m v1 → m v2 → m v
findWithDefault :: a → k → m a → a
```

Fig. 4. Definition of finite mappings.

mappings from c to i, which is expressed by the type constraint *Mapping m c*:

$$\textbf{type}\ Inh\ f\ as\ i = \forall\ m\ c.(?below :: c \to as, ?above :: as, i \in as, Mapping\ m\ c) \Rightarrow f\ c \to m\ i$$

The type class *Mapping* describes the interface that finite mappings provide. Its definition is given in Fig. 4. For now, only the first three methods are of interest: The two infix operators $\mapsto$ and & allow us to construct singleton mappings $x \mapsto y$ and construct the union $m \,\&\, n$ of two mappings. The constant $\emptyset$ denotes the empty mapping.

The definition of *Inh* given above does not ensure that the returned mapping is complete, i.e. that each position 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 explicit assignment is made in the mapping of the semantic function.

The semantic function for the inherited environment attribute of type inference is given by $typeInf_I$ in Fig. 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 the variable binding 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:

$$(\otimes) :: (s_1 \in as, s_2 \in as) \Rightarrow Syn\ f\ as\ s_1 \to Syn\ f\ as\ s_2 \to Syn\ f\ as\ (s_1, s_2)$$
$$(s_1 \otimes s_2)\ t = (s_1\ t, s_2\ t)$$

The implementation for inherited attributes is more difficult as we have to honour the copy rule. That is, given two semantic functions i_1 and i_2, where i_1 assigns an attribute value for a given child position but i_2 does not, the product of i_1 and i_2 must assign an attribute value consisting of the value given by i_1 and a copy for the second attribute. To this end, we use the *prodMapWith* method provided by the *Mapping* type class.

$$(\circledast) :: (Functor\ f, i_1 \in as, i_2 \in as) \Rightarrow Inh\ f\ as\ i_1 \to Inh\ f\ as\ i_2 \to Inh\ f\ as\ (i_1, i_2)$$
$$(i_1 \circledast i_2)\ t = prodMapWith\ (\lambda x\ y \to (x, y))\ above\ above\ (i_1\ t)\ (i_2\ t)$$

The first argument to *prodMapWith* is the function that is used to combine values in the two mappings, in this case pairing. The next two arguments are the default values that are to be used in case only one of the two mappings contains a value for a given child position. By passing *above* as the argument here, this implementation honours the copy rule.

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:

$$runAG :: Traversable\ f \Rightarrow Syn'\ f\ (s, i)\ s \to Inh'\ f\ (s, i)\ i \to (s \to i) \to Tree\ f \to s$$
$$\textbf{type}\ Syn'\ f\ as\ s = \forall\ c.\quad (?below :: c \to as, ?above :: as) \Rightarrow f\ c \to s$$
$$\textbf{type}\ Inh'\ f\ as\ i\ = \forall\ m\ c.(?below :: c \to as, ?above :: as, Mapping\ m\ c) \Rightarrow f\ c \to m\ i$$

Please cite this article in press as: P. Bahr, E. Axelsson, Generalising tree traversals and tree transformations to DAGs: Exploiting sharing without the pain, Sci. Comput. Program. (2016), http://dx.doi.org/10.1016/j.scico.2016.03.006

The types Syn' and Inh' are like Syn and Inh but without the constraints of the form $\ldots \in as$. Those constraints are not needed here because *runAG* operates on the full attribute space. Yet it is possible to pass functions of type Syn and Inh to *runAG* as we will see in the following example, which defines type inference using *runAG*.

We define type inference as a run of the AG defined in Fig. 3:

$$\begin{aligned}&typeInf :: Env\ Type \to Tree\ ExpF \to Maybe\ Type\\&typeInf\ env = runAG\ typeInf_S\ typeInf_I\ (\lambda\_ \to env)\end{aligned}$$

In this example, the initialisation function for the inherited attribute is simply a constant function that returns the environment. In section 3.5, we shall see an example that uses the full power of the initialisation function.

The translation of the type inference algorithm into an AG is required to use our framework. However, type checking algorithms are an excellent class of examples that can benefit from AGs beyond the issue of sharing. Bidirectional type checking [57] has become increasingly popular for implementing type checking for advanced type systems. A bidirectional type checking algorithm combines type inference and type checking, i.e. it switches between synthesising type information and checking type information. AGs provide a convenient framework to implement bidirectional type checking algorithms: synthesised attributes are used to infer types (as in the example above) and inherited attributes are used to check types.

3.4. Example: leavesBelow

As another example of how to define an AG, we consider the function *leavesBelow* from the introduction. The first step is to define a functor corresponding to the type of integer trees:

$$\begin{aligned}&\textbf{data}\ IntTreeF\ a = Leaf\ Int \mid Node\ a\ a\\&\quad \textbf{deriving}\ (Functor, Foldable, Traversable)\end{aligned}$$

The inherited attribute is an integer that corresponds to the accumulated parameter in *leavesBelow*, i.e. it gives the depth at which we should start collecting the leaves:

$$\begin{aligned}&leavesBelow_I :: Inh\ IntTreeF\ as\ Int\\&leavesBelow_I\ (Leaf\ i) \quad = \emptyset\\&leavesBelow_I\ (Node\ t_1\ t_2) = t_1 \mapsto d' \,\&\, t_2 \mapsto d'\\&\quad \textbf{where}\ d' = above - 1\end{aligned}$$

In the first case, there are no children, so we return the empty mapping. In the *Node* case, the inherited attribute is decreased by one before being passed on to the children.

The synthesised attribute is the set of leaves at and below the depth given by the inherited attribute:

$$\begin{aligned}&leavesBelow_S :: (Int \in as) \Rightarrow Syn\ IntTreeF\ as\ (Set\ Int)\\&leavesBelow_S\ (Leaf\ i)\\&\quad \mid (above :: Int) \leqslant 0 \quad = Set.singleton\ i\\&\quad \mid otherwise \quad = Set.empty\\&leavesBelow_S\ (Node\ t_1\ t_2) = below\ t_1 \cup below\ t_2\end{aligned}$$

In the *Leaf* case, we check whether the leaf should be collected by querying the inherited attribute, and in the *Node* case, we simply join the set of leaves from the children.

The two semantic functions can be combined and run using *runAG*:

$$\begin{aligned}&leavesBelow :: Int \to Tree\ IntTreeF \to Set\ Int\\&leavesBelow\ d = runAG\ leavesBelow_S\ leavesBelow_I\ (\lambda\_ \to d)\end{aligned}$$

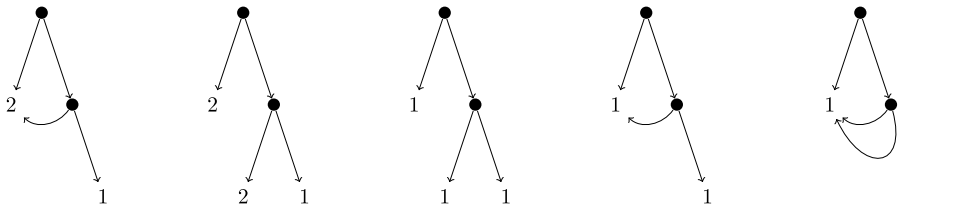
3.5. Example: Richard Bird's repmin

A classic example of a tree traversal with interesting information flow is Bird's *repmin* problem [14]. The problem is the following: 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. For example, applied to the tree in Fig. 5b, we obtain the tree in Fig. 5c. Bird shows how this transformation can be implemented by a single traversal in a lazy functional language.

To code *repmin* as an AG, we introduce two attribute types:

$$\begin{aligned}&\textbf{newtype}\ Min_S = Min_S\ Int\ \textbf{deriving}\ (Eq, Ord)\\&\textbf{newtype}\ Min_I = Min_I\ Int\end{aligned}$$

Please cite this article in press as: P. Bahr, E. Axelsson, Generalising tree traversals and tree transformations to DAGs: Exploiting sharing without the pain, Sci. Comput. Program. (2016), http://dx.doi.org/10.1016/j.scico.2016.03.006



(a) Input DAG. (b) Unravelling. (c) repmin as AG. (d) repmin as RAG. (e) repmin as PAG.

Fig. 5. Input and output of different versions of repmin.

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

```
globMin :: (?above :: as, Min_I ∈ as) ⇒ Int
globMin = let Min_I i = above in i
```

The semantic function for the Min_S attribute is as follows:

```
min_S :: Syn IntTreeF as Min_S
min_S (Leaf i)    = Min_S i
min_S (Node a b) = min (below a) (below b)
```

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

```
min_I :: Inh IntTreeF as Min_I
min_I _ = ∅
```

Finally, we need to be able to synthesise a new tree that depends on the globally smallest integer available from the Min_I attribute. To do so we define a synthesised attribute of type *Tree IntTreeF* computed by the following semantic function:

```
rep :: (Min_I ∈ as) ⇒ Syn IntTreeF as (Tree IntTreeF)
rep (Leaf i)    = In (Leaf globMin)
rep (Node a b) = In (Node (below a) (below b))
```

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

```
repmin :: Tree IntTreeF → Tree IntTreeF
repmin = snd ∘ runAG (min_S ⊗ rep) min_I init
  where init (Min_S i, _) = Min_I i
```

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

Using Haskell's lazy semantics, *runAG* computes the attributes of the given AG by a single traversal of the input tree. Therefore, like Bird's implementation of *repmin*, our implementation of *repmin* above also only traverses the input tree once.

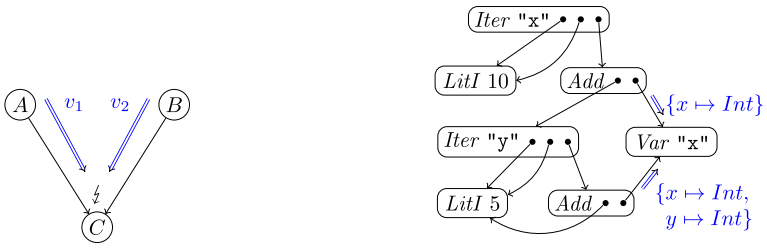
3.6. Informal semantics

Instead of reproducing the implementation of *runAG* here, we shall informally describe the semantics of an AG and describe how *runAG* implements this semantics. The formal semantics and its implementation in Haskell is given later in sections 5 and 8, respectively.

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 $\rho_a : N \to D_a$ 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 may 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.

Fig. 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.

Please cite this article in press as: P. Bahr, E. Axelsson, Generalising tree traversals and tree transformations to DAGs: Exploiting sharing without the pain, Sci. Comput. Program. (2016), http://dx.doi.org/10.1016/j.scico.2016.03.006



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

Fig. 6. Confluence of inherited attributes.

4. Attribute grammars on DAGs

Our goal is to take algorithms intended to work on trees and apply them – 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 3.6 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. Fig. 6a 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 [54]. 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 3.5. However, for the type inference AG, this is clearly not the case. One example that shows the problem is the DAG in Fig. 6b, 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 Fig. 6a, 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 Fig. 6b, forming the intersection of the two environments of the node *Var* `"x"` yields the environment $\{x \mapsto Int\}$.

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

```
runAG_G :: Traversable f ⇒ (i → i → i) →
  Syn f (s, i) s → Inh f (s, i) i → (s → i) → Dag f → s
```

The interface differs in two points from *runAG*: (1) it takes DAGs as input and (2) it takes a binary operator of type $i \to i \to i$, which is 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:

```
typeInf_G :: Env Type → Dag ExpF → Maybe Type
typeInf_G env = runAG_G intersection typeInf_S typeInf_I (λ_ → env)
```

We defer the discussion of the formal semantics of AGs on DAGs as well as the implementation of $runAG_G$ until sections 5 and 8, respectively. But we briefly explain how DAGs of type *Dag f* are represented. We represent DAGs with explicit nodes and edges, with nodes represented by integers:

```
type Node = Int
```

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

```
data Dag f = Dag { root  :: Node,
                 , edges :: IntMap (f Node) }
```

Please cite this article in press as: P. Bahr, E. Axelsson, Generalising tree traversals and tree transformations to DAGs: Exploiting sharing without the pain, Sci. Comput. Program. (2016), http://dx.doi.org/10.1016/j.scico.2016.03.006

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. Moreover, finite mappings are represented by the type *IntMap*, which is Haskell's implementation of PATRICIA trees [53].

Following Gill [32], we provide a function that observes the implicit sharing of a tree of type *Tree f* and turns it into a DAG of type *Dag f*:

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

As a final example, we turn the *repmin* function from section 3.5 into a function $repmin_G$ that works on DAGs.

$$\begin{aligned} &repmin_G :: Dag\ IntTreeF \to Tree\ IntTreeF \\ &repmin_G = snd \circ runAG_G\ const\ (min_S \otimes rep)\ min_I\ init \\ &\quad \textbf{where}\ init\ (Min_S\ i, \_) = Min_I\ i \end{aligned}$$

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. For instance, applied to the DAG pictured in Fig. 5a, $repmin_G$ produces the tree in Fig. 5c. This is the same result we would obtain if we first unravelled the DAG to a tree (pictured in Fig. 5b) and then applied *repmin*. In section 6 we will introduce a generalisation of AGs that can preserve the sharing present in the input DAG and thus produces the DAG pictured in Fig. 5d. Beyond that we will also be able to make use of the nature of the transformation to introduce additional sharing in the result (pictured in Fig. 5e).

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, for example, is 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.*

In terms of our Haskell implementation, Theorem 1 can be read as follows: given an initialisation function $init :: s \to i$ and semantic functions $inh :: Inh\ f\ i$ and $syn :: Syn\ f\ s$ such that *inh* returns $\emptyset$ for all inputs, we have the following for all $t :: Tree\ f$:

$$runAG\ syn\ inh\ init\ t = runAG_G\ const\ syn\ inh\ init\ g$$

where g is the DAG obtained by applying *reifyDag* to t.

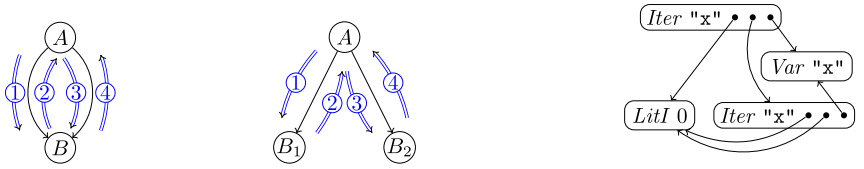
In particular, we can immediately apply Theorem 1 to the repmin AG. We obtain that $repmin_G$ applied to a DAG g yields the same result as *repmin* applied to $\mathcal{U}(g)$. That is, we get the same result for *repmin t* and $fmap\ repmin_G\ (reifyDag\ t)$.

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 [42,41] introduced AGs, he gave an algorithm to check for circular dependencies and proved that AGs terminate in the absence of such circularity.

Please cite this article in press as: P. Bahr, E. Axelsson, Generalising tree traversals and tree transformations to DAGs: Exploiting sharing without the pain, Sci. Comput. Program. (2016), http://dx.doi.org/10.1016/j.scico.2016.03.006



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

Fig. 7. Cyclic dependency in non-circular AG on DAGs.

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 Fig. 7. Fig. 7a shows a simple DAG consisting of two nodes, and Fig. 7b 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 Fig. 7a: 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 Fig. 7c 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 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. We do not have a simple equality relation as we have for a copying AG. In fact, it should be expected that type inference on a DAG g is more restrictive than on its unravelling $\mathcal{U}(g)$: 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 greater than A, also the result of f applied to B is greater 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 graph of partial mappings, and $\lesssim$ on *Maybe Type* is the least quasi-order with *Nothing* $\lesssim t$ for all $t ::$ *Maybe Type*. According to 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.

Please cite this article in press as: P. Bahr, E. Axelsson, Generalising tree traversals and tree transformations to DAGs: Exploiting sharing without the pain, Sci. Comput. Program. (2016), http://dx.doi.org/10.1016/j.scico.2016.03.006

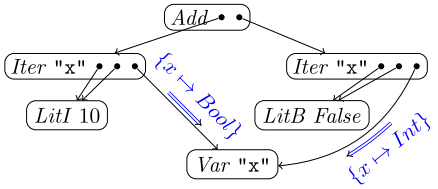


Fig. 8. DAG that is not well-scoped.

As alluded to above, in general, we cannot hope to obtain a completeness property for type inference on DAGs. However, given a mild restriction, we can, in fact, obtain completeness. The DAG in Fig. 8 provides a counterexample for completeness: the result for the DAG g is *Nothing*, while for $\mathcal{U}(g)$ it is *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 Fig. 8 as well as the one in Fig. 7c.

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 attribute at each node. 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 had 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 $r' = \mathit{Nothing}$.

Finally, we show that $\mathit{typeInf}_G$ terminates. Semantically, non-termination means that there is 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, given that the underlying order is a well-well founded order on inherited attributes. A partial order $\leq$ is called well-founded if there is no infinite sequence of elements that is strictly decreasing w.r.t. $\leq$, that is, there is no infinite sequence $(e_i)_{i\in\mathbb{N}}$ with $e_{i+1} < e_i$ for all $i \in \mathbb{N}$.

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

Proposition 2 immediately applies to the type inference AG. Thus it remains to be shown that runs are unique. As we have seen in Fig. 7c, 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 $\mathit{typeInf}_G$ 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 Fig. 8 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 2.

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* represents inputs and *Node* represents NAND gates (any n-ary Boolean function can be computed by a network of NAND gates).

[1] See the `Dag.Rename` module in the accompanying repository.

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

Please cite this article in press as: P. Bahr, E. Axelsson, Generalising tree traversals and tree transformations to DAGs: Exploiting sharing without the pain, Sci. Comput. Program. (2016), http://dx.doi.org/10.1016/j.scico.2016.03.006

```
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 = runAG_G (+) 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 AG defined by the above semantic functions 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.

5. Semantics

With the goal of keeping the presentation simple, we give the semantics of AGs in a set theoretic setting. Moreover, in order to be able to formulate a semantics of AGs 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 those that are representable by finitary containers [15].

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 (*Traversable*) 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 $(s_0, \ldots, s_{l-1}) \in \Pi_{i<l} S_i$. Moreover, we use the notation $\langle n_1, \ldots, n_l \rangle$ for finite sequences over $\mathbb{N}$, i.e. in particular, $\langle\rangle$ denotes the empty sequence; and we use the binary operator $\cdot$ to denote the concatenation of finite sequences.

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 a tree 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 the formal definition of DAGs, we use a representation similar to the Haskell implementation, viz. a mapping from nodes to their child nodes.

Please cite this article in press as: P. Bahr, E. Axelsson, Generalising tree traversals and tree transformations to DAGs: Exploiting sharing without the pain, Sci. Comput. Program. (2016), http://dx.doi.org/10.1016/j.scico.2016.03.006

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 $\xrightarrow{g}$, which is the least transitive relation such that $n \xrightarrow{g} n_j$, whenever $E(n) = (s, (n_i)_{i<l})$. We write $\xleftarrow{g}$ for the inverse of $\xrightarrow{g}$. A graph $g = (N, E, r)$ is called a *DAG* if (a) each node $n \in N$ is reachable from r, i.e. $r \xrightarrow{g} n$, and (b) g is *acyclic*, i.e. $\xleftarrow{g}$ 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 $N' = \{m \in N | n \xrightarrow{g} m\}$ is the set of nodes reachable from n in g, and E' is the restriction of E to N'.

Recall that a strict partial order $<$ is called well-founded iff there is no infinite sequence of elements $(e_i)_{i\in\mathbb{N}}$ that is decreasing w.r.t. $<$, i.e. $e_{i+1} < e_i$ for all $i \in \mathbb{N}$. Since DAGs are finite, $\xrightarrow{g}$ is well-founded iff $\xleftarrow{g}$ is well-founded. Moreover, each tree $t \in \mathsf{Tree}(F)$ gives rise to a DAG $\mathcal{G}(t) \in \mathsf{DAG}(F)$, given by the triple $(\mathcal{P}(t), E, \langle\rangle)$, 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 the following tree $\mathcal{U}(g)$, called the *unravelling* of g:

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

The mapping $\mathcal{U}(\cdot)\colon \mathsf{DAG}(F) \to \mathsf{Tree}(F)$ 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 $\xleftarrow{g}$ for each DAG $g \in \mathsf{DAG}(F)$.

Similarly to positions in trees, we define *paths* in a DAG. Given a DAG $g = (N, E, r)$ 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 over a finitary container $F = (\mathsf{Sh}, \mathsf{ar})$ is a tuple $(S, I, D, \alpha, \delta)$ consisting of:

- finite, disjoint sets S, I of *synthesised* resp. *inherited* attributes,
- a family $D = (D_a)_{a\in S\cup I}$ of sets, called *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

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

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 resp. 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 over 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 $[\![G]\!](t)$.

Please cite this article in press as: P. Bahr, E. Axelsson, Generalising tree traversals and tree transformations to DAGs: Exploiting sharing without the pain, Sci. Comput. Program. (2016), http://dx.doi.org/10.1016/j.scico.2016.03.006

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 [42,41]. 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. For example, if $(a, \langle 0 \rangle) < (b, \langle\rangle)$ then the attribute b of the root of the tree can only be computed after the attribute a of the first child of the root has been computed.

Below we give the properties that the order $<$ satisfies. For each $p \in \mathcal{P}(t)$ with $t|_p = (s, (t_i)_{i<l})$, we have the following:

- For all $a \in S$ and $b \in I$, we have $(a, \langle\rangle) < (b, \langle\rangle)$ or α_b is independent of a.
- For all $a \in S$, $b \in I$, and $i, j < l$, we have $(a, p \cdot \langle i \rangle) < (b, p \cdot \langle j \rangle)$ or $\delta_{b,j}((s, \cdot), \cdot)$ is independent of (a, i).
- For all $a, b \in I$, and $j < l$, we have $(a, p) < (b, p \cdot \langle j \rangle)$ or $\delta_{b,j}((s, \cdot), \cdot)$ is independent of a.
- For all $a, b \in S$, and $i < l$, we have $(a, p \cdot \langle i \rangle) < (b, p)$ or $\delta_b((s, \cdot), \cdot)$ is independent of (a, i).
- For all $a \in I$ and $b \in S$, we have $(a, p) < (b, p)$ or $\delta_b((s, \cdot), \cdot)$ is independent of a.

We say that α_b is *independent* of a if $\alpha_a((e_c)_{c \in S})$ has the same value for each $e_a \in D_a$ and we say that a function $f((s, \cdot), \cdot) \colon D_S^l \times D_I \to M$ is *independent* of (a, j) or a if $f((s, (d_i)_{i<l}), (e_b)_{b \in I})$ with $d_i = (d_{i,b})_{b \in S}$ has the same value for all $d_{j,a} \in D_a$, respectively, for all $e_a \in D_a$.

In the following, when we say that an AG is non-circular, we assume that a well-founded order as described above exists for any input tree.

Proposition 1. *Every non-circular AG has a unique run on any given 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 Fig. 6, 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 over F, $\oplus = (\oplus_a \colon D_a \times D_a \to D_a)_{a \in I}$ 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 $\rho_a \colon N \to D_a$ 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 two 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:

Please cite this article in press as: P. Bahr, E. Axelsson, Generalising tree traversals and tree transformations to DAGs: Exploiting sharing without the pain, Sci. Comput. Program. (2016), http://dx.doi.org/10.1016/j.scico.2016.03.006

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 $a \in I$, $(s, \overline{d}) \in F(D_S)$, $j < \mathrm{ar}(s)$ and $(e_b)_{b\in I} \in D_I$. A family $(\oplus_a \colon D_a \times D_a \to D_a)_{a\in I}$ 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 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 $\oplus = (\oplus_a \colon D_a \times D_a \to D_a)_{a\in I}$, 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 $(\!|G, \oplus|\!)(g) = [\![G]\!](\mathcal{U}(g))$.*

Proof. Due to the correspondence of runs according to Theorem 1, uniqueness of runs of G on DAGs follows from the uniqueness of its runs on trees (cf. Proposition 1). Hence, both $(\!|G, \oplus|\!)(g)$ and $[\![G]\!](\mathcal{U}(g))$ are defined, and by Theorem 1 they are equal. □

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 $(\oplus_a \colon D_a \times D_a \to D_a)_{a\in A}$ 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 $f \colon S \to T$ 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, $\oplus = (\oplus_a \colon D_a \times D_a \to D_a)_{a\in S\cup I}$ 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 $(\!|G, \oplus|\!)(g) \lesssim_S [\![G]\!](\mathcal{U}(g))$.*

Proof. Given the unique runs ρ and ρ' on g and $\mathcal{U}(g)$, respectively, we have the following according to Theorem 2:

$$(\!|G, \oplus|\!)(g) = \rho_S(r) = \rho_S(g[\langle\rangle]) \lesssim_S \rho'_S(\langle\rangle) = [\![G]\!](\mathcal{U}(g)) \qquad \square$$

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 Fig. 8.

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 2. *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 $\oplus$.*

Please cite this article in press as: P. Bahr, E. Axelsson, Generalising tree traversals and tree transformations to DAGs: Exploiting sharing without the pain, Sci. Comput. Program. (2016), http://dx.doi.org/10.1016/j.scico.2016.03.006

6. Transforming and constructing DAGs

The definition of $repmin_G$ from the beginning of section 4 uses $runAG_G$ 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 [25,11] 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 is that both Theorem 1 and Theorem 2 can be generalised to cover it, too.

For the sake of demonstration we first consider a simple special case that is often sufficient to express transformations of DAGs. This special case is represented by *rewriting attribute grammars* (or *RAGs* for short), which provide an additional semantic function that allows us to rewrite the input tree respectively DAG. The more general case is covered later in sections 6.2 and 6.3.

6.1. Special case: simple rewriting

Rewriting attribute grammars (RAGs) extend AGs 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*:

$$\textbf{type}\ Rewrite\ f\ as\ g = \forall\ c.(?below :: c \to as, ?above :: as) \Rightarrow f\ c \to g\ c$$

The difference between *Rewrite* and *Syn* is that the latter may produce values of an arbitrary type s, whereas the former produces values of type $g\ c$, where c is the type of child nodes. Intuitively, each node – represented as element of the type $f\ c$ – is rewritten to a new node of type $g\ c$. In this representation, child nodes – i.e. elements of type c – are not only used to reference other attribute values via *above* and *below*, but also to define how newly constructed nodes are connected to other nodes.

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

$$\begin{aligned} &rep' :: (Min_I \in as) \Rightarrow Rewrite\ IntTreeF\ as\ IntTreeF \\ &rep'\ (Leaf\ i) \quad\ = Leaf\ globMin \\ &rep'\ (Node\ a\ b) = Node\ a\ b \end{aligned}$$

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:

$$\begin{aligned} runRewrite :: (Traversable\ f, Functor\ g) \Rightarrow\ & Syn\ f\ (s,i)\ s \to Inh\ f\ (s,i)\ i \to Rewrite\ f\ (s,i)\ g \\ & \to (s \to i) \to Tree\ f \to Tree\ g \end{aligned}$$

The definition of *repmin* can thus be reformulated:

$$\begin{aligned} &repmin :: Tree\ IntTreeF \to Tree\ IntTreeF \\ &repmin = runRewrite\ min_S\ min_I\ rep'\ init \\ &\quad \textbf{where}\ init\ (Min_S\ i) = Min_I\ i \end{aligned}$$

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

$$\begin{aligned} runRewrite_G :: (Traversable\ f, Functor\ g) \Rightarrow\ & (i \to i \to i) \to Syn\ f\ (s,i)\ s \to Inh\ f\ (s,i)\ i \\ & \to Rewrite\ f\ (s,i)\ g \to (s \to i) \to Dag\ f \to Dag\ g \end{aligned}$$

The definition of $repmin_G$ is adjusted accordingly:

$$\begin{aligned} &repmin_G :: Dag\ IntTreeF \to Dag\ IntTreeF \\ &repmin_G = runRewrite_G\ const\ min_S\ min_I\ rep'\ init \\ &\quad \textbf{where}\ init\ (Min_S\ i) = Min_I\ i \end{aligned}$$

Now $repmin_G$ has the desired type – and the implementation of $runRewrite_G$ has the expected property that sharing of the input DAG is preserved. For example, $repmin_G$ transforms the DAG in Fig. 5a into the DAG in Fig. 5d. However, $repmin_G$

Please cite this article in press as: P. Bahr, E. Axelsson, Generalising tree traversals and tree transformations to DAGs: Exploiting sharing without the pain, Sci. Comput. Program. (2016), http://dx.doi.org/10.1016/j.scico.2016.03.006

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 rewriting AGs:

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 above theorem is an instance of the more general Theorem 4 for parametric AGs, which we shall discuss in more detail in section 6.3.

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 arbitrarily many 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 = ∀ c.(?below :: c → as, ?above :: as) ⇒ f c → Free g c
data Free f a = In (f (Free f a))
              | Ret a
```

The implementation of *runRewrite* and $runRewrite_G$ can be changed to accommodate this more general definition of *Rewrite*. In section 7, we shall look at an extended example that uses this more general version of *Rewrite* to implement a simplifier for a simple functional language.

Note that it is possible to reimplement *Tree* approximately in terms of *Free* by ruling out the use of the *Ret* constructor:

```
type Tree f = Free f Zero
data Zero   -- empty type
```

This implementation of *Tree* will be used throughout the rest of the paper.

6.2. Parametric attribute grammars

The rewriting AGs from the previous section introduced a new type of semantic function – represented by the type *Rewrite* – that describes how to transform an input tree or DAG. While this rewrite pattern fits many applications, it is also quite limited in at least two ways: (a) only a single rewrite transformation is performed, and (b) rewrites are propagated bottom–up only.

Relaxing both restrictions opens new applications that are essential for program transformations. What we want to achieve is that tree/DAG transformations as facilitated by *Rewriting* can be performed and referred to in the same flexible fashion as synthesised and inherited attribute.

For instance if we wish to implement *loop-invariant code motion*, i.e. moving code outside of a loop for performance optimisation, we need to propagate several transformations upwards: (1) a set of code fragments that we want to hoist out of loops, and (2) the final optimised program. A simple example that requires bottom–up and top–down propagation of transformations is inlining: code fragments that need to be inlined have to be propagated top–down, while the resulting transformed program is constructed bottom–up.

In this section we present *Parametric AGs* (or *PAGs* for short) as a solution to this problem. PAGs generalise ordinary AGs by incorporating the idea of RAGs into both synthesised and inherited attributes. In particular, instead of 'just' types, attribute domains become functors in PAGs. Semantic functions for these attribute will then become parametrically polymorphic functions – hence the name parametric AGs. As for RAGs, the parametric polymorphism will allow us to instantiate these semantic functions for both trees and DAGs. In the case for DAGs, this will mean that synthesised and inherited attributes will allow us to propagate DAGs (or DAG fragments) upwards respectively downwards during the computation.

Since we are generalising AGs, we need to redefine some basic concepts. To clarify where this occurs, we underline the redefined concepts. For example, we write $\underline{Syn}$ instead of *Syn* and $\underline{\in}$ instead of $\in$.

The best way to illustrate the representation of PAGs in Haskell is by contrasting it with the representation of ordinary AGs. In the following we give the definition of *Syn* for AGs, and right below it the corresponding definition for PAGs:

```
type Syn f as s     = ∀ c.   (?below :: c → as,   ?above :: as,   s ∈ as) ⇒ f c → s
type Syn f as s g = ∀ c n.(?below :: c → as n, ?above :: as n, s ∈ as) ⇒ f c → s (Free g n)
```

Note that we have chosen to name concepts in PAGs the same as the corresponding concepts in AGs – even though they may have different types respectively kinds. In particular, $\underline{\in}$ is now a binary type class over types of kind $* \to *$ instead of $*$.

Please cite this article in press as: P. Bahr, E. Axelsson, Generalising tree traversals and tree transformations to DAGs: Exploiting sharing without the pain, Sci. Comput. Program. (2016), http://dx.doi.org/10.1016/j.scico.2016.03.006

The first change in the above definition is that $\underline{Syn}$ now takes an additional argument $g :: * \to *$. This argument serves a purpose similar to the argument g of *Rewrite*: it is the functor describing the target tree or DAG data structure. Secondly, attributes are now of kind $* \to *$, and the function is parametric in the argument n that is passed to the attribute types. The idea is that this type n represents the nodes in the tree/DAG data structure. The parametric polymorphism ensures that the only thing we can do is shuffle nodes around; we cannot inspect them. In particular, we can use these nodes to construct new trees or DAGs and store them in the synthesised attribute that the semantic function computes. The occurrence of $Free\ g\ n$ in the codomain enables this.

The same intuition applies to the generalisation of the type *Inh* for PAG:

$$\begin{aligned}
&\textbf{type}\ Inh\ f\ as\ i \quad = \forall\, m\ c.\ \ (?below :: c \to as,\quad ?above :: as,\quad i \in as, Mapping\ m\ c)\\
&\qquad\qquad\qquad\qquad \Rightarrow f\ c \to m\ i\\
&\textbf{type}\ \underline{Inh}\ f\ as\ i\ g = \forall\, m\ c\ n.(?below :: c \to as\ n, ?above :: as\ n, i \underline{\in} as, Mapping\ m\ c)\\
&\qquad\qquad\qquad\qquad \Rightarrow f\ c \to m\ (i\ (Free\ g\ n))
\end{aligned}$$

Similarly to ordinary AGs, we also allow an initialisation function to initialise inherited attributes. However, in accordance with the generalisation provided by PAGs, this initialisation function is parametric in the node type n and allows construction of trees:

$$\textbf{type}\ Init\ s\ i\ g = (\forall\, n.s\ n \to i\ (Free\ g\ n))$$

Tying all these components of a PAG together, we obtain the following interface for running a PAG on trees:

$$\begin{aligned}
&runPAG :: \forall\, f\ i\ s\ g\ n.(Traversable\ f, Functor\ g, Functor\ i, Functor\ s)\\
&\quad \Rightarrow \underline{Syn}\ f\ (s :*: i)\ s\ g \to \underline{Inh}\ f\ (s :*: i)\ i\ g \to Init\ s\ i\ g\\
&\quad \to Tree\ f \to s\ (Tree\ g)
\end{aligned}$$

where $:*:$ is the pointwise product on functors. In analogy to the ordinary product type, we use *ffst* and *fsnd* for the first and second projection on $:*:$, respectively.

The interface for running PAGs on DAGs is similar. Like for ordinary AGs, the only addition we need is a conflict resolution function:

$$\begin{aligned}
&runPAG_G :: \forall\, f\ i\ s\ g.(Traversable\ f, Traversable\ g, Traversable\ i, Traversable\ s)\\
&\quad \Rightarrow (\forall\, n.i\ n \to i\ n \to i\ n) \to \underline{Syn}\ f\ (s :*: i)\ s\ g \to \underline{Inh}\ f\ (s :*: i)\ i\ g \to Init\ s\ i\ g\\
&\quad \to Dag\ f \to s\ (Dag\ g)
\end{aligned}$$

To illustrate PAGs on a simple example we reconsider the repmin transformation from section 3.5. In section 6.1, we have used a RAG to implement repmin such that it preserves the sharing of the original input. However, the nature of the repmin transformation provides the opportunity to introduce *additional* sharing: after the repmin transformation, each leaf node has the same label. That means in principle we should be able to produce a DAG that has only a single leaf node instead of many leaf nodes with the same label. Fig. 5e illustrates the desired result DAG. PAGs will allow us to do just that. In the definition of $repmin_G$, we have an inherited attribute, computed by min_I, that propagates the minimum label throughout the DAG. This attribute is then used to relabel all leaf nodes accordingly. With PAGs, we can redefine the inherited attribute such that instead of the minimum label, it contains a node with the minimum labelling. Then, instead of relabelling each leaf node, we can *replace* each leaf node by this single node in the inherited attribute.

To code repmin as a PAG, we first have to change the types of the attributes accordingly:

$$\begin{aligned}
&\textbf{newtype}\ \underline{Min}_S\ n = \underline{Min}_S\ Int\ \textbf{deriving}\ (Functor, Foldable, Traversable)\\
&\textbf{newtype}\ \underline{Min}_I\ n = \underline{Min}_I\ n \quad \textbf{deriving}\ (Functor, Foldable, Traversable)
\end{aligned}$$

The type $\underline{Min}_S$ essentially remains the same, but we have to turn it into a type constructor of kind $* \to *$. More interesting is the type $\underline{Min}_I$: instead of the type *Int* in the original definition, we use the type variable n. Recall that this type variable n represents the nodes in the tree/DAG that we want to construct.

The corresponding semantic functions follow the original definition closely:

$$\begin{aligned}
&\underline{min}_S :: \underline{Syn}\ IntTreeF\ as\ \underline{Min}_S\ f\\
&\underline{min}_S\ (Leaf\ i) \quad = \underline{Min}_S\ i\\
&\underline{min}_S\ (Node\ a\ b) = \underline{Min}_S\ (min\ (unMin_S\ (\underline{below}\ a))\ (unMin_S\ (\underline{below}\ b)))\\
&\underline{min}_I :: \underline{Inh}\ IntTreeF\ as\ \underline{Min}_I\ f\\
&\underline{min}_I\ _ = \emptyset
\end{aligned}$$

Please cite this article in press as: P. Bahr, E. Axelsson, Generalising tree traversals and tree transformations to DAGs: Exploiting sharing without the pain, Sci. Comput. Program. (2016), http://dx.doi.org/10.1016/j.scico.2016.03.006

However, instead of using the overloaded *min* function on the type Min_S, we have to explicitly use the projection function $unMin_S$ that extracts the integer value from Min_S:

$$unMin_S :: \underline{Min}_S\ a \to Int$$
$$unMin_S\ (\underline{Min}_S\ x) = x$$

The function *globMin* to retrieve the Min_I attribute is also changed accordingly:

$$\underline{globMin} :: (?above :: as\ n, \underline{Min}_I \in as) \Rightarrow n$$
$$\underline{globMin} = \mathbf{let}\ \underline{Min}_I\ i = \underline{above}\ \mathbf{in}\ i$$

Instead of an integer *globMin* returns a node.

The actual transformation is achieved similar to the rewrite function we used in section 6.1. Such rewrite functions are simply a special case of a synthesised attribute with the identity functor as domain:

$$\mathbf{data}\ I\ a = I\ \{unI :: a\}\ \mathbf{deriving}\ (Functor, Foldable, Traversable)$$

Therefore, the transformation function $\underline{rep}$ has to use I and unI explicitly:

$$\underline{rep} :: (\underline{Min}_I \in as) \Rightarrow \underline{Syn}\ IntTreeF\ as\ I\ IntTreeF$$
$$\underline{rep}\ (Leaf\ \_) = I\ (Ret\ \underline{globMin})$$
$$\underline{rep}\ (Node\ a\ b) = I\ (In\ ((Node\ (Ret\ (unI\ (\underline{below}\ a))))\ (Ret\ (unI\ (\underline{below}\ b)))))$$

Finally, we can run the thus defined PAG on trees:

$$\underline{repmin} :: Tree\ IntTreeF \to Tree\ IntTreeF$$
$$\underline{repmin} = unI \circ ffst \circ runPAG\ (\underline{rep} \otimes \underline{min}_S)\ \underline{min}_I\ init$$
$$\quad \mathbf{where}\ init\ (\_ :*: \underline{Min}_S\ i) = \underline{Min}_I\ (In\ (Leaf\ i))$$

The important difference between this definition and the definition in section 6.1 is that here the initialisation function *init* constructs the unique leaf node that is used for the whole transformation.

The same PAG can then also be used on DAGs using the $runPAG_G$ function. Like for the AG and the RAG version, we use *const* as the conflict resolution function:

$$\underline{repmin}_G :: Dag\ IntTreeF \to Dag\ IntTreeF$$
$$\underline{repmin}_G = unI \circ ffst \circ runPAG_G\ const\ (\underline{rep} \otimes \underline{min}_S)\ \underline{min}_I\ init$$
$$\quad \mathbf{where}\ init\ (\_ :*: \underline{Min}_S\ i) = \underline{Min}_I\ (In\ (Leaf\ i))$$

Theorems 1 and 2 generalise to the setting of PAGs. First, consider the PAG version of Theorem 1:

Theorem 4 *(Sketch). Given a copying, non-circular PAG 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.*

Note that in contrast to Theorem 1 the above theorem assumes non-circularity.

Applied to the above definitions of $\underline{repmin}$ and $\underline{repmin}_G$, we can conclude that for each DAG g, the unravelling of $\underline{repmin}_G\ g$ is equal to $\underline{repmin}\ g$.

Next, we look at the PAG version of Theorem 2:

Theorem 5 *(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 $\mathcal{U}\ (r) \lesssim r'$.*

The above formulations of Theorems 4 and 5 are somewhat informal. In particular, the subtlety of the role of parametricity is not explicit here: the binary operator $\oplus$ works on the instantiation of the attribute domains to DAGs, whereas $\lesssim$ works on the instantiation to trees. In addition, we use the unravelling operator $\mathcal{U}$ on the result r in Theorem 5, by which we mean that all DAGs 'occurring' in r are unravelled. We will treat these issues more carefully in the next section, which covers the theory of PAGs.

6.3. Semantics

In this section we give the formal semantics of PAGs and generalise the correspondence theorems from section 5 accordingly. Both ordinary AGs and rewriting AGs arise as a special case of this general theory.

Please cite this article in press as: P. Bahr, E. Axelsson, Generalising tree traversals and tree transformations to DAGs: Exploiting sharing without the pain, Sci. Comput. Program. (2016), http://dx.doi.org/10.1016/j.scico.2016.03.006

The idea is to introduce trees in the attributes and in the semantic functions as a type variable that we can instantiate to the type of trees or the type of DAGs. For the semantics, this means that attribute domains are now functors instead of sets, and semantic functions are natural transformations between functors instead of functions between sets.

In the definition below we use products and sums on endofunctors. In addition, given a finitary container F, we write F^* for the free monad of the functor $\mathsf{Ext}(F)$ induced by F. Similar to the notation that we used for families of sets and families of functions, we use the following notation for families $(D_a \colon \mathsf{Set} \to \mathsf{Set})_{a \in I}$ of endofunctors and families $(f_a \colon D \dot{\to} D_a)_{a \in I}$ of natural transformations defined on them: we write D_A, with $A \subseteq I$, for the functor $\Pi_{a \in A} D_a$ and f_A for the natural transformation of type $D \dot{\to} D_A$ that, for each set X, maps each $d \in D(X)$ to $(f_{a,X}(d))_{a \in A}$.

Definition 9. A *parametric attribute grammar (PAG)* $G = (S, I, D, \alpha, \delta)$ from a finitary container $F_1 = (\mathsf{Sh}, \mathsf{ar})$ to a finitary container F_2 consists of:

- finite, disjoint sets S, I of *synthesised* resp. *inherited* attributes,
- a family $D = (D_a \colon \mathsf{Set} \to \mathsf{Set})_{a \in S \cup I}$ of endofunctors (called *attribute domains*),
- a family $\alpha = (\alpha_a \colon D_S \dot{\to} D_a \circ F_2^*)_{a \in I}$ of natural transformations (called *initialisation functions*),
- a family $\delta = (\delta_a)_{a \in S \cup I}$ of *semantic functions*, where

$$\delta_a \colon (F_1 \circ D_S) \times D_I \dot{\to} D_a \circ F_2^* \qquad \text{if } a \in S$$

$$\delta_a \colon (F_1 \circ D_S) \times D_I \dot{\to} \mathsf{FM} \circ D_a \circ F_2^* \text{if } a \in I$$

where FM is the free monoid functor, i.e. for any set X we have that $\mathsf{FM}(X) = \sum_{n \geq 0} X^n$.
Additionally, we require for the case $a \in I$ that

$$\delta_{a,X}((s, x), y) \in (D_a(F_2^*(X)))^{\mathsf{ar}(s)} \quad \text{for all } X, s, x, y.$$

In order to instantiate the semantic functions to trees and DAGs it is not enough to simply instantiate the natural transformations to the corresponding sets $\mathsf{Tree}(F_2)$ and $\mathsf{DAG}(F_2)$. The semantic functions would then have the codomain $D_a(F_2^*(\mathsf{Tree}(F_2)))$ respectively $D_a(F_2^*(\mathsf{DAG}(F_2)))$. Instead, we want to obtain functions with codomain $D_a(\mathsf{Tree}(F_2))$ and $D_a(\mathsf{DAG}(F_2))$, respectively. That is easily achieved by composition with functions of type $F_2^*(\mathsf{Tree}(F_2)) \to \mathsf{Tree}(F_2)$ and $F_2^*(\mathsf{DAG}(F_2)) \to \mathsf{DAG}(F_2)$, respectively. In turn, these functions are given by appropriate F_2-algebras, i.e. functions of type $F_2(\mathsf{Tree}(F_2)) \to \mathsf{Tree}(F_2)$ and $F_2(\mathsf{DAG}(F_2)) \to \mathsf{DAG}(F_2)$, respectively. The basis for this construction is the fact that each F-algebra $a \colon F(X) \to X$ gives rise to an Eilenberg–Moore F^*-algebra $a^* \colon F^*(X) \to X$ (see e.g. Proposition 4.5 in Barr and Wells [12]). An important property of this construction is that it preserves homomorphisms.

Thus, we only need to give algebras of type $F_2(\mathsf{Tree}(F_2)) \to \mathsf{Tree}(F_2)$ and $F_2(\mathsf{DAG}(F_2)) \to \mathsf{DAG}(F_2)$. The former is simply the isomorphism in_{F_2} between $F_2(\mathsf{Tree}(F_2))$ and $\mathsf{Tree}(F_2)$. The latter is a bit more involved. We define, for each finitary container F, the function $gr_F : F(\mathsf{DAG}(F)) \to \mathsf{DAG}(F)$ as follows:

$$\begin{aligned} gr_F(s, (g_i)_{i<l}) &= (N, E, r) \text{ for all } (s, (g_i)_{i<l}) \in F(\mathsf{DAG}(F)) \\ \text{where } g_i &= (N_i, E_i, r_i) \in \mathsf{DAG}(F) \\ N &= \{r\} \uplus \bigcup_{i<l} N_i \\ E(r) &= (s, (r_i)_{i<l}) \\ E(n) &= E_i(n) \quad \text{if } n \in N_i \end{aligned}$$

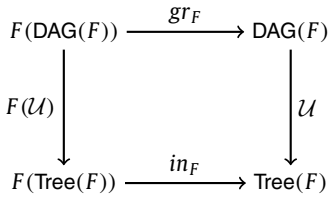
In the above construction, we assume that if $n \in N_i \cap N_j$, then $E_i(n) = E_j(n)$. If this is not the case nodes are renamed accordingly.

Lemma 1. *For every finitary container F, the unravelling operator $\mathcal{U} \colon \mathsf{DAG}(F) \to \mathsf{Tree}(F)$ is a homomorphism between the F^*-algebras gr_F^* and in_F^*, i.e. the following square commutes:*

$$\begin{array}{ccc} F^*(\mathsf{DAG}(F)) & \xrightarrow{gr_F^*} & \mathsf{DAG}(F) \\ \downarrow{\scriptstyle F^*(\mathcal{U})} & & \downarrow{\scriptstyle \mathcal{U}} \\ F^*(\mathsf{Tree}(F)) & \xrightarrow{in_F^*} & \mathsf{Tree}(F) \end{array}$$

Please cite this article in press as: P. Bahr, E. Axelsson, Generalising tree traversals and tree transformations to DAGs: Exploiting sharing without the pain, Sci. Comput. Program. (2016), http://dx.doi.org/10.1016/j.scico.2016.03.006

Proof. Any F-algebra homomorphism from a to b is also an homomorphism from a^* to b^* (see e.g. Proposition 4.5 in Barr and Wells [12]). Thus, to show commutativity of the above diagram it suffices to show commutativity of the following diagram:



$$
\begin{aligned}
& \mathcal{U}\,(gr_F(s,(g_i)_{i<l})) \\
= \quad & \{\ \text{definition of } \mathcal{U}\ \} \\
& (s,(\mathcal{U}\,(gr_F(s,(g_i)_{i<l})|_{r_i}))_{i<l}) \\
= \quad & \{\ \text{definition of } gr_F\ \} \\
& (s,(\mathcal{U}\,(g_i))_{i<l}) \\
= \quad & \{\ in_F \text{ is the identity}\ \} \\
& in_F(s,(\mathcal{U}\,(g_i))_{i<l}) \qquad \square
\end{aligned}
$$

The following two definitions describe the instantiation of PAGs to AGs using the F^*-algebras in_F^* and gr_F^*. The resulting AGs G^T and G^G embody the semantics of the original PAG G for tree and DAGs, respectively.

Definition 10. Let $G = (S, I, D, \alpha, \delta)$ be a PAG from F_1 to F_2. We construct an AG $G^T = (S, I, D^T, \alpha^T, \delta^T)$ by instantiating the attribute domains and semantic functions to $\mathsf{Tree}(F_2)$ as follows:

$$
\begin{array}{lll}
 & D_a^T = D_a(\mathsf{Tree}(F_2)) & \text{for all } a \in S \cup I \\
\alpha_a^T \colon D_S^T \to D_a^T & \alpha_a^T = D_a(in_{F_2}^*) \circ \alpha_{a,\mathsf{Tree}(F_2)} & \text{for all } a \in I \\
\delta_a^T \colon F_1(D_S^T) \times D_I^T \to D_a^T & \delta_a^T = D_a(in_{F_2}^*) \circ \delta_{a,\mathsf{Tree}(F_2)} & \text{for all } a \in S \\
\delta_a^T \colon F_1(D_S^T) \times D_I^T \to \mathsf{FM}(D_a^T) & \delta_a^T = \mathsf{FM}(D_a(in_{F_2}^*)) \circ \delta_{a,\mathsf{Tree}(F_2)} & \text{for all } a \in I
\end{array}
$$

A run of G on a tree $t \in \mathsf{Tree}(F_1)$ is a run of G^T on t.

Definition 11. Let $G = (S, I, D, \alpha, \delta)$ be a PAG from F_1 to F_2. We construct an AG $G^G = (S, I, D^G, \alpha^G, \delta^G)$ by instantiating the attribute domains and semantic functions to $\mathsf{DAG}(F_2)$ as follows:

$$
\begin{array}{ll}
D_a^G = D_a(\mathsf{DAG}(F_2)) & \text{for all } a \in S \cup I \\
\alpha_a^G = D_a(gr_{F_2}^*) \circ \alpha_{a,\mathsf{DAG}(F_2)} & \text{for all } a \in I \\
\delta_a^G = D_a(gr_{F_2}^*) \circ \delta_{a,\mathsf{DAG}(F_2)} & \text{for all } a \in S \\
\delta_a^G = \mathsf{FM}(D_a(gr_{F_2}^*)) \circ \delta_{a,\mathsf{DAG}(F_2)} & \text{for all } a \in I
\end{array}
$$

That is, we have that

$$
\begin{array}{ll}
\alpha_a^G \colon D_S^G \to D_a^G & \text{for all } a \in I \\
\delta_a^G \colon F_1(\mathsf{DAG}(F_2) \times D_S^G) \times \mathsf{DAG}(F_2) \times D_I^G \to D_a^G & \text{for all } a \in S \\
\delta_a^G \colon F_1(\mathsf{DAG}(F_2) \times D_S^G) \times \mathsf{DAG}(F_2) \times D_I^G \to \mathsf{FM}(D_a^G) & \text{for all } a \in I
\end{array}
$$

Let $\oplus = (\oplus_a \colon D_a^G \times D_a^G \to D_a^G)_{a \in I}$ be a family of associative and commutative binary operators and $g \in \mathsf{DAG}(F_1)$. A run of G on g modulo $\oplus$ is a run of G^G modulo $\oplus$ on g.

The two instantiations G^T and G^G can be related via unravelling:

Lemma 2. *Let $G = (S, I, D, \alpha, \delta)$ be a PAG from F_1 to F_2. Then we have the following:*

(i) For all $a \in I$ and $d \in D_S(\mathsf{DAG}(F_2))$,

$$D_a(\mathcal{U})(\alpha_a^G(d)) = \alpha_a^T(D_S(\mathcal{U})(d))$$

(ii) For all $a \in I$, $(s, (d_i)_{i<l}) \in F_1(D_S(\mathsf{DAG}(F_2)))$, $e \in D_I(\mathsf{DAG}(F_2))$, and $j < l$,

$$D_a(\mathcal{U})(\delta_{a,j}^G((s,(d_i)_{i<l}),e)) = \delta_{a,j}^G((s,(D_S(\mathcal{U})(d_i))_{i<l}), D_I(\mathcal{U})(e))$$

Please cite this article in press as: P. Bahr, E. Axelsson, Generalising tree traversals and tree transformations to DAGs: Exploiting sharing without the pain, Sci. Comput. Program. (2016), http://dx.doi.org/10.1016/j.scico.2016.03.006

(iii) For all $a \in S$, $(s, (d_i)_{i<l}) \in F_1(D_S(\mathsf{DAG}(F_2)))$, *and* $e \in D_I(\mathsf{DAG}(F_2))$,

$$D_a(\mathcal{U})(\delta_a^G((s, (d_i)_{i<l}), e)) = \delta_a^G((s, (D_S(\mathcal{U})(d_i))_{i<l}), D_I(\mathcal{U})(e))$$

Proof. Straightforward calculation. □

The notion of non-circularity of AGs carries over to PAGs in a straightforward manner by instantiation to trees: a PAG G is non-circular iff the AG G^T is non-circular.

As an immediate consequence of the definition of runs of PAGs as runs of AGs, we obtain the following corollary about non-circular PAGs:

Corollary 3. *Every non-circular PAG has a unique run on any given tree.*

Also the notion of copying AGs carries over to PAGs:

Definition 12. A PAG $G = (S, I, D, \alpha, \delta)$ from F_1 to F_2 is called *copying*, if $\delta_{a,j,A}((s, \overline{d}), (e_b)_{b\in I}) = e_a$ for all sets A, $a \in I$, $(s, \overline{d}) \in F_1(D_S(A))$, $j < \mathsf{ar}(s)$ and $(e_b)_{b\in I} \in D_I(A)$.

Given a copying, non-circular PAG G, we can show that, for each run of G on a DAG g, we find an equivalent run of G on $\mathcal{U}(g)$, and vice versa. Equivalence of runs is defined as follows: given a PAG $G = (S, I, D, \alpha, \delta)$ from F_1 to F_2, we say that a run ρ of G on a DAG $g \in \mathsf{DAG}(F_1)$ and a run ρ' of G on $\mathcal{U}(g)$ are *equivalent* if $\rho'_a(p) = D_a(\mathcal{U})(\rho_a(g[p]))$ for all $a \in S \cup I$ and $p \in \mathcal{P}(g)$.

Theorem 4. *Given a copying, non-circular PAG* $G = (S, I, D, \alpha, \delta)$ *from* F_1 *to* F_2, *a copying* $\oplus = (\oplus_a \colon D_a^G \times D_a^G \to D_a^G)_{a\in I}$, *and a DAG* $g = (N, E, r) \in \mathsf{DAG}(F_1)$, *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 $\rho'_a(p) = D_a(\mathcal{U}())(\rho_a(g[p]))$. For the converse direction, we construct a run ρ' on g from scratch according to the semantics of PAGs. The run ρ' is defined by well-founded recursion using the non-circularity of G. We then prove by well-founded induction that any run ρ on $\mathcal{U}(g)$ must be equivalent to ρ'. By using Lemma 2, we can reuse large parts of the proof of Theorem 1. □

Corollary 4. *Given* G, $\oplus$, *and* g *as in Theorem 4, we have that* $(\!|G, \oplus|\!)(g) = [\![G]\!](\mathcal{U}(g))$.

Proof. Due to the correspondence of runs according to Theorem 4, uniqueness of runs of G on DAGs follows from the uniqueness of its runs on trees (cf. Corollary 3). Hence, both $(\!|G, \oplus|\!)(g)$ and $[\![G]\!](\mathcal{U}(g))$ are defined, and by Theorem 4 they are equal. □

Similarly to notion of circularity, we lift the definition of monotonicity to PAGs by instantiation of PAGs to trees. However, since the conflict resolution operators $\oplus_a$ for PAGs are defined on the instantiation of PAGs to DAGs, decreasingness is defined via unravelling:

Definition 13. Let $G = (S, I, D, \alpha, \delta)$ be a PAG equipped with a quasi-order $\lesssim_a$ on D_a^T for each $a \in S \cup I$. G is called *monotone* w.r.t. $(\lesssim_a)_{a\in S\cup I}$ if G^T is monotone w.r.t. $(\lesssim_a)_{a\in S\cup I}$. Moreover, $\oplus = (\oplus_a \colon D_a^G \times D_a^G \to D_a^G)_{a\in I}$ is called decreasing w.r.t. $(\lesssim_a)_{a\in I}$ if we have, for all $a \in I$, that

$$D_a(\mathcal{U})(d_1 \oplus_a d_2) \lesssim_a D_a(\mathcal{U})(d_1), D_a(\mathcal{U})(d_2) \text{ for all } d_1, d_2 \in D_a^G$$

Having established the necessary terminology, we can generalise Theorem 2 to PAGs:

Theorem 5. *Let* $G = (S, I, D, \alpha, \delta)$ *be a non-circular PAG,* $\oplus = (\oplus_a \colon D_a^G \times D_a^G \to D_a^G)_{a\in I}$ *associative and commutative, and* $(\lesssim_a)_{a\in S\cup I}$ *quasi-orders such that* G *is monotone and* $\oplus$ *is decreasing. 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 that*

$$D_a(\mathcal{U})(\rho_a(g[p])) \lesssim_a \rho'_a(p) \text{ for all } a \in S \cup I \text{ and } p \in \mathcal{P}(g).$$

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 5. *Given* G, $\oplus$, $(\lesssim_a)_{a\in S\cup I}$, *and* g *as in Theorem 5, and provided that* $(\!|G, \oplus|\!)(g)$ *is defined, we have that* $D_S(\mathcal{U})((\!|G, \oplus|\!)(g)) \lesssim_S [\![G]\!](\mathcal{U}(g))$.

Please cite this article in press as: P. Bahr, E. Axelsson, Generalising tree traversals and tree transformations to DAGs: Exploiting sharing without the pain, Sci. Comput. Program. (2016), http://dx.doi.org/10.1016/j.scico.2016.03.006

Proof. Given the unique runs ρ and ρ' on g and $\mathcal{U}(g)$, respectively, we have the following according to Theorem 5:

$$D_S(\mathcal{U})((\!|G, \oplus|\!)(g)) = D_S(\mathcal{U})(\rho_S(r)) = D_S(\mathcal{U})(\rho_S(g\,[\langle\rangle])) \lesssim_S \rho'_S(\langle\rangle) = [\![G]\!]\,(\mathcal{U}\,(g)) \qquad \square$$

7. Extended example

As a demonstration of the versatility of our approach, we have implemented a size-based simplifier for a subset of the Feldspar EDSL [5]. The code is found in the file `Feldspar.hs` in the accompanying repository [8]. The simplifier is implemented as a RAG as follows:

```
simplifyDag :: Dag Feldspar → Dag Feldspar
simplifyDag = runRewrite_G intersection (sizeInf_S ⊗ constFold_S) sizeInf_I simplifier (const Map.empty)
            ∘ renameFeld
```

The RAG consists of four parts:

- $sizeInf_S$ synthesises a *Size* attribute, which gives a conservative approximation of the set of values an expression might take on.
- $constFold_S$ synthesises a *Maybe Value* attribute, which gives the value of constant expressions of Boolean or integer type.
- $sizeInf_I$ computes the inherited environment attribute. The environment gives the size of variables in scope.
- *simplifier* rewrites a node based on the inferred *Size* and *Value* attributes.

We use *intersection* to resolve inherited attributes for shared nodes, just like for $typeInf_G$. The function *renameFeld* makes sure that the DAG is well-scoped according to the definition in section 4.3.

The AG consisting of $sizeInf_S$ and $sizeInf_I$ is similar in structure to the type inference AG in section 3. Size is represented as a list of ranges, and a range is a pair of an upper bound and a lower bound:

```
type Size  = [Range]
type Range = (Maybe Integer, Maybe Integer)
```

Lists of ranges are needed for array expressions. For example, the size $[r_1, r_2, re]$ is for a two-dimensional array where the extent of the outer dimension is within r_1, the extent of the inner dimension is within r_2 and each element is in the range re.

The additional attribute computed by $constFold_S$ is used to propagate constant values. For integer expressions, $constFold_S$ falls back to the inferred size, since an expression with a singleton range must be a constant.

The *simplifier* function makes use of the inferred sizes and constants to rewrite nodes. Here are a few interesting cases:

```
simplifier :: (Size ∈ as, Maybe Value ∈ as, Env Size ∈ as) ⇒ Rewrite Feldspar as Feldspar
simplifier _
  | Just (B b) ← above          = In (LitB b)
  | Just (I i) ← above          = In (LitI i)
simplifier (Add a b)
  | Just 0 ← valueOfI a         = Ret b
  | Just 0 ← valueOfI b         = Ret a
simplifier (Min a b)
  | Just True ← liftA2 (⩽) ua lb = Ret a
  | Just True ← liftA2 (⩽) ub la = Ret b
  where [(la, ua)] = sizeOf a
        [(lb, ub)] = sizeOf b

...
```

The following two functions are used to query the synthesised attributes:

```
sizeOf   :: (?below :: a → as, Size ∈ as)       ⇒ a → Size
valueOfI :: (?below :: a → as, Maybe Value ∈ as) ⇒ a → Maybe Integer
```

The first two cases of *simplifier* rewrite directly to a literal if the expression is constant. The *Add* cases simplify expressions of the form $0 + b$ and $a + 0$. Finally, the case for *Min* uses the size of the operands to reduce the node statically if the sizes are disjoint (or overlap by at most one value). The function *liftA2* is used to lift the $(\leqslant)$ operator to values of type *Maybe Integer* such that the result is *Nothing* whenever either argument is *Nothing*.

Please cite this article in press as: P. Bahr, E. Axelsson, Generalising tree traversals and tree transformations to DAGs: Exploiting sharing without the pain, Sci. Comput. Program. (2016), http://dx.doi.org/10.1016/j.scico.2016.03.006

7.1. Practical relevance

The graph-based simplifier above can simplify Feldspar programs represented as graphs without any loss of sharing. Does this property of the simplifier have any practical relevance?

An alternative, employed by the current Feldspar implementation, is to represent Feldspar programs as trees and express the simplifier as a standard recursive function over such trees. As discussed in section 2.1, this approach requires adding a syntactic let binding construct to manage the size of the ASTs for programs with lot of sharing. A problem with let bindings is that they get in the way of the simplifier. For example, in order to simplify the Haskell expression **let** $a = 0$ **in** $a + x$ to x using the rule $0 + x \to x$, the compiler has to first inline the let binding. Feldspar inlines all bindings in order to maximise the effect of the simplifier, but this strategy risks blowing up the AST. In general, compilers rely on heuristics to decide which binders to inline [56], and there is always a risk of missing important simplifications due to let bindings getting in the way.

It is reasonable to ask to what extent AST blow-up occurs in practice. Can we get away with Feldspar's inline-always strategy, or perhaps by relying on explicitly expressing sharing in cases where it matters [40]?

It turns out to be quite easy to accidentally blow up the AST in an embedded DSL. The following function from Feldspar's source code (slightly adapted) computes the number of leading zeroes in the bit representation of an integer:

```
nlz x = bitCount $ complement $ foldl go x $ takeWhile (<32) $ map (2^) $ [(0 :: Integer)..]
  where go b s = b .|. (b .>>. s)
```

It performs a left fold of the function *go* over the list $[1, 2, 4, 8, 16]$. The operations *bitCount*, (.|.), etc. operate on Feldspar's AST type. For example, (.|.) takes two ASTs, representing the operands, and constructs an AST representing the bitwise *or* of the operands. The argument to *foldl* is a static Haskell list, so the function will unfold at Feldspar's compile time, yielding an AST with only primitive Feldspar functions in it.

Note that the argument b of the local function *go* is used twice in each iteration. This means that the AST produced by *nlz* will have a lot of sharing. This problem was not taken into consideration when adding *nlz* to Feldspar, and it has caused problems in one of our projects. Feldspar has an implementation of the Fast Fourier Transform (FFT) which uses *nlz* to compute the number of stages needed in the FFT. One Feldspar application made use of an FFT followed by an inverse FFT with some transformations in between. This lead to nested applications of *nlz* resulting in an AST for which the compiler effectively did not terminate. Only by accident did we later discover that *nlz* was the culprit, and the problem could be solved by sharing the variable b explicitly.

Although the technique in this paper has not made it into the Feldspar implementation, we are confident that it would have helped in the above situation. As a simple demonstration, the file `Feldspar.hs` in the accompanying repository [8] implements *nlz* and shows that while *runRewrite* is sensitive to the blow-up caused by nested uses of *nlz*, $\mathit{runRewrite}_G$ manages just fine.

8. Implementation

8.1. Representation of DAGs

Our implementation represents DAGs by explicit mappings from nodes, which are represented by integers, to their outgoing edges. In section 4, we presented the following definition of the type *Dag*:

```
data Dag f = Dag {root  :: Node,
                  edges :: IntMap (f Node)}
```

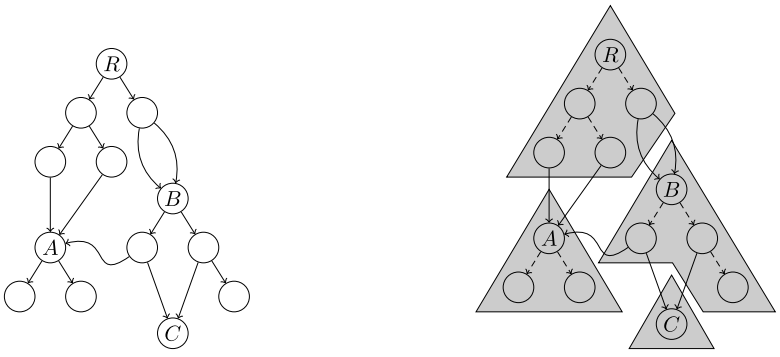
The *edges* field provides a mapping that maps each node to its outgoing edges, which are represented by the type *f Node*.

However, this naive representation is inefficient. Typical DAGs have a large chunks that are tree-shaped, with some edges in between that provide sharing. For example, consider the DAG pictured in Fig. 9a. There are only three nodes (A, B, and C) that have more than one incoming edge and are thus shared. The remaining nodes – with only a single incoming edge each – can be thought of as nodes in a tree rooted in one of the shared nodes (or the root R of the whole DAG). This idea is illustrated in Fig. 9b. Each of the shaded areas constitutes a self-contained tree. These trees are connected via edges between them. This two-level representation – a DAG whose nodes are trees – is more efficient. Only edges between the trees (pictured as solid arrows) have to be represented explicitly via an *IntMap*, i.e. a PATRICIA tree [53]. The edges within the tree structures (pictured as dashed arrows) can be represented using an algebraic data type.

Concretely, instead of representing edges by the type *f Node*, we shall represent edges using the type *f* (*Free f Node*). That is, we use the free monad *Free f* to represent the nested tree structures. Apart from that, we also represent the root of the DAG by the type *f* (*Free f Node*), rather than simply *Node*. In sum, we obtain the following representation:

```
data Dag f = Dag {root      :: f (Free f Node),
                  edges     :: IntMap (f (Free f Node)),
                  nodeCount :: Int}
```

Please cite this article in press as: P. Bahr, E. Axelsson, Generalising tree traversals and tree transformations to DAGs: Exploiting sharing without the pain, Sci. Comput. Program. (2016), http://dx.doi.org/10.1016/j.scico.2016.03.006



(a) Naive representation. (b) Hybrid representation.

Fig. 9. Representation of DAGs.

In addition, the *Dag* data structure also contains the number of (explicit) nodes in the *Dag*. This information will be helpful for allocating memory for implementing AGs efficiently.

Note that we could have also chosen to represent edges using the type *Free f Node* instead of *f* (*Free f Node*). However, with the former, we would also represent "empty" edges, which represent indirection. Such empty edges could potentially be useful, but we found no application for them in our implementation of AGs.

Apart from an overall more compact representation that allows us to implement AGs more efficiently, the *Dag* data structure provides specific benefits for the implementation of both RAGs and PAGs. Transformations on DAGs described by RAGs and PAGs introduce embedded tree-shaped fragments inside the result DAG by way of the free monad *Free f*. If we were to implement RAGs and PAGs using the naive representation of DAGs, we would have to allocate fresh nodes for each implicit inner node inside a free monad data structure of type *Free f Node*. Such an implementation would be very inefficient and would lose a lot of the speedup gains obtained by sharing.

8.2. Implementing attribute grammars on trees and DAGs

8.2.1. Preparations

Before we start with the actual implementation of AGs, we need to implement one important auxiliary functionality: an instance for the *Mapping* type class, which is essential for the implementation of inherited attributes. The intention of the *Mapping* class is to provide a representation of mappings that assign attribute values to child positions. In the simplest case, the type *Inh* is a mapping of type $f\ c \to m\ i$, with the type constraint *Mapping m c*. The idea of implementing an instance of *Mapping* is to *uniquely number* the child nodes of type *c*. Then a mapping from child positions to attribute values of type *v* is provided by a simple integer map of type *IntMap v*.

For numbering arbitrary values, we introduce the following type *Numbered*:

```
data Numbered a = Numbered Int a

unNumbered :: Numbered a → a
unNumbered (Numbered _ x) = x
```

There are many different ways of numbering the elements of type *c* in a structure of type *f c*, given that *f* is *Traversable*. Here we simply use a state monad to keep track of the counter:

```
number :: Traversable f ⇒ f c → f (Numbered c)
number x = evalState (Traversable.mapM run x) 0
where run :: c → State Int (Numbered c)
      run b = do n ← get
                 put (n + 1)
                 return (Numbered n b)
```

We then wrap the *IntMap* type in a **newtype** to construct the type that we use to instantiate the *Mapping* type class. Implementing the methods of *Mapping* is straightforward using the underlying *IntMap*.

```
newtype NumMap k v = NumMap (IntMap v) deriving (Functor, Foldable, Traversable)

instance Mapping (NumMap k) (Numbered k) where
  ...
```

Please cite this article in press as: P. Bahr, E. Axelsson, Generalising tree traversals and tree transformations to DAGs: Exploiting sharing without the pain, Sci. Comput. Program. (2016), http://dx.doi.org/10.1016/j.scico.2016.03.006

```
runAG :: ∀ f s i.Traversable f ⇒ Syn' f (s, i) s → Inh' f (s, i) i → (s → i) → Tree f → s
runAG syn inh init t = sFin where
  sFin = run iFin t
  iFin = init sFin
  run :: i → Tree f → s
  run i (In t) = s where
    recurse (Numbered n c) = let i' = lookupNumMap i n m
                             in Numbered n (run i' c, i')
    t'  = fmap recurse (number t)
    m = explicit inh (s, i) unNumbered t'
    s  = explicit syn (s, i) unNumbered t'
```

Fig. 10. Implementation of *runAG*.

In addition, we provide a lookup function for *NumMap*, which simply uses the underlying lookup function of the *IntMap* type:

```
lookupNumMap :: a → Int → NumMap t a → a
lookupNumMap d k (NumMap m) = IntMap.findWithDefault d k m
```

Next we shall see how this implementation of the *Mapping* interface is used for implementing AGs on trees and DAGs.

8.2.2. Implementing attribute grammars on trees

We first take a look at the simplest case: the implementation of AGs on trees. The implementation of *runAG* is shown in Fig. 10. At the top-level, *runAG* computes *sFin* and *iFin*, the synthesised resp. inherited attribute at the root of the tree. The traversal of the tree is performed by the *run* function, which takes the inherited attribute value at the current node and returns the synthesised attribute of the current node. Before applying the semantic functions *inh* and *syn*, we number the child nodes using *number* and recursively apply *run* to the child nodes via *fmap recurse*. In order to apply the semantic functions, we have to supply the implicit parameters *?above* and *?below*. To this end, we use the combinator *explicit*, which turns these implicit parameters into explicit arguments:

```
explicit :: ((?above :: q, ?below :: a → q) ⇒ b) → q → (a → q) → b
explicit x ab be = x where ?above = ab; ?below = be
```

The *?above* parameter is given the value (i, s), consisting of the inherited attribute given as argument to *run* and the synthesised attribute we have computed. The *?below* parameter is provided by the *unNumbered* function, which simply strips away the numbering that we have performed earlier, thus exposing the attribute values that have been recursively computed via *recurse*.

An important aspect of this implementation is its circular nature. It essentially depends on the non-strict semantics of Haskell. The circularity can be seen immediately in the definition of *sFin* and *iFin*, which refer to each other, and the definition of *s* at the bottom, which refers to itself.

8.2.3. Implementing attribute grammars on DAGs

The implementation of AGs on DAGs is a bit more intricate. The main idea, however, is quite simple: We construct mappings from nodes in the DAG to inherited and synthesised attributes. To this end our implementation assumes that nodes are numbers from 0 to *nodeCount* − 1; the latter given by the field *nodeCount* of the *Dag* record. We make sure that all operations that construct DAGs – e.g. *reifyDag* – or transform DAGs – e.g. $runRewrite_G$ – maintain this invariant. To construct and maintain these mappings between nodes and attribute values, we use two *mutable* array data structures: *imap* of type *MVector st* (*Maybe i*) to store the inherited attribute values of type *i*, and *smap* of type *MVector st s* to store the synthesised attribute values of type *s*. The type variable *st* is used for the *ST* monad to restrict the side effects for dealing with ephemeral data structures to the *runM* function.

The difference in the types for *imap* and *smap* – the fact that *imap* uses *Maybe* – is crucial and characterises the difference between computing inherited vs. synthesised attributes. As we have illustrated in Fig. 6, we may have to compute inherited attributes for a given node several times – once for each incoming edge. The *Maybe* type allows us to keep track of whether we already computed an attribute value for a given node. If so, we need to use the conflict resolution function *res* to combine the previously computed value with a newly incoming value. If not, we can safely store a newly incoming value as the attribute value of the node. This behaviour is implemented in the *runF* auxiliary function in Fig. 11.

Apart from this caching of attribute values using arrays, the implementation of $runAG_G$ follows a pattern similar to *runAG*. The auxiliary function *run* applies the semantic functions *inh* and *syn* using *explicit* and numbering of child nodes. However, in contrast to *runAG*, we do not use the *number* function to perform the numbering. Instead we make use of the fact that the *ST* monad allows us to allocate a single counter reference *count*, which is then used to do the numbering.

Please cite this article in press as: P. Bahr, E. Axelsson, Generalising tree traversals and tree transformations to DAGs: Exploiting sharing without the pain, Sci. Comput. Program. (2016), http://dx.doi.org/10.1016/j.scico.2016.03.006

```
runAG_G :: ∀ f i s.Traversable f ⇒ (i → i → i) → Syn f (s, i) s → Inh f (s, i) i → (s → i)
                                 → Dag f → s
runAG_G res syn inh init Dag {edges, root, nodeCount} = sFin where
  sFin = runST runM
  iFin = init sFin
  runM :: ∀ st.ST st s
  runM = mdo
    imap ← MVec.new nodeCount    -- construct empty mapping from nodes to inh. attrs.
    MVec.set imap Nothing        -- set inh. attrs. to Noting
    smap ← MVec.new nodeCount    -- allocate mapping from nodes to syn. attrs.
    count ← newSTRef 0           -- allocate counter for numbering child nodes
    let  -- run the AG on an edge with the given input inh. attr. and produce
         -- the output syn. attr.
      run :: i → f (Free f Node) → ST st s
      run i t = mdo
          -- apply the semantic functions
        let s = explicit syn (s, i) unNumbered result
            m = explicit inh (s, i) unNumbered result
              -- recurses into the child nodes and numbers them
            run' :: Free f Node → ST st (Numbered (s, i))
            run' c = do n ← readSTRef count
                        writeSTRef count (n + 1)
                        let i' = lookupNumMap i n m
                        s' ← runF i' c    -- recurse
                        return (Numbered n (s', i'))
        writeSTRef count 0    -- re-initialise counter
        result ← Traversable.mapM run' t
        return s
      runF :: i → Free f Node → ST st s    -- recurses through the tree structure
      runF i (Ret x) = do    -- we found a node: update the mapping for inh. attrs.
        old ← MVec.unsafeRead imap x
        let new = case old of
                    Just o    → res o i
                    _         → i
        MVec.unsafeWrite imap x (Just new)
        return (smapFin ! x)
      runF i (In t) = run i t
        -- This function is applied to each edge
      iter (n, t) = do s ← run (fromJust (imapFin ! n)) t
                       MVec.unsafeWrite smap n s
    s ← run iFin root                    -- first apply to the root
    mapM _ iter (IntMap.toList edges)    -- then apply to the edges
      -- finalise the mappings for attribute values
    imapFin ← Vec.unsafeFreeze imap
    smapFin ← Vec.unsafeFreeze smap
    return s
```

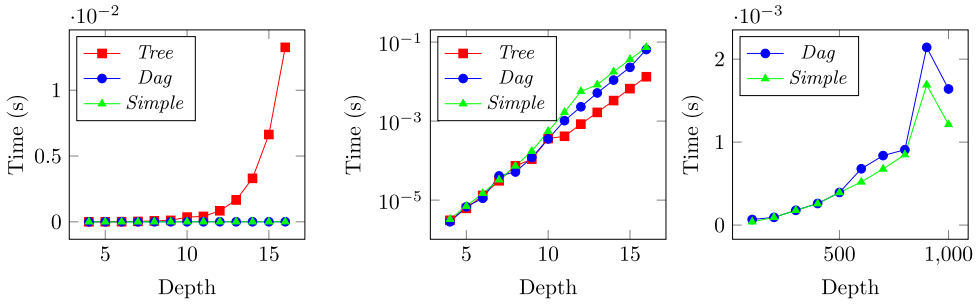
Fig. 11. Implementation of $runAG_G$.

Like *runAG*, a characteristic feature of $runAG_G$ is its circularity; we make essential use of Haskell's lazy evaluation. To achieve this circularity in a monadic function definition we use the **mdo** keyword, which provides a convenient interface for the underlying *MonadFix* instance of the *ST* monad. Like in the definition for *runAG*, we use circularity for applying the semantic functions. But in addition, circularity is also used for accessing the final attribute values stored in the two arrays *imap* and *smap*. We use the function *unsafeFreeze* to turn these two mutable arrays into immutable arrays, which are then used in the construction of *imap* and *smap*.

8.2.4. Implementation of PAGs

The implementation of AGs on trees and DAGs can be readily generalised to PAGs: we simply follow the instantiation of a PAG G to the AG G^T respectively G^G as described in section 6.3. This instantiation is straightforward for trees. However, the instantiation to DAGs, adds considerable complexity. We need to allocate fresh nodes and edges that are described by the semantic functions of the PAG G. Our hybrid representation of DAGs (cf. section 8.1) makes this process substantially simpler and more efficient. Nonetheless, we have to make sure that all DAGs that are constructed by the PAG are self-contained and

Please cite this article in press as: P. Bahr, E. Axelsson, Generalising tree traversals and tree transformations to DAGs: Exploiting sharing without the pain, Sci. Comput. Program. (2016), http://dx.doi.org/10.1016/j.scico.2016.03.006



(a) Full sharing. (b) No sharing. (c) Large input, full sharing.

Fig. 12. Benchmark: *leavesBelow* implemented as AG.

satisfy the invariant that nodes are numbered from 0 to *nodeCount* − 1. Self-containedness is non-trivial, since during the run of a PAG, semantic functions have access to all attributes and thus a common pool of DAG nodes and edges. As a consequence, a single node may end up being shared among several DAGs that are constructed by the PAG. We elide the implementation details of the PAG implementation and refer the reader to the accompanying source code repository [8].

8.3. Performance results

We have implemented benchmarks to evaluate the efficiency of our approach. The benchmarks are found in the code associated with this paper [8].

The benchmarks focus on two extreme situations: (1) DAGs with a lot of sharing (each node is shared), and (2) DAGs with no sharing at all. The assumption behind the paper is that we have structures with a lot of sharing. The first extreme tests that we succeed in dealing with this case more efficiently. At the same time, the second extreme is meant to show what is the overhead of working with DAGs in cases when it is not needed.

The measurements were done on balanced trees of different depths. The trees were represented as ordinary trees and as DAGs (i.e. using the *Tree* and *Dag* types). To gauge the performance trade-off that the hybrid DAG representation (as described in section 8.1) offers, we also measure the performance of the naive DAG representation, which is labelled as "*Simple*" in the diagrams.

The DAGs were constructed in two different ways: with sharing at each level and with no sharing at all. A balanced tree of depth d has $2^d - 1$ nodes, and thus the DAG without sharing also has $2^d - 1$ nodes. The DAG with sharing at each level has d nodes.

At first, we consider the AG implementation of *leavesBelow* from the introduction. It is computationally inexpensive (since all leave nodes in the input are labelled the same) and thus provides a measure of the overhead of working with DAGs. Fig. 12a shows the time that the different implementations of *leavesBelow* take for different input sizes (measured by depth). The run time is proportional to the size of the tree: the run time grows exponentially without sharing and linearly with sharing.

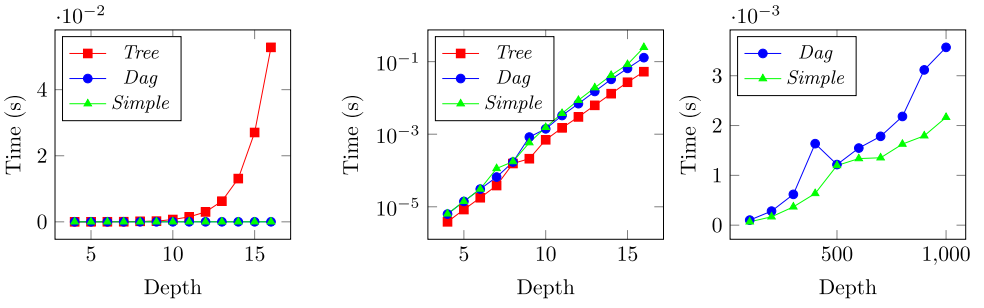
We can see that there is some overhead in using the *Dag* representation instead of *Tree* when there is no sharing. Fig. 12b show the overhead of the DAG representation in case of no sharing. The overhead stays below a factor of 4 for the hybrid DAG representation, and a factor of 7 for the naive representation. Fig. 12c shows that the run time for large DAGs with maximal sharing increases roughly linearly even at large depths – with no significant difference between the hybrid and the naive DAG representation.

The purpose of the measurements in Fig. 12b is to see the overhead of AGs on DAGs in case of little sharing. To this end we have disabled an optimisation in our implementation that falls back to the tree-based implementation of AGs in case of no sharing. With this optimisation in place, the overhead of using $runAG_G$ instead of *runAG* is reduced to zero for input with no sharing. However, this optimisation only works if the input DAG is represented optimally, i.e. using explicit pointers only if there is sharing. While this property is guaranteed by the DAGs produced by *reifyDag*, DAGs that are the result of a transformation e.g. by $runRewrite_G$ may not have this property.

Next, we look at the performance of the *runRewrite* and $runRewrite_G$ implementation. To this end, we consider the RAG implementation of *repmin* example from section 3.5 using the same input as for the *leavesBelow* benchmark above. The resulting measurements, shown in Fig. 13, have the same characteristics as the corresponding measurements from Fig. 12: the DAG version is asymptotically better than the tree version in case of sharing. In case of no sharing the overhead remains below a factor of 2.5 for the hybrid DAG representation and around 3 for the naive representation. However, *repmin* is a very simple transformation, and the performance of the naive DAG representation regresses considerably if we consider a variation of *repmin* that replaces each leaf node with two leaf nodes – one with the global minimum label and one with the old label – the speedup afforded by the hybrid transformation becomes more pronounced.

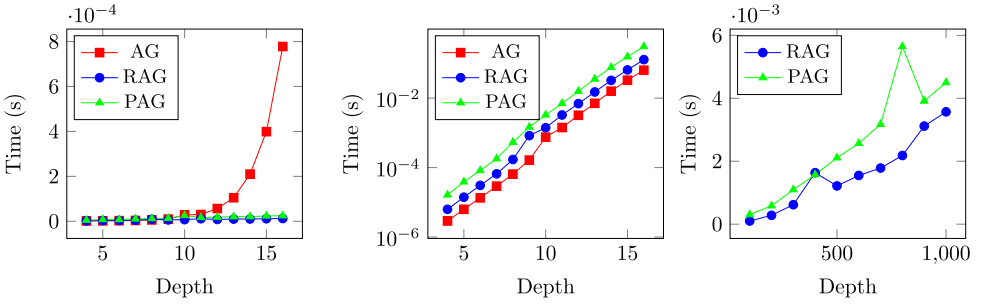
We have also considered the performance of the implementation of PAGs: Fig. 14 compares the PAG-based implementation of *repmin* – as discussed in section 6.2 – with the RAG- and AG-based implementations. We can see that the PAG

Please cite this article in press as: P. Bahr, E. Axelsson, Generalising tree traversals and tree transformations to DAGs: Exploiting sharing without the pain, Sci. Comput. Program. (2016), http://dx.doi.org/10.1016/j.scico.2016.03.006



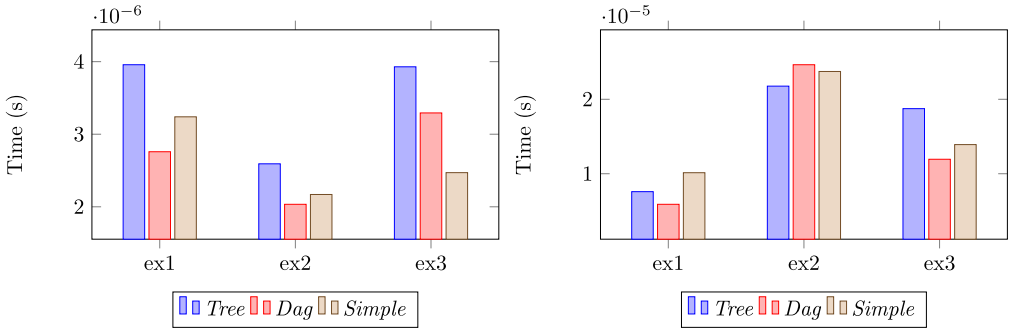
(a) Full sharing. (b) No sharing. (c) Large input, full sharing.

Fig. 13. Benchmark: *repmin* implemented as RAG.



(a) Full sharing. (b) No sharing. (c) Large input, full sharing.

Fig. 14. Benchmark: *repmin* implemented as AG vs. RAG vs. PAG.



(a) Type inference. (b) Feldspar simplifier.

Fig. 15. Benchmark: type inference and Feldspar simplifier.

implementation has a asymptotic run time characteristic similar to the RAG-based implementation. However, we can observe some overhead of the PAG-based implementation.

Finally, we look at two more realistic computations: the type inference implementation from section 2 and the size-based simplifier from section 7. We applied each to three small example programs. The results are depicted in Fig. 15. The input programs are fairly small and only have very little sharing. The examples are too simple to draw any real conclusions from, but we observe that all three options are of comparable speed, as expected when the amount of sharing is small.

In our benchmarks here we have focused on the difference between AGs on trees and DAGs, and the effect of sharing on the runtime performance. However, it is important to realise that there is also an overhead associated with AGs – at least when implemented by leveraging lazy evaluation as we did in this paper. In practical application this overhead can be substantial. Using our shallow embedding of AGs we have found an overhead amounting to as much as an order of magnitude. We expect that most of this overhead can be eliminated by implementing AGs using statically scheduled attribute evaluation. For AG on DAG a scheduling algorithm can be extracted from the proof of Proposition 2. Such a scheduling algorithm could then be used to compile AG definitions into efficient Haskell code in the style of Kuiper and Swierstra [44].

Please cite this article in press as: P. Bahr, E. Axelsson, Generalising tree traversals and tree transformations to DAGs: Exploiting sharing without the pain, Sci. Comput. Program. (2016), http://dx.doi.org/10.1016/j.scico.2016.03.006

9. 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 [32]; 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 [54] introduced a purely functional representation of graphs, called *structured graphs*, using Chlipala's *parametric higher-order abstract syntax* [20]. 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 [55] generalised structured graphs to indexed data structures with particular focus on EDSLs. Bahr [7] used structured graphs to lift compiler implementations and their correctness proofs from trees to DAGs. However, his approach is limited to folds. 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 [60], 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 know, 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. [16] introduce the 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 may even be cyclic). The motivation behind UnQL is similar to ours; however, UnQL does not appear to support propagation and merging of accumulating parameters (which correspond to our inherited attributes) in recursive functions. More recently, Hidaka et al. [35] have introduced the language λ_{FG}, which is based on the underlying calculus of UnQL (called UnCAL), but works on ordered graphs. While λ_{FG} is more expressive than UnQL, it still lacks the ability to merge accumulating parameters.

Recursion schemes Generic recursion schemes [50] provide fixed schemes for traversing regular data types. The most common recursion scheme is the generic fold. Gibbons [31] introduced a recursion scheme for traversals with accumulating parameters reminiscent of an AG with inherited attributes. Recursion schemes are normally defined for trees and, as such, do not deal with sharing.

Tree compression We use DAGs as compact representations of trees with the goal of improving runtime performance of tree traversals and tree transformations. But there are many more approaches to compress trees [62]. For example, tree grammars have been extensively studied as compact representation for trees [49,47,17]. DAGs can only express repetition of subtrees (i.e. common subexpressions) in order to achieve compression. Tree grammars, on the other hand, can also express repetition of tree *patterns* and thus offer more opportunity for compression, which may result in an exponentially smaller representation compared to DAGs [47]. Recently, so-called *top DAGs* have been introduced by Bille et al. [13], which also can express tree pattern repetition. The downside of these more expressive representations, is the lack of recursion schemes that both match the expressiveness of attribute grammars and our efficient implementation (cf. the discussion on automata below). Moreover, the goal of our work is to leverage the sharing information that is already present in embedded DSL implementations. For this purpose DAGs are sufficient; the more sophisticated compression offered by tree grammars and top DAGs requires considerable computational effort: although there are fast approximations, finding a minimal tree grammar is NP-hard [19] while a minimal DAG can be constructed in linear time [21].

Tree and graph automata There is a strong relationship between tree automata and attribute grammars: bottom–up acceptors correspond to synthesised attributes and top–down acceptors correspond to inherited attributes. The difference is that automata are typically 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 [30], i.e. tree automata that characterise tree transformations, and our representation of these automata in Haskell is based on Hasuo et al. [33]. Our representation of AGs in Haskell is taken from Bahr's *modular tree automata* [6], which are in turn derived from representations of tree automata based on the work of Hasuo et al. [33]. However, we slightly adjusted the representation of top–down state propagation to obtain a more abstract interface that allowed us to implement semantic functions of inherited attributes more efficiently. Moreover, we derived our notion of parametric attribute grammars from Bahr and Day [11], who recognised that the addition of parametricity in the state space corresponds to the generalisation of tree transducers to macro tree transducers [25].

While a number of generalisations of tree automata to graphs have been studied, a unified notion of graph automata remains elusive [59]. Only specialised graph automata for particular applications have been proposed thus far, 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 $\oplus$) [47,29], restrict themselves to bottom–up state propagation only [18,3,26], or assume that the in-degree of nodes is fixed for each

Please cite this article in press as: P. Bahr, E. Axelsson, Generalising tree traversals and tree transformations to DAGs: Exploiting sharing without the pain, Sci. Comput. Program. (2016), http://dx.doi.org/10.1016/j.scico.2016.03.006

node label (i.e. data constructor) [37,58]. 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. [43] 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) [24], 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 [52] were the first to present an embedding of AGs directly into Haskell with the aim of compositionality of AG definitions. Viera et al. [66] presented an improved embedding that also ensured static well-formedness of AG definitions. 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. Nonetheless, one could imagine using Template Haskell to also produce an appropriate, specialised DAG type that corresponds to a given algebraic data type.

The idea to utilise the structure of attributes that happen to be tree-structured – as our parametric AGs from section 6 do – also appears in the literature on AGs, albeit with a different motivation: so-called *higher-order attribute grammars* [67] permit the execution of the AG nested within those tree-structured attributes. By composing parametric AGs sequentially similarly to the composition of tree transducers [30], we can achieve the same effect.

Higher-order attribute grammars implicitly introduce sharing when duplicating higher-order attributes. Saraiva et al. [63] 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.

There is a large body of work on reference AGs [34,23,48,64,28], an extension of AGs that allows attributes to be references to nodes in the tree. This is similar to our notion of parametric AGs, where attributes may be (or contain) references to nodes in the tree (or DAG). However, the references in reference AGs are to nodes in the original input tree and the semantic functions have access to the attributes of referenced nodes. Thus, reference AGs permit very flexible non-local access of attributes. In contrast, parametric attributes contain references to nodes in the trees (or DAGs) that are constructed as output, and such nodes have no attributes themselves. The references in reference AGs are also different from the sharing structure that we consider in this paper: we consider sharing that is inherent in the input (in the form of a DAG structure), whereas the sharing in a reference AGs is part of the dynamic behaviour of the AG. Recent implementations of reference attribute grammars [64,28] use caching that utilises this sharing structure to avoid recomputation of attributes for shared subtrees.

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 important 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. But there are also DFA frameworks that do support bidirectional analyses [38,39].

The differences become more pronounced if we consider the parametric AGs described in section 6, 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 [45]. However, we are not aware of a DFA framework that combines bidirectional analyses with graph transformations.

Overloaded projections The $\in$ type class and the *pr* method provide projection of single attributes from a collection of attributes. Class-based encodings of projections have been used in various other contexts. For example, the “classy optics” method [68] gives a general way to create functional lenses that focus on sub-structures based on their type rather than their position in the structure. Lenses are move powerful than *pr* as they provide both a “getter function” (projection) and a “setter function” that replaces the sub-structure in focus.

10. 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 (Theorems 2 and 5) provides a widely applicable proof principle since it is parametric in the quasi-order. We have presented

Please cite this article in press as: P. Bahr, E. Axelsson, Generalising tree traversals and tree transformations to DAGs: Exploiting sharing without the pain, Sci. Comput. Program. (2016), http://dx.doi.org/10.1016/j.scico.2016.03.006

four examples for which correspondence by monotonicity gives useful results: *leavesBelow*, *typeInf*, *gateDelay* and *simplify* (cf. [10] for the formal argument).

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 2 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 *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 [51,27,61] 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 [36,69].
- Support deep pattern matching in AGs. This can be done by extending the *Inh*, *Syn*, 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.

Acknowledgements

We would like to thank the attendees of PEPM 2015 as well as the anonymous referees for their insightful comments and suggestions. 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, N. Ghani, Categories of containers, in: FoSSaCS, 2003.
[2] A.V. Aho, R. Sethi, J.D. Ullman, Compilers: Principles, Techniques, and Tools, Addison–Wesley Longman Publishing Co., Inc., Boston, MA, USA, ISBN 0-201-10088-6, 1986.
[3] S. Anantharaman, P. Narendran, M. Rusinowitch, Closure properties and decision problems of dag automata, Inf. Process. Lett. 94 (5) (2005) 231–240.
[4] E. Axelsson, Functional programming enabling flexible hardware design at low levels of abstraction, PhD thesis, Chalmers University of Technology, 2008.
[5] E. Axelsson, K. Claessen, G. Dévai, Z. Horváth, K. Keijzer, B. Lyckegård, A. Persson, M. Sheeran, J. Svenningsson, A. Vajda, Feldspar: a domain specific language for digital signal processing algorithms, in: MEMOCODE, 2010.
[6] P. Bahr, Modular tree automata, in: MPC, 2012.
[7] P. Bahr, Proving correctness of compilers using structured graphs, in: M. Codish, E. Sumii (Eds.), Functional and Logic Programming, in: Lecture Notes in Computer Science, vol. 8475, Springer International Publishing, 2014, pp. 221–237.
[8] P. Bahr, E. Axelsson, Associated source code repository, https://github.com/emilaxelsson/ag-graph.
[9] P. Bahr, E. Axelsson, Generalising tree traversals to dags: exploiting sharing without the pain, in: Proceedings of the 2015 Workshop on Partial Evaluation and Program Manipulation, ACM, New York, NY, USA, Jan. 2015, pp. 27–38.
[10] P. Bahr, E. Axelsson, Generalising tree traversals to DAGs: exploiting sharing without the pain, Technical report, with full proofs, 2015, available from authors' web site http://www.diku.dk/~paba/ag-dag.pdf.
[11] P. Bahr, L.E. Day, Programming macro tree transducers, in: WGP, 2013.
[12] M. Barr, C. Wells, Toposes, Triples and Theories, 1st edition, Springer, New York, 1984.
[13] P. Bille, I.L. Gørtz, G.M. Landau, O. Weimann, Tree compression with top trees, Inf. Comput. 243 (0) (2015) 166–177, 40th International Colloquium on Automata, Languages and Programming (ICALP 2013).
[14] R. Bird, Using circular programs to eliminate multiple traversals of data, Acta Inform. 21 (3) (1984) 239–250.
[15] R. Bird, J. Gibbons, S. Mehner, J. Voigtländer, T. Schrijvers, Understanding idiomatic traversals backwards and forwards, in: Haskell, 2013.
[16] P. Buneman, M. Fernandez, D. Suciu, UnQL: a query language and algebra for semistructured data based on structural recursion, VLDB J. 9 (1) (2000) 76–110.
[17] G. Busatto, M. Lohrey, S. Maneth, Efficient memory representation of XML document trees, Inf. Syst. (ISSN 0306-4379) 33 (4–5) (2008) 456–474, Selected Papers from the Tenth International Symposium on Database Programming Languages (DBPL 2005).
[18] W. Charatonik, Automata on DAG representations of finite trees, Research report, Max-Planck-Institut für Informatik, March 1999.
[19] M. Charikar, E. Lehman, D. Liu, R. Panigrahy, M. Prabhakaran, A. Sahai, A. Shelat, The smallest grammar problem, IEEE Trans. Inf. Theory (ISSN 0018-9448) 51 (7) (July 2005) 2554–2576.
[20] A. Chlipala, Parametric higher-order abstract syntax for mechanized semantics, in: ICFP, 2008.
[21] P.J. Downey, R. Sethi, R.E. Tarjan, Variations on the common subexpression problem, J. ACM (ISSN 0004-5411) 27 (4) (1980) 758–771, http://dx.doi.org/10.1145/322217.322228.
[22] R.A. Eisenberg, D. Vytiniotis, S. Peyton Jones, S. Weirich, Closed type families with overlapping equations, in: POPL, 2014.
[23] T. Ekman, G. Hedin, The JastAdd system — modular extensible compiler construction, Sci. Comput. Program. (ISSN 0167-6423) 69 (1–3) (2007) 14–26.
[24] J. Engelfriet, Bottom–up and top–down tree transformations — a comparison, Math. Syst. Theory 9 (2) (1975) 198–231.
[25] J. Engelfriet, H. Vogler, Macro tree transducers, J. Comput. Syst. Sci. 31 (1) (1985) 71–146.
[26] B. Fila, S. Anantharaman, Running tree automata on trees and/or dags, Technical report, LIFO, 2006.
[27] J. Fokker, S.D. Swierstra, Abstract interpretation of functional programs using an attribute grammar system, in: LDTA, 2009.
[28] N. Fors, G. Cedersjö, G. Hedin, JavaRAG: a java library for reference attribute grammars, in: Proceedings of the 14th International Conference on Modularity, ACM, New York, NY, USA, ISBN 978-1-4503-3249-1, 2015, pp. 55–67.

Please cite this article in press as: P. Bahr, E. Axelsson, Generalising tree traversals and tree transformations to DAGs: Exploiting sharing without the pain, Sci. Comput. Program. (2016), http://dx.doi.org/10.1016/j.scico.2016.03.006

[29] A. Fujiyoshi, Recognition of directed acyclic graphs by spanning tree automata, Theor. Comput. Sci. 411 (38–39) (2010) 3493–3506.
[30] Z. Fülöp, H. Vogler, Syntax-Directed Semantics: Formal Models Based on Tree Transducers, Springer-Verlag, New York, 1998.
[31] J. Gibbons, Generic downwards accumulations, Sci. Comput. Program. 37 (2000) 37–65.
[32] A. Gill, Type-safe observable sharing in Haskell, in: Haskell, 2009.
[33] I. Hasuo, B. Jacobs, T. Uustalu, Categorical views on computations on trees (extended abstract), in: ICALP, 2007.
[34] G. Hedin, Reference attributed grammars, Informatica (Slovenia) 24 (3) (2000) 301–317.
[35] S. Hidaka, K. Asada, Z. Hu, H. Kato, K. Nakano, Structural recursion for querying ordered graphs, in: Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming, ACM, New York, NY, USA, 2013, pp. 305–318.
[36] P. Johann, N. Ghani, Foundations for structured programming with GADTs, in: POPL, 2008.
[37] T. Kamimura, G. Slutzki, Transductions of dags and trees, Math. Syst. Theory 15 (1) (1981) 225–249.
[38] U.P. Khedker, D.M. Dhamdhere, A generalized theory of bit vector data flow analysis, ACM Trans. Program. Lang. Syst. 16 (5) (1994) 1472–1511.
[39] U.P. Khedker, D.M. Dhamdhere, A. Mycroft, Bidirectional data flow analysis for type inferencing, Comput. Lang. Syst. Struct. (ISSN 1477-8424) 29 (1–2) (2003) 15–44.
[40] O. Kiselyov, Implementing explicit and finding implicit sharing in embedded DSLs, in: DSL, 2011.
[41] D. Knuth, Semantics of context-free languages: correction, Math. Syst. Theory 5 (2) (1971) 95–96.
[42] D.E. Knuth, Semantics of context-free languages, Theory Comput. Syst. 2 (2) (1968) 127–145.
[43] N. Kobayashi, K. Matsuda, A. Shinohara, K. Yaguchi, Functional programs as compressed data, High.-Order Symb. Comput. (2013) 1–46.
[44] M.F. Kuiper, S.D. Swierstra, Using attribute grammars to derive efficient functional programs, Technical report RUU-CS-86-16, Department of Information and Computing Sciences, Utrecht University, 1986.
[45] S. Lerner, D. Grove, C. Chambers, Composing dataflow analyses and transformations, in: POPL, 2002, pp. 270–282.
[46] J.R. Lewis, J. Launchbury, E. Meijer, M.B. Shields, Implicit parameters: dynamic scoping with static types, in: POPL, 2000.
[47] M. Lohrey, S. Maneth, The complexity of tree automata and XPath on grammar-compressed trees, Theor. Comput. Sci. (ISSN 0304-3975) 363 (2) (2006) 196–210, Implementation and Application of Automata 10th International Conference on Implementation and Application of Automata (CIAA 2005).
[48] E. Magnusson, G. Hedin, Circular reference attributed grammars — their evaluation and applications, Sci. Comput. Program. (ISSN 0167-6423) 68 (1) (2007) 21–37, http://www.sciencedirect.com/science/article/pii/S0167642307000767, Special issue on the {ETAPS} 2003 Workshop on Language Descriptions, Tools and Applications (LDTA'03).
[49] S. Maneth, G. Busatto, Tree transducers and tree compressions, in: I. Walukiewicz (Ed.), Foundations of Software Science and Computation Structures, in: Lecture Notes in Computer Science, vol. 2987, Springer, Berlin, Heidelberg, ISBN 978-3-540-21298-0, 2004, pp. 363–377.
[50] E. Meijer, M. Fokkinga, R. Paterson, Functional programming with bananas, lenses, envelopes and barbed wire, in: Functional Programming Languages and Computer Architecture, in: LNCS, vol. 523, Springer, 1991, pp. 124–144.
[51] A. Middelkoop, Inference with attribute grammars, PhD thesis, Universiteit Utrecht, Feb. 2012.
[52] O.D. Moor, K. Backhouse, S.D. Swierstra, First-class attribute grammars, Informatica 24 (2000) 2000.
[53] D.R. Morrison, PATRICIA—practical algorithm to retrieve information coded in alphanumeric, J. ACM (ISSN 0004-5411) 15 (4) (Oct. 1968) 514–534.
[54] B.C. Oliveira, W.R. Cook, Functional programming with structured graphs, in: ICFP, 2012.
[55] B.C.d.S. Oliveira, A. Löh, Abstract syntax graphs for domain specific languages, in: PEPM, 2013.
[56] S. Peyton Jones, S. Marlow, Secrets of the Glasgow Haskell Compiler inliner, J. Funct. Program. 12 (2002) 393–434.
[57] B.C. Pierce, D.N. Turner, Local type inference, ACM Trans. Program. Lang. Syst. 22 (1) (Jan. 2000) 1–44.
[58] D. Quernheim, K. Knight, Dagger: a toolkit for automata on directed acyclic graphs, in: FSMNLP, 2012.
[59] J.-C. Raoult, Problem #70: design a notion of automata for graphs, http://rtaloop.mancoosi.univ-paris-diderot.fr/problems/70.html, 2005, The RTA list of open problems.
[60] T. Rompf, M. Odersky, Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs, in: GPCE, 2010.
[61] M. Rosendahl, Abstract interpretation using attribute grammars, in: WAGA, 1990.
[62] S. Sakr, XML compression techniques: a survey and comparison, J. Comput. Syst. Sci. (ISSN 0022-0000) 75 (5) (2009) 303–322.
[63] J. Saraiva, D. Swierstra, M. Kuiper, Functional incremental attribute evaluation, in: Compiler Construction, 2000.
[64] A.M. Sloane, L.C. Kats, E. Visser, A pure embedding of attribute grammars, Sci. Comput. Program. (ISSN 0167-6423) 78 (10) (2013) 1752–1769, http://dx.doi.org/10.1016/j.scico.2011.11.005, Special section on Language Descriptions Tools and Applications (LDTA'08 & '09) & Special section on Software Engineering Aspects of Ubiquitous Computing and Ambient Intelligence (UCAml 2011).
[65] T. Uustalu, V. Vene, Coding recursion à la Mendler, in: Proceedings of 2nd Workshop on Generic Programming, 2000.
[66] M. Viera, S.D. Swierstra, W. Swierstra, Attribute grammars fly first-class, in: ICFP, 2009.
[67] H.H. Vogt, S.D. Swierstra, M.F. Kuiper, Higher order attribute grammars, in: PLDI, 1989.
[68] G. Wilson, Getting the most out of monad transformers, https://github.com/gwils/next-level-mtl-with-classy-optics (accessed on Jan. 13, 2016).
[69] A.R. Yakushev, S. Holdermans, A. Löh, J. Jeuring, Generic programming with fixed points for mutually recursive datatypes, in: ICFP, 2009.

Please cite this article in press as: P. Bahr, E. Axelsson, Generalising tree traversals and tree transformations to DAGs: Exploiting sharing without the pain, Sci. Comput. Program. (2016), http://dx.doi.org/10.1016/j.scico.2016.03.006