Linearly Ordered Attribute Grammars

with Automatic Augmenting Dependency Selection

L. Thomas van Binsbergen
Royal Holloway, University of London
ltvanbinsbergen@acm.org

Jeroen Bransen
Utrecht University
j.bransen@uu.nl

Atze Dijkstra
Utrecht University
atze@uu.nl

Abstract

Attribute Grammars (AGs) extend Context-Free Grammars with attributes: information gathered on the syntax tree that adds semantics to the syntax. AGs are very well suited for describing static analyses, code-generation and other phases incorporated in a compiler.

AGs are divided into classes based on the nature of the dependencies between the attributes. In this paper we examine the class of Linearly Ordered Attribute Grammars (LOAGs), for which strict, bounded size evaluators can be generated. Deciding whether an Attribute Grammar is linearly ordered is an NP-hard problem. The Ordered Attribute Grammars form a subclass of LOAG for which membership is tested in polynomial time by Kastens' algorithm (1980). On top of this algorithm we apply an augmenting dependency selection algorithm, allowing it to determine membership for the class LOAG. Although the worst-case complexity of our algorithm is exponential, the algorithm turns out to be efficient for practical full-sized AGs. As a result, we can compile the main AG of the Utrecht Haskell Compiler without the manual addition of augmenting dependencies.

The reader is provided with insight in the difficulty of deciding whether an AG is linearly ordered, what optimistic choice is made by Kastens' algorithm and how augmenting dependencies can resolve these difficulties.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: semantics; D.3.4 [*Programming Languages*]: compilers, code generation; F.3.1 [*Logics and meanings of programs*]: mechanical verification

Keywords attribute grammars; ordered attribute grammars; linearly ordered attribute grammars; Kastens' algorithm; augmenting dependencies; compilers; semantics; Utrecht Haskell Compiler

1. Introduction

Attribute Grammars (AGs) extend Context-Free Grammars (CFGs) with attributes at each non-terminal [11]. Attribute definitions describe computations, in terms of other attributes and terminal symbols, that gather information on (parts of) the abstract syntax tree associated with the grammar. These computations are useful to perform different kinds of static analyses and code-generation. Besides being suitable for implementing compilers, programming with AGs provides some general advantages [6]:

- AGs can be seen as a domain-specific language for tree-based computations.
- AGs relieve the programmer of the task of efficiently combining multiple computations on the same tree[1].
- Descriptions of separate computations are easily divided into coherent code-fragments, increasing the maintainability and reusability of the source code.
- Most of the trivial pieces of code can be generated, allowing the programmer to focus on exactly those pieces that require creativity and expertise.
- AGs enable declarative programming in imperative settings.

Among other tools and compilers, the Utrecht University Attribute Grammar Compiler (UUAGC)[2] and the Utrecht Haskell Compiler (UHC)[3] have been largely implemented using AGs. In both cases, the UUAGC is used to compile the AGs, generating Haskell code as output.

The UUAGC generates folds and algebras for executing the semantics of an AG from its description (source text) [16]. In case the attribute definitions are cyclic, the UUAGC relies on Haskell's lazy evaluation to remain executable, potentially leading to loops at runtime. The UUAGC can also generate *strict evaluators* for AGs that are non-circular [3, 10]. For some non-circular AGs we can determine an evaluation order statically. AGs for which this is possible form the class of *Linearly Ordered Attribute Grammars* (LOAGs). The approach of finding a static evaluation order has been introduced by Kastens in 1980 [8]. It allows the generation of evaluators that are strict, efficient and require little memory. These properties are desirable especially for large scale projects such as the UHC. However, finding a linear order for the attributes of an AG is an NP-complete problem [7].

The polynomial runtime algorithm given by Kastens orders only a subset of the LOAGs, making some optimistic choices. These choices are guided by the *dependencies* between attributes. LOAGs of considerable size are likely to contain combinations of dependencies that prevent Kastens' algorithm from finding an evaluation order. The main AG in the UHC is an example of such an AG. *Augmenting dependencies* can be used to help Kastens' algorithm

PEPM '15, January 13–14, 2015, Mumbai, India.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3297-2/15/01...$15.00.
http://dx.doi.org/10.1145/2678015.2682543

[1] A computation can be efficient in multiple regards, e.g. time and space complexity. This topic is briefly discussed in the future work section of this paper.

[2] http://www.cs.uu.nl/wiki/HUT/AttributeGrammarManual

[3] http://www.cs.uu.nl/wiki/UHC

finding the order. This approach was successfully employed in the development of the UHC and in other large AG projects found in literature [12, 14]. Finding the right combination of augmenting dependencies is not only tedious work, it also demands insight in the produced evaluation order, where knowledge on this matter is otherwise unnecessary. This paper deals with this problem, making the following contributions:

- We explain the required constructions for ordering LOAGs.
- We show why Kastens' algorithm is only suitable for a subset of LOAG.
- We show how augmenting dependencies can be found automatically.
- We present an algorithm capable of ordering all LOAGs by selecting augmenting dependencies automatically with a backtracking strategy. Although the algorithm is exponential in theory, we argue that backtracking is rare for practical AGs.

Section 2 introduces the running example of this paper and introduces AGs informally. A formal definition of AGs is given in Section 3. LOAGs are defined and examined in Section 4. In Section 5 we show why Kastens' algorithm can not find a static evaluation order for all LOAGs. Section 6 explains how augmenting dependencies are selected automatically. The most important Haskell functions of our algorithm for ordering all LOAGs with automatic augmenting dependency selection are given in Section 7.

2. Running Example

We introduce Attribute Grammars informally using a running example. Every AG consists of three constructs that we introduce one at a time: abstract syntax, attributes and semantic functions.

As a running example we consider a simplistic module system IMODULE that declares modules similarly to Haskell. Each module consists of a header and a body. The header declares the functions that constitute the interface of the module, identifying which functions the module exports. The function declarations in the header are written as type signatures. The body of the module contains the required function definitions, datatype definitions and optional unexported helper definitions. Figure 1 gives an example.

```
module BinIntTrees
  flatten :: Tree → [Int]
where
  data Tree = Bin Tree Tree
            | Leaf Int
  flatten (Leaf i)  = [i]
  flatten (Bin l r) = flatten l ++ flatten r
```

Figure 1. A simple module defined with IMODULE.

The goal of the running example is to verify that the module's body implements the interface of the module, while gathering the exported definitions. Additionally, we wish to verify that all the type signatures and datatype definitions rely only on types that are available to them. Attribute Grammars are used to define the abstract syntax of this system, gather the exported definitions and perform the static analyses for determining whether the module is valid.

2.1 Abstract Syntax

In the pipeline common to most compilers (parsing → validating → generating) AGs are typically used in the second and third phase. The *abstract syntax tree*, of which instances are generated by a parser in the form of *parse trees*, forms the basis of an AG description. The abstract syntax of a language is a context-free grammar describing the syntax without literals or keywords as terminal symbols. For example, the concrete syntax displayed in Figure 1 contains keywords **module**, **where** and **data** that are irrelevant to the *semantics* of the language. In Figure 2 a description of the abstract syntax of IMODULE is given in notation accepted by the UUAGC[4]. The constructor functions of a datatype (non-terminal `Dat`) are represented by a list of types for every constructor. We allow ourselves to simplify the representation of datatypes because a precise representation would not add to the purpose of the example.

$$\begin{array}{llllllll}
\mathbf{data}\ \mathtt{Module} & \mid \mathit{Module} & h & : [\mathtt{TySig}] & b & : \mathtt{Body} \\
\mathbf{data}\ \mathtt{Body} & \mid \mathit{Body} & ds & : [\mathtt{Dat}] & fs & : [\mathtt{Fun}] \\
\mathbf{data}\ \mathtt{TySig} & \mid \mathit{TySig} & id & : \underline{\mathit{FunId}} & ty & : [\underline{\mathit{TyId}}] \\
\mathbf{data}\ \mathtt{Dat} & \mid \mathit{Dat} & id & : \underline{\mathit{TyId}} & cons & : [[\underline{\mathit{TyId}}]] \\
\mathbf{data}\ \mathtt{Fun} & \mid \mathit{Fun} & id & : \underline{\mathit{FunId}} & def & : \underline{\mathit{FunDef}}
\end{array}$$

Figure 2. Description of the abstract syntax of IMODULE.

The abstract syntax consists of non-terminals `Module`, `Body`, `TySig`, `Dat` and `Fun`, each with a single, equally named production. Each production has a set of child nodes that are either terminal (e.g. $\underline{\mathit{TyId}}$) or non-terminal. Non-terminal `Module` has two children, the module's header and its body, identified by h and b respectively. We say that h and b are *non-terminal occurrences* of type `[TySig]` and `Body` respectively. In a parse tree every node is an *instance* of one of the non-terminals and is *derived* by one of the production rules of that non-terminal. Declarations for lists of non-terminals, e.g. `[TySig]`, are missing. The required non-terminals and productions for lists are generated by the UUAGC and shown in Figure 3.

$$\begin{array}{ll}
\mathbf{data}\ [\mathtt{TySig}] & \mid \mathit{Nil} \\
 & \mid \mathit{Cons}\ hd : \mathtt{TySig}\ tl : [\mathtt{TySig}] \\
\mathbf{data}\ [\mathtt{Fun}] & \mid \mathit{Nil} \\
 & \mid \mathit{Cons}\ hd : \mathtt{Fun}\quad tl : [\mathtt{Fun}] \\
\mathbf{data}\ [\mathtt{Dat}] & \mid \mathit{Nil} \\
 & \mid \mathit{Cons}\ hd : \mathtt{Dat}\quad tl : [\mathtt{Dat}]
\end{array}$$

Figure 3. Generated abstract syntax of IMODULE.

2.2 Attributes

The next step is to add *attributes* to the grammar. Attributes are associated with non-terminals and defined at production level. For every attribute a of non-terminal X there is one *attribute occurrence* at every occurrence of X. Similarly, we speak of *attribute instances* at the level of parse trees.

Synthesized attributes (top-down) are used to gather information and can be viewed as results of computations. *Inherited* attributes (bottom-up) are used to share information and can be viewed as parameters of a computation. Figure 4 shows the attribute declarations we use for IMODULE.

An attribute is not uniquely identified by its name. Instead, it is identified by the combination of a non-terminal, a name and a direction (inh or syn), e.g. `Module`.$err(syn)$. In the same way, an attribute occurrence is identified by the combination of a production, a node, a name and a direction, e.g. *Module*.b.$err(syn)$.

[4] The UUAGC uses datatypes and constructors to define abstract syntax. Throughout this paper we use formal language terminology and speak of non-terminals and production rules instead.

```
                -- Types from other modules
attr Module inh ts   : [TyId]
                -- The exported function definitions
            syn ex   : [(FunId, FunDef)]
                -- Whether the module is invalid
            syn err : Bool
                -- All types defined by the module
            syn ts   : [TyId]
                -- Functions declared in the header
attr Body   inh ss   : [FunId]
                -- Types from other modules
            inh ts   : [TyId]
                -- The exported function definitions
            syn ex   : [(FunId, FunDef)]
                -- Whether the body is invalid
            syn err : Bool
                -- All types defined by the module
            syn ts   : [TyId]
attr [TySig] TySig   -- Declaration for two non-terminals
            inh ts   : [TyId]
            syn ss   : [FunId]
            syn err : Bool
attr [Dat] Dat
            inh ts   : [TyId]
            syn ts   : [TyId]
            syn err : Bool
attr [Fun] Fun
            inh ss   : [FunId]
            syn ex   : [(FunId, FunDef)]
```

Figure 4. Attribute declarations for IMODULE.

2.3 Semantic functions

With the attributes in place, it is now possible to define the semantics of IMODULE. *Semantic function definitions* describe for every attribute occurrence how it is computed in terms of other attribute occurrences and terminal symbols.

The flow of information from attribute to attribute is shown in graphs. In the *dependency graphs* we use throughout this paper, an arrow $a \rightarrow b$ implies that a is used to calculate b and thus that b depends on a. In other words, our graphs are actually data flow graphs rather than dependency graphs, but the latter can be obtained by reversing all edges so we speak about dependency graphs in the rest of this paper.

Figures 8 and 9 show dependency graphs for productions *Module* and *Body*. The parent node of a production, identified by *lhs*, is positioned above its children. Each non-terminal occurrence is connected with its attribute occurrences: synthesized attributes at its right and inherited attributes at its left.

There are two types of dependencies: dependencies that appear between parent and child nodes (black) and the dependencies that appear above parent nodes or below child nodes (gray). The black dependencies are *direct dependencies*, extracted directly from the semantic functions. The gray dependencies are *induced dependencies*, i.e. dependencies between *those* attributes at some *other* production. The productions that induce the dependencies between occurrences of children h, ds, and fs are not shown.

2.3.1 Semantics

This section explains how the attributes are used to describe the semantics of IMODULE on a high-level. The explanation follows the order shown in Figure 5.

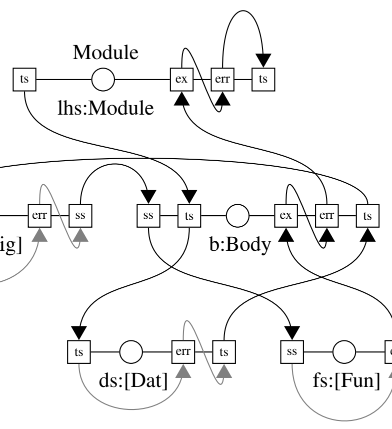

Figure 5. A linear order for evaluating the attributes of IMODULE. The order is not complete, the gray arrows represent parts that have not been fully specified. Note that this graph is not a dependency graph.

Recall that our goals are verifying that only available types are used, verifying that the module's body implements the interface and gathering the definitions of the exported functions.

First we verify that all the type signatures and datatype definitions use only available types. Types are made available by other modules ($\texttt{lhs}.ts(inh) \rightarrow \texttt{b}.ts(inh)$), e.g. Int from Prelude, or through datatype definitions in the module's body ($\texttt{b}.ts(syn)$). Child ds in *Body* uses the types arriving from other modules to verify whether its datatype declarations use only available types ($\texttt{ds}.ts(inh) \rightarrow \texttt{ds}.err(syn)$) and appends the newly constructed types to the received set, passing the new set upwards ($\texttt{ds}.ts(syn) \rightarrow \texttt{b}.ts(syn)$). The new set of types is presented to the module's header ($\texttt{b}.ts(syn) \rightarrow \texttt{h}.ts(inh)$), enabling the header to verify whether its type signatures are valid ($\texttt{h}.ts(inh) \rightarrow \texttt{h}.err(syn)$).

Secondly, we construct the list of exported definitions and test whether this list is complete. The signatures produced by the header are passed downwards into the body of the module ($\texttt{h}.ss(syn) \rightarrow \texttt{b}.ss(inh)$). *Body*'s child `fs` (function definitions) uses the signatures arriving from the header to produce a list of exports ($\texttt{fs}.ss(inh) \rightarrow \texttt{fs}.ex(syn)$), by returning only the definitions that are required according to the module's interface. The exports are passed upwards to the module declaration ($\texttt{fs}.ex(syn) \rightarrow \texttt{b}.ex(syn)$). The exported function definitions ($\texttt{lhs}.ex$) and exported types ($\texttt{lhs}.ts$) are directly copied from the body. The module reports an error ($\texttt{lhs}.err$) when the lists of exports is incomplete ($\texttt{b}.ex(syn)$ is compared with $\texttt{h}.ss(syn)$) or when b or h report an error.

2.3.2 Definitions

Dependency graphs are useful as visual aids. It might therefore be a design choice to draw these graphs before giving the actual AG description. To fully describe an AG, every dependency needs to be reflected in some semantic function definition. Some dependencies represent the flow of information from one attribute to another without modification (e.g. $\textit{Body}.\texttt{lhs}.ts(inh) \rightarrow \textit{Body}.\texttt{ds}.ts(inh)$ and $\textit{Module}.\texttt{b}.ex(syn) \rightarrow \textit{Module}.\texttt{lhs}.ex(syn)$). The semantic function definitions for such dependencies can be generated by the UUAGC based on the attribute's name and direction.

The semantic function definitions for the attributes of non-terminal lists are also generated by the UUAGC. For example, the equation for [Dat].lhs.$ts(syn)$, of type $[\,\underline{TyId}\,]$, is generated using $(+\!\!+)$ to combine results from individual Dat-elements with $[\,]$ as a base element (corresponding to Haskell's monoid instance for lists). The UUAGC allows the user to specify which union function and which base element to use for an attribute. The generated semantic function definitions are shown in Figure 7. Figure 6 appends the manual semantic function definitions to iMODULE. Occurrences of attributes and terminals are referenced using the @-symbol in the right-hand side of the equations.

```
sem Module | Module
  lhs.err = ¬ (all (∈ (map fst @b.ex)) @h.ss)
              ∨ @h.err ∨ @b.err
  h.ts    = @b.ts
  b.ss    = @h.ss
sem TySig | TySig
  lhs.ss  = [@id]
  lhs.err = ¬ (all (∈ @lhs.ts) @ty)
sem Dat | Dat
  lhs.ts  = @id : @lhs.ts
  lhs.err = ¬ (all (all (∈ @lhs.ts)) @cons)
sem [Dat] | Cons
  lhs.ts  = @tl.ts
  tl.ts   = @hd.ts
sem Fun | Fun
  lhs.ex  = if (@id ∈ @lhs.ss)
              then [(@id, @def)]
              else []
```

Figure 6. The manual semantic function definitions for the semantics of iMODULE.

2.3.3 Reflection

We have described semantics, useful in the second phase of a compiler for iMODULE, in a small number of simple steps. The AG computing the semantics of our system is orderable, while Kastens' algorithm is not able to recognise it as such, as we shall see in Section 5. Section 6 explains the use of augmenting dependencies and shows which augmenting dependency successfully hints at the order proposed in Figure 5 and how the augmenting dependency is found.

3. Attribute Grammars

This section formalises AGs and other concepts required in subsequent sections.

Definition 1. *An* Attribute Grammar *(AG) is a triple* $\langle G, A, E\rangle$*, where* $G = \langle V = \Sigma \cup N, P, S\rangle$ *is a context-free grammar.* V *is partitioned into a set of terminal symbols* Σ *and a set of non-terminal symbols* N*.* P *is a non-empty set of* productions*, with* $p \in P$ *of the form* $p : X_{p,0} \rightarrow \alpha_1 X_{p,1}, \alpha_2 X_{p,2}, \ldots, \alpha_{|p|} X_{p,|p|} \alpha_{|p|+1}$*, with* $\alpha_i \in \Sigma^*$ *and* non-terminal occurrences $X_{p,i}$*. A non-terminal occurrence* $X_{p,i}$ *is an occurrence of non-terminal* $X \in N$ *iff* $\mathcal{T}(X_{p,i}) = X$*.* $S \in N$ *is the start symbol of the grammar.* A *is a set of* attributes $(X \cdot a)$*, with* $X \in N$ *and* a *an* attribute identifier*.* A *is divided into sets of inherited and synthesized attributes for every* $X \in N$*, i.e.* $A_{inh}(X)$ *and* $A_{syn}(X)$ *respectively.* $(X \cdot a)$ *denotes that the attribute identified by* a *is associated with non-terminal* X*.* E *is the set of* semantic function definitions*, with* $(X_{p,i} \cdot a, \lambda) \in E$ *denoting that* λ *is the definition for the* attribute occurrence $(X_{p,i} \cdot a)$.

```
sem Module | Module b.ts   = @lhs.ts
                    lhs.ts = @b.ts
                    lhs.ex = @b.ex
sem Body   | Body   ds.ts   = @lhs.ts
                    fs.ss   = @lhs.ss
                    lhs.ex  = @fs.ex
                    lhs.ts  = @ds.ts
                    lhs.err = @ds.err
sem [TySig] | Nil   lhs.err = False
                    lhs.ss  = []
            | Cons  lhs.err = @hd.err ∨ @tl.err
                    lhs.ss  = @hd.ss ++ @tl.ss
                    hd.ts   = @lhs.ts
                    tl.ts   = @lhs.ts
sem [Dat]   | Nil   lhs.err = False
                    lhs.ts  = @lhs.ts
            | Cons  lhs.err = @hd.err ∨ @tl.err
                    hd.ts   = @lhs.ts
sem [Fun]   | Nil   lhs.ex  = []
            | Cons  lhs.ex  = @hd.ex ++ @tl.ex
                    hd.ss   = @lhs.ss
                    tl.ss   = @lhs.ss
```

Figure 7. The generated semantic function definitions for the semantics of iMODULE.

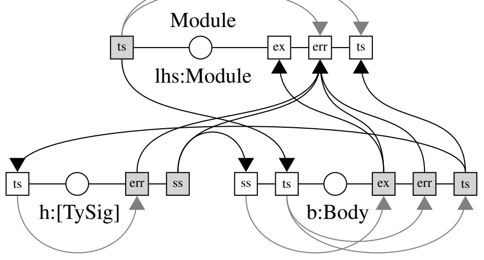

Figure 8. Induced dependency graph for production *Module*. Direct dependencies are black, induced dependencies are gray.

3.1 Input and output dependencies

The attribute occurrences of production $p \in P$ are divided into *input* and *output* occurrences. The input occurrences of p, denoted with $O_{inp}(p)$, are the occurrences made *available* to p by the context of p. Output occurrences of p, denoted with $O_{out}(p)$, are the occurrences that the semantic functions of p *deliver* to the context of p [13, 15].

$$O_{inp}(p) = \{\, X_{p,0} \cdot a \mid (X \cdot a) \in A_{inh}(\mathcal{T}(X_{p,0})) \,\} \cup \{\, X_{p,i} \cdot a \mid i > 0,\ (X \cdot a) \in A_{syn}(\mathcal{T}(X_{p,i})) \,\} \quad (1)$$

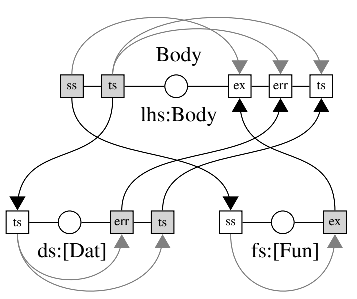

Figure 9. Induced dependency graph for production *Body*. Direct dependencies are black, induced dependencies are gray.

$$\begin{aligned} O_{out}(p) = &\{\ X_{p,0} \cdot a \ \mid\ (X \cdot a) \in A_{syn}(\mathcal{T}(X_{p,0}))\ \} \ \cup \\ &\{\ X_{p,i} \cdot a \ \mid\ i > 0,\ (X \cdot a) \in A_{inh}(\mathcal{T}(X_{p,i}))\ \} \end{aligned} \tag{2}$$

The dependency graphs in this paper show input occurrences with gray backgrounds and output occurrences with white backgrounds.

In this paper we consider AGs that are *normalised*: only input occurrences of a production can be used in the right-hand side of semantic function definitions for that production and only output occurrences of a production are considered to have semantic function definitions. The class of AGs we consider is therefore a subset of the class of AGs written in Bochmann Normal Form as only the first of the two restrictions is used to define this class [2].

3.2 Direct dependencies

From the semantic functions we extract the *direct dependencies*. We define the set $SF_P(b)$ to be the occurrences referenced in the right-hand side of the semantic function of b. Using SF_P we formalise the *direct dependency graph* of production p:

$$\begin{aligned} D_P(p) = \{\ &(X_{p,i} \cdot a \to X_{p,j} \cdot b) \\ \mid\ &(X_{p,j} \cdot b \in O_{out}(p)) \\ ,\ &(X_{p,i} \cdot a \in SF_P(X_{p,j} \cdot b))\ \} \end{aligned} \tag{3}$$

4. Linearly Ordered Attribute Grammars

We are interested in finding an evaluation order of an AG's attributes statically, as the order can be used to generate simple and efficient evaluators. The Linearly Ordered Attribute Grammars form the largest class of AGs for which this is possible [7, 15].

Definition 2. *An AG* $= \langle G, A, D \rangle$*, with context-free grammar* $G = \langle \Sigma \cup N, P, S \rangle$*, is a* Linearly Ordered Attribute Grammar *or* LOAG*, if there exist linear orders* $LO(p)$ *for all* $p \in P$ *such that:*

- *Every linear order* $LO(p)$ *respects the direct dependencies, i.e. if* $(X_{p,i} \cdot a \to X_{p,j} \cdot b) \in D_P(p)$ *then* $(X_{p,i} \cdot a < X_{p,j} \cdot b) \in LO(p)$*.*
- *The relative ordering of the attributes is the same for all occurrences of a non-terminal, i.e. if* $(X_{p,i} \cdot a < X_{p,i} \cdot b) \in LO(p)$ *then* $(X_{q,j} \cdot a < X_{q,j} \cdot b) \in LO(q)$ *for all p, q, i and j with* $\mathcal{T}(X_{p,i}) = \mathcal{T}(X_{q,j})$*.*

From the linear orders on productions we can obtain a linear order $LO(\delta)$ for any valid parse tree δ of the input AG. A strict evaluator can be generated that evaluates all attribute instances of δ in the order specified by $LO(\delta)$. Since we are interested in finding the linear order statically we can only argue about non-terminals, attributes, productions and attribute occurrences.

LOAGs have been a popular subject in AG literature, although defined slightly differently in several instances [1, 7, 13, 15]. The subclass OAG, for which Kastens' algorithm can generate evaluators in polynomial runtime, has been more popular in practical implementations due to the complexity of generating evaluators for LOAGs. This paper shows that LOAGs are useful in practice too, by giving an algorithm that is efficient for practical LOAGs. The definition of LOAGs given here supports this purpose and allows simple comparison with Kastens' OAG.

4.1 LOAG preconditions

A linear order that respects all direct dependencies can only exist if there are no dependency cycles. An acyclic direct dependency graph is a precondition for LOAGs.

Every parse tree is the product of 'gluing' multiple production rules together [11]. We therefore also have to test for any dependency cycles produced by 'gluing' productions together. For that purpose we introduce the notion of an *induced dependency graph*. An acyclic induced dependency graph is a second precondition for LOAGs.

4.2 Induced dependencies

If there is a path between two attribute occurrences $(X_{p,i} \cdot a)$ and $(X_{p,i} \cdot b)$ then there has to be a dependency $(X_{q,j} \cdot a \to X_{q,j} \cdot b)$, for all $X_{q,j}$ with $\mathcal{T}(X_{q,j}) = \mathcal{T}(X_{p,i})$, in order to take all possible ways in which production rules can be 'glued' together into account. We have to be pessimistic and consider all such *induced dependencies*.

Adding induced dependencies might result in new paths between two attributes of a non-terminal occurrence and hence to more induced dependencies. See Figure 10 for an example of how induced dependencies are propagated.

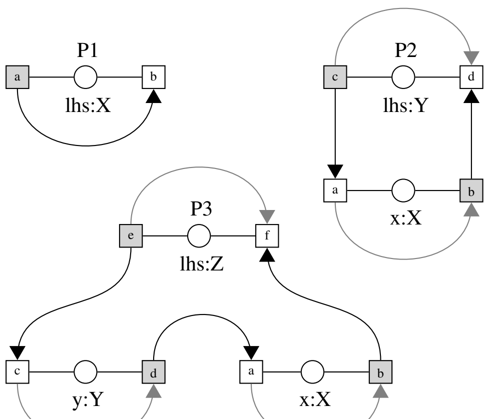

Figure 10. Dependency $(a \to b)$ in P1 induces a dependency $(a \to b)$ in P2 and P3, which causes a path from c to d in P2 and thus an induced dependency $(c \to d)$ in P2 and P3. The new path from e to f in P3 induces a dependency $(e \to f)$ in P3.

In any induced dependency graph all the occurrences of the same non-terminal (e.g. `Body` in productions *Module* and *Body* of Figures 8 and 9) display the same dependencies between its attributes. These shared dependencies are collected at non-terminal level in the graph ID_S (left-hand side of Figure 17)[5]. To recognise paths of dependencies we use ID_P^+, the transitive closure of ID_P.

[5] Kastens introduces ID_S as the induced dependency graph for symbols (hence the S), although only non-terminal symbols have attributes.

$$ID_P(p) = D_P(p) \;\cup \{\, (X_{p,i} \cdot a \rightarrow X_{p,i} \cdot b) \;\mid\; q \in P \\ ,\; (X_{q,j} \cdot a \rightarrow X_{q,j} \cdot b) \in ID_P^{+}(q) \\ ,\; (\mathcal{T}(X_{p,i}) = \mathcal{T}(X_{q,j})) \,\} \tag{4}$$

$$ID_S(X) = \{\, (X \cdot a \rightarrow X \cdot b) \;\mid\; p \in P, \\ (X_{p,i} \cdot a \rightarrow X_{p,i} \cdot b) \in ID_P(p),\; X = \mathcal{T}(X_{p,i}) \,\} \tag{5}$$

4.3 Interfaces and visit-sequences

To find a linear order on attribute instances of any parse tree, we first determine in which order the generated evaluator will examine the nodes of any parse tree. We do so by deciding for every non-terminal which *visits* are made to it and in which order. Every visit is a pair of inherited and synthesized attributes that can be seen as a function that receives the inherited attributes as parameters and returns the synthesized attributes as a result.

Definition 3. *An* interface $I_f(X)$ *for non-terminal* X *is a sequence of* n *visits, i.e.* $I_f(X) = (v_i)_{i=1}^n$, *where every visit* v_i *is a pair* (I_i, S_i), *with* $I_i \subseteq A_{inh}(X)$ *and* $S_i \subseteq A_{syn}(X)$. *The interface must be complete and the visits disjoint:*

$$\bigcup_{(I_i,S_i)\in I_f(X)} I_i = A_{inh}(X) \qquad \bigcup_{(I_i,S_i)\in I_f(x)} S_i = A_{syn}(X) \tag{6}$$

$$\forall((I_i,S_i) \in I_f(X), (I_j,S_j) \in I_f(X), i \neq j)\ (I_i \cap I_j = \emptyset) \\ \forall((I_i,S_i) \in I_f(X), (I_j,S_j) \in I_f(X), i \neq j)\ (S_i \cap S_j = \emptyset) \tag{7}$$

Reflecting on Figure 5 we see that all nodes are visited once by the linear order we proposed, except node `b` of type `Body` which is visited twice. The first visit to `b` is $(\{ts\},\{ts\})$ and the second is $(\{ss\},\{ex, err\})$. These visits make up the interface for non-terminal `Body` shown in Figure 11.

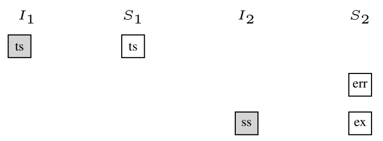

Figure 11. A possible interface for non-terminal `Body`.

Visits are connected by *visit-sequences*. For every non-terminal X and for every visit $v_i \in I_f(X)$, we show how every production of X *implements* visit v_i with a visit-sequence. A visit-sequence is a sequence of eval- and visit-instructions. The generated evaluator executes these instructions in the order specified by the visit-sequence. Every eval-instruction is associated with an attribute occurrence and tells the evaluator to execute the semantic function definition of that attribute occurrence. Every visit-instruction is associated with a non-terminal occurrence K and a visit number i, and tells the evaluator to execute the visit-sequences implementing the i-th visit to K. For every visit-sequence s in a valid set of visit-sequences - with s implementing visit $v_i = (I_i, S_i)$ to an occurrence K of non-terminal X, derived by production p - the following conditions must hold:

1. Before s is executed, every j-th visit to K, with $j < i$, must be executed.
2. Before s is executed, every occurrence of attributes I_i at K must be evaluated.
3. Every occurrence of attributes S_i at the parent of p must be evaluated in s.
4. If occurrence b, depending on a, is evaluated in s:
   (a) Then a must be evaluated in s before b,
   (b) or there must be a visit-instruction in s, before b is evaluated, of which the corresponding visit-sequence evaluates a.

Figures 12 and 13 show visit-sequences for productions ***Module*** and ***Body*** that encode the linear order proposed in Figure 5. The first three conditions on a valid set of visit-sequences can easily be checked using these figures:

1. There is no visit-sequence in which there is a **visit** 2 before a **visit** 1.
2. Every **visit** is preceded by the evaluation of the inherited attributes of that visit (the parameters of that computation).
3. Every visit-sequence evaluates the synthesized attributes of the implemented visit (the results of that computation).

1 : **eval** `b`.ts
2 : **visit** 1 `b`
3 : **eval** `h`.ts
4 : **visit** 1 `h`
5 : **eval** `b`.ss
6 : **visit** 2 `b`
7 : **eval** `lhs`.ex
8 : **eval** `lhs`.err
9 : **eval** `lhs`.ts

Figure 12. A visit-sequence for the production ***Module*** of non-terminal `Module` that implements visit $(\{ts\},\{ex, err, ts\})$.

1 : **eval** `ds`.ts
2 : **visit** 1 `ds`
3 : **eval** `lhs`.ts

1 : **eval** `fs`.ss
2 : **visit** 1 `fs`
3 : **eval** `lhs`.ex
4 : **eval** `lhs`.err

Figure 13. Two visit-sequences for production ***Body*** of non-terminal `Body` separated by a line break. The top visit-sequence implements visit $(\{ts\},\{ts\})$ and the bottom visit-sequence implements visit $(\{ss\},\{ex, err\})$.

The fourth condition can be checked by looking at the individual linear orders that the visit-sequences encode: no dependency in the induced dependency graphs given in Figures 8 and 9 is contradicted by the linear orders shown in Figures 14 and 15.

Similarly to the way production rules are 'glued' together to form a valid parse tree δ of the input grammar, the linear orders encoded by visit-sequences can be 'glued' together to form a linear order on δ [13]. Such 'gluing' is only possible if all visit-sequences rely on the same set of interfaces, one for every non-terminal. This explains the need for constructing interfaces and the second requirement of Definition 2. Figure 16 shows how the orders of Figures 12 and 13 are 'glued' together. The result is the same order as the one shown in Figure 5.

We can now describe a high-level algorithm for LOAGs:

- Test the two preconditions for LOAGs.
- Only if they hold:

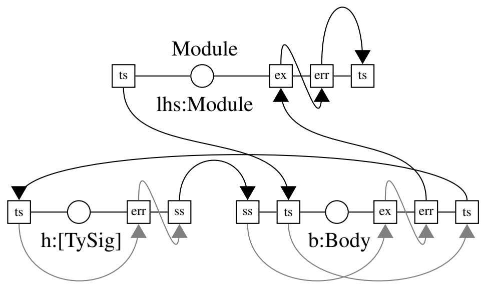

Figure 14. The linear order encoded by the visit-sequence of Figure 12. The gray arrows represent the linear orders of other visit-sequences.

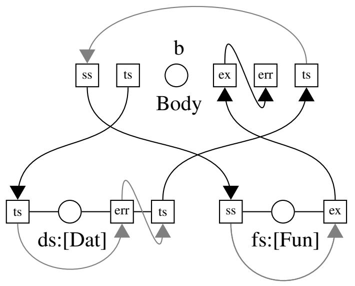

Figure 15. The linear order encoded by the visit-sequences of Figure 13. The gray arrows represent the linear orders of other visit-sequences.

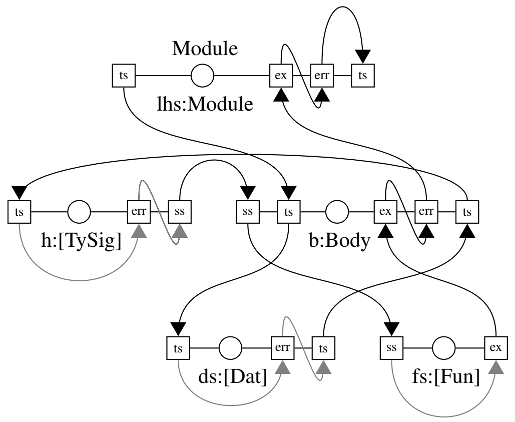

Figure 16. Linear orders encoded by visit-sequences can be 'glued' together if the visit-sequences adhere the same interfaces for their shared non-terminals. Note that the gray arrows around node b of type Body in Figures 14 and 15 have been replaced by paths.

- Use graph ID_S to find interfaces for all non-terminals.
- Use the interfaces and graph ID_P to implement all visits of the interfaces using visit-sequences.

Kastens' algorithm is an implementation of this high-level algorithm. The next section shows how Kastens' algorithm finds the interfaces and why this method does not suffice for all LOAGs.

5. Kastens' algorithm

In his 1980 paper, Kastens presents a polynomial algorithm for deciding whether an AG is an *Ordered Attribute Grammar* (OAG), a subclass of LOAG [8]. OAG is a proper subclass of LOAG[6], hence every OAG is an LOAG but there exist LOAGs that are not an OAG [3, 12].

5.1 Constructing interfaces

Kastens' algorithm constructs $I_f(X)$ from $ID_S(X)$ by partitioning the attributes of X in disjoint sets P_i, with $i \geqslant 1$, as follows:

- Assign all synthesized attributes that have no outgoing dependencies to P_1.
- Assign all inherited attributes that have no outgoing dependencies to P_2.
- Assign all synthesized attributes a to the set P_i if i is odd, all the inherited attributes that depend on a are assigned to P_j with $j < i$ and all synthesized attributes that depend on a are assigned to P_k with $k \leqslant i$.
- Assign all inherited attributes a to the set P_i if i is even, all the synthesized attributes that depend on a are assigned to P_j with $j < i$ and all inherited attributes that depend on a are assigned to P_k with $k \leqslant i$.

Note that the set partitioning constructed by the above procedure is almost of the same shape as our interfaces, except that we pair P_i with P_{i-1} to form a visit (if i is even) and our indices are ascending with respect to the dependencies: if attributes a and b are in different visits and $(a \to b) \in ID_S$ then a is assigned to a visit with lower index then b.

Interfaces are thus constructed from the partitioning as follows: If m is the highest even index with $P_m \cup P_{m-1} \neq \emptyset$, then $v = m/2$ is the number of visits in $I_f(X)$. $I_f(X)$ is formed by saying (P_m, P_{m-1}) is the first visit, (P_{m-2}, P_{m-3}) is the second visit, up until the v-th visit (P_2, P_1).

Figure 17 shows the interfaces calculated by Kastens' algorithm for non-terminals [TySig] and Body of iMODULE. The right-hand side of the figure shows the first (and only) visit (I_1, S_1) of the interfaces for both non-terminals. The induced dependencies from ID_S have been added to the picture (solid gray). It shows that ID_S can be split into connected components that we call *threads* (T_1 and T_2).

Interfaces must satisfy the requirement that the induced dependencies do not point 'in the wrong direction' (westwards in our figures). Interfaces that do not satisfy this requirement do not allow visit-sequences that satisfy condition 4 of the conditions on visit-sequences (Section 4.3).

Kastens' method for creating interfaces ensures that no dependency points westwards; however, there is still freedom in splitting visits and moving attributes across visits in the interface. For example, the attributes of T_2 are assigned to the same visit as the attributes of T_1, although the induced dependencies do not enforce this. If we split the visit of non-terminal Body such that the attributes Body.$ts(inh)$ and Body.$ts(syn)$ are in an earlier visit, we would get another interface (the one from Figure 11), in which no induced dependency will point westward as well. As we shall see later, the combination of the interfaces from Figure 17 is invalid and it is impossible to discover this by looking at the individual interfaces.

Deciding which interfaces to select is a combinatorial problem and constitutes the NP-hardness of finding a linear order. When the

[6] The removal of the adverb linearly suggests that the ordered attribute grammars form a greater class even though they form a strict subclass. We decide to stick with the established terminology however.

interface contains multiple threads with multiple visits, many different interfaces without arrows pointing westwards are possible. Kastens' algorithm optimistically decides to construct the interface containing the smallest number of visits. The next section shows that this choice is not always correct.

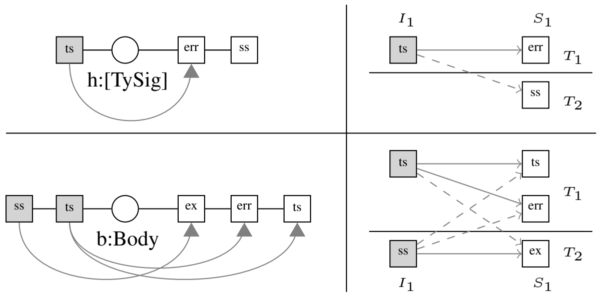

Figure 17. Graphs ID_S (left) and interfaces (right) with induced intra-visit dependencies (gray) and non-induced intra-visit dependencies (dashed gray) for `[TySig]` (top) and `Body` (bottom).

5.2 Intra-visit dependencies

By fixing the interfaces we introduce a third type of dependency, the *intra-visit dependency*, that has to be taken into account at condition 4 of the conditions on valid sets of visit-sequences (Section 4.3). From condition 2 it follows that every synthesized attribute of a visit v_i depends on all inherited attributes of v_i. From condition 1 it follows that all attributes of v_i depend on all attributes of visit v_j when $j < i$. Given $I_f(X)$, the interface of X, we gather the intra-visit dependencies of X in the set $IVD(X)$:

$$\begin{aligned} IVD(X) = \{\, & X\cdot a \to X\cdot b \mid (I_i, S_i) \in I_f(X), \\ & X\cdot a \in I_i,\ X\cdot b \in S_i \,\} \\ & \cup \\ \{\, & X\cdot a \to X\cdot b \mid (I_{i-1}, S_{i-1}) \in I_f(X), \\ & (I_i, S_i) \in I_f(X), \\ & X\cdot a \in S_{i-1},\ X\cdot b \in I_i \,\} \end{aligned} \tag{8}$$

All induced dependencies are represented by the intra-visit dependencies, i.e. $ID_S(X) \subseteq IVD^+(X)$. However, some intra-visit dependencies might not be induced (dashed gray in Figure 17).

The intra-visit dependencies imposed by the interfaces might produce cycles together with the direct dependencies at production level, making it impossible to construct a set of valid visit-sequences. Selecting a combination of interfaces for all nonterminals simultaneously such that no cycles occur is a combinatorial problem as fixing one interface might impose restrictions on the other interfaces.

5.3 Ordered Attribute Grammars

An AG is an LOAG if there exist interfaces whose intra-visit dependencies do not lead to cycles with the direct dependencies. An AG is an OAG if Kastens' algorithm finds these interfaces.

Definition 4. *An AG is an* Ordered Attribute Grammar *or* OAG *iff the graph ED_P is acyclic with $IVD(X)$ based on $I_f(X)$ calculated by Kastens' algorithm.*

$$\begin{aligned} ED_P(p) = & D_P(p) \cup \\ & \{\, (X_{p,i}\cdot a \to X_{p,i}\cdot b) \\ & \mid\ (X\cdot a \to X\cdot b) \in IVD(X),\ X = \mathcal{T}(X_{p,i}) \,\} \end{aligned} \tag{9}$$

It is important to note that the definition above would be a definition for LOAGs if it had not stated that the interfaces are calculated by Kastens' algorithm (shown in Section 5).

The interfaces of Figure 17 are the ones constructed by Kastens' algorithm. These interfaces impose the intra-visit dependencies (`Body`.$ss(inh)$ $\to$ `Body`.$ts(syn)$) and (`[TySig]`.$ts(inh)$ $\to$ `[TySig]`.$ss(syn)$) (among others). Together with the direct dependencies (`h`.ss $\to$ `b`.ss) and (`b`.ts $\to$ `h`.ts) of production ***`Module`*** these intra-visit dependencies generate the cycle (`h`.ts $\to$ `h`.ss $\to$ `b`.ss $\to$ `b`.ts $\to$ `h`.ts) shown in Figure 18. The AG description given for IMODULE is therefore *not* an OAG. However, if the interface of `Body` is split into two visits there is no dependency cycle. We have given this interface in Figure 11. Figure 14 shows that there is no dependency cycle using this interface.

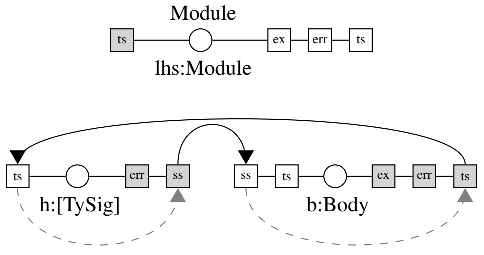

Figure 18. The intra-visit dependencies imposed by the interfaces calculated by Kastens' algorithm form a cycle with the direct dependencies in production ***`Module`***.

In the next section we show that Kastens' algorithm can be forced to find this interface by adding an augmenting dependency.

6. Augmenting Dependencies

In his 1980 paper, Kastens introduces another class that trivially equals LOAG.

Definition 5. *An AG is an* Arranged Orderly Attribute Grammar (AOAG) *if it is recognised as an OAG by extending the set of direct dependencies with a set of* augmenting dependencies *called ADS.*

Every AOAG is an LOAG because the AOAG is recognised as an OAG with the right set of augmenting dependencies (proving we can construct the required linear orders). Every LOAG is an AOAG because for the LOAG there exists a set of interfaces that impose a set of intra-visit dependencies that cause no cycles. These intra-visit dependencies can be taken as augmenting dependencies and added to the AG description to recognise the LOAG as an OAG. Therefore the classes are equal.

6.1 Manually adding augmenting dependencies

A reference to an attribute can be added to the right-hand side of a semantic function definition, without changing the semantics of that expression, using conditional expressions. Add the attribute a of the augmenting dependency $(a \to b)$ to the semantic function definition of b as the **else**-branch of an **if-then-else** expression with a guard that is always `True`. The **then**-branch contains the old semantic function definition. This method is being used frequently in practice, for example in [3, 12, 14]. A difficulty of this approach is that the resulting AG may not be well-typed.

Augmenting dependencies can also be made explicit by adding syntax to the AG compiler (as has been done in the UUAGC) for the special purpose of extending D_P to contain the set of augmenting dependencies. This way the types of the attributes are no longer a concern and arbitrary augmenting dependencies can be added.

$$D'_P(p) = D_P(p) \cup \{ \ (X_{p,i} \cdot a \to X_{p,i} \cdot b) \\ \mid \ (X \cdot a \to X \cdot b) \in ADS) \\ , X = \mathcal{T}(X_{p,i}) \ \} \tag{10}$$

Adding augmenting dependencies by hand is only a solution for small AGs that are not likely to be changed. Although the method has been used to compile the UHC, it requires a great deal of trial-and-error to find the right augmenting dependencies for such a large AG description. Moreover, adding (and also removing) static analyses often requires a revision of the set of augmenting dependencies.

6.2 Automatically adding augmenting dependencies

We consider the following problem: "*given an AG that is not an OAG* (there is at least one dependency cycle) *that does satisfy the LOAG preconditions* (there are no direct or induced dependency cycles), *find a set of augmenting dependencies that show the AG is an AOAG*".

If there is a dependency cycle while the LOAG preconditions hold, that dependency cycle must contain some intra-visit dependencies that are not part of ID_S (or there was a cycle containing only direct or induced dependencies). We can prevent this dependency cycle by selecting the reverse of one of these intra-visit dependencies as an augmenting dependency.

In our example, Kastens' algorithm can be forced to generate the interface shown in Figure 11 for non-terminal `Body` by adding the augmenting dependency `b`.$ts(syn)$ $\to$ `b`.$ss(inh)$. Figure 19 shows how the augmenting dependency is used to force a different relative ordering for the threads of non-terminal `Body`. Recall that in the figures depicting interfaces no dependencies may point westward.

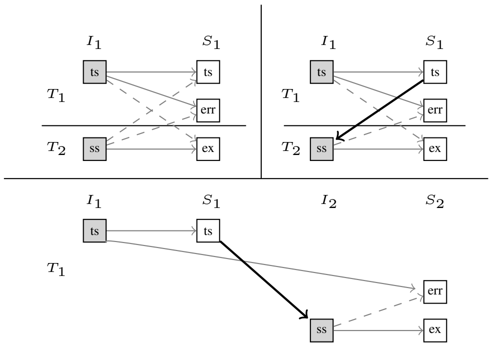

Figure 19. Choosing the augmenting dependency $ts(syn) \to ss(inh)$ results in a different interface for non-terminal `Body`. Top-left: the old interface. Top-right: the old interface with the augmenting dependency highlighted. Bottom: the new interface.

6.2.1 Selecting augmenting dependencies

The goal of an augmenting dependency is to impose a new dependency between attributes of the same non-terminal, such that the threads calculated for that non-terminal are different, leading to a different interface and a different set of imposed intra-visit dependencies.

We argue that if there exists some set of augmenting dependencies that prove an AG is ordered, then we can prove that the AG is ordered using only intra-visit dependencies not in ID_S. We find the set with a backtracking algorithm. As *candidates* to our selection procedure we only consider the reverses of non-induced intra-visit dependencies (the dashed gray dependencies). Moreover, these intra-visit dependencies should be part of a dependency cycle.

Since the problem is NP-hard, our set of candidates can not always be perfect: it may contain dependencies that, by adding them to the direct dependencies, lead to a cyclic ID_P (the preconditions for LOAGs no longer hold). Let ID_P be the induced dependency graph before adding $(a \to b)$ as an augmenting dependency and ID'_P afterwards. We examine which dependencies in ID_P will lead to cycles in ID'_P:

1. A dependency $(b \to a) \in ID_P$.
2. A dependency $(b' \to a') \in ID_P$ causing a cycle with $(a' \to b') \in ID'_P$ that is imposed as an induced dependency from the new dependency $(a \to b)$.
3. A dependency $(c \to d) \in ID_P$ causing a cycle with $(d \to c) \in ID'_P$ that is imposed as an induced dependency, through a series of steps, by the new dependency $(a \to b)$.
4. Both $(c \to d) \in ID'_P$ and $(d \to c) \in ID'_P$ are induced by $(a \to b)$.

In the first three cases, the augmenting dependency is not a candidate. In the fourth case, the augmenting dependency must induce a dependency, and its reverse, simultaneously.

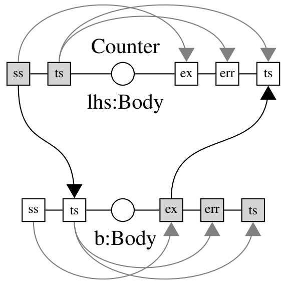

Figure 20. The choice of (`Body.b.`ts(syn) $\to$ `Body.b.`ss(inh)) as an augmenting dependency results in a cycle in production *`Counter`* that is unresolvable, as it induces both the dependency (*`Counter`*`.lhs.`ts(inh) $\to$ *`Counter`*`.lhs.`ex(syn)) and the dependency (*`Counter`*`.b.`ex(syn) $\to$ *`Counter`*`.b.`ts(inh)).

From the cycle in Figure 18 we extract candidates (`b`.$ts(syn)$ $\to$ `b`.$ss(inh)$) and (`h`.$ss(syn)$ $\to$ `h`.$ts(inh)$) which both successfully alter the interfaces such that IMODULE is recognised as an OAG. If we add production *`Counter`* (shown in Figure 20) to the AG description, we observe that selecting candidate (`b`.$ts(syn)$ $\to$ `b`.$ss(inh)$) induces both (`b`.$ex(syn)$ $\to$ `b`.$ts(inh)$) and (`lhs`.$ts(inh)$ $\to$ `lhs`.$ex(syn)$) in *`Counter`*. Graph ID_P becomes cyclic and backtracking is required for the algorithm to choose (`h`.ss $\to$ `h`.ts) as an augmenting dependency instead. This counter-example to our selection procedure is both contrived and type incorrect. Any AG which forces our algorithm to backtrack will contain constructions similar to the one described and we expect that such dependency combinations are rare in practical AGs. Our claim is supported by the tests we ran as no backtracking was required for any of the UHC's AG descriptions (more about this in Section 8).

7. AOAG Algorithm

We have written the functions using our own $AOAG$ monad built on top of Haskell's ST monad. The code given in this section is a pseudo version of our actual implementation in the UUAGC.

7.1 Calculating interfaces

Function *aoag* is the main function of the AOAG algorithm[7] and is given in Figure 21.

```
    -- receives direct dependency graph $D_P$
$aoag$   :: $Graph\ s \to AOAG\ s\ (Maybe\ (Graph\ s))$
$aoag\ dp$ = do
      -- induced dependencies, Section 4.2
  $m\_ids \leftarrow induced\ dp$
  case $m\_ids$ of
    $Nothing \to return\ Nothing$   -- cycle in $ID_P$
    $(idp, ids) \to$ do
        -- construct set partitioning, Figure 22
        -- by calling partition on all non-terminals
      $itfs$              $\leftarrow partition\_nts\ ids$
        -- all ivds and non-induced ivds, Section 5.2
      $(ivds, ni\_ivds) \leftarrow intras\ itfs$
      $mc$                $\leftarrow oagtest\ dp\ ivds$
      case $mc$ of   -- did oagtest find a cycle in $ED_P$?
          -- no, it is an OAG and we return the interfaces
        $Nothing \to return\ (Just\ itfs)$
          -- yes, start backtracking algorithm
        $Just\ c \to explore\ c\ dp\ idp\ ids\ itfs\ ni\_ivds$
```

Figure 21. The main function of the AOAG algorithm

The function returns a set of interfaces which can be used together with function *intras* and D_P to calculate ED_P from Definition 4 (not shown). Graph ED_P is used to calculate the visit-sequences exactly as in Kastens' algorithm and Pennings [8, 13]. Note that we do not convert the set partitionings to interfaces and use the set partitionings directly (even though we refer to them as interfaces).

The dependency graphs (of type *Graph s*) are *STRef*s that point to the graph hidden in state behind the usual graph operations. Since we use pointers, we have to make copies of ID_P and the interfaces to allow backtracking.

Function *partition* in Figure 22 calculates the interfaces according to Kastens' algorithm by assigning every attribute a to a set partition P_i (see Section 5) based on the successors of a according to $ID_S(X)$.

7.2 Backtracking algorithm

Backtracking is performed by function *explore*, given in Figure 23. It uses function *insertCandidate* that inserts a dependency a to the induced dependency graph in such a way that all the effects of choosing a as a candidate are propagated through ID_P and ID_S. If a cycle is encountered *Nothing* is returned, otherwise all the new induced dependencies imposed by a are returned. The new induced dependencies may demand a rectification of the interfaces. The interfaces are rectified by increasing the partition index of certain attributes until no more dependencies are pointing 'in the wrong direction'. An example of rectification is shown in Figure 19 and the function *reschedule* is shown in Figure 24. Function *depOccs* (of type $Edge \to [Edge]$) transforms dependencies from non-terminal level to production level by generating all the occurrences of the given dependency.

[7] We have named the algorithm after the class AOAG because the algorithm adds augmenting dependencies to the direct dependencies of the input AG. In the future work section we discuss an algorithm that solves the problem more directly and refer to this algorithm as the LOAG algorithm.

```
$partition :: Graph\ s \to Nonterminal \to AOAG\ s\ ()$
$partition\ ids\ x$ = do
  $as \leftarrow attributes\ x$          -- $A(X)$
  $forM\_\ as\ (assign\ ids)$   -- attempt to assign all
 where
  $assign :: Graph\ s \to Vertex \to AOAG\ s\ ()$
  $assign\ ids\ a$ = do
    $dir$         $\leftarrow direction\ a$    -- synthesized or inherited
    $mnr$         $\leftarrow partition\_ix\ a$   -- has a been assigned?
    $when\ (isNothing\ mnr)$ \$ do   -- only if not:
          $succs \leftarrow successors\ ids\ a$
          case $succs$ of    -- has no successors
            $[\,]$  $\to$ let  $part$   | $dir \equiv Syn = 1$
                                | $dir \equiv Inh = 2$
                    in   $wrap\_up\ a\ part$   -- assign
            $ss$  $\to$ do
                -- returns the highest partition index to
                -- which an successors has been
                -- assigned, or Nothing if not all
                -- successors have been assigned yet
              $mmax\_part \leftarrow max\_partition\ ss$
                -- can we assign $a$ yet?
              case $mmax\_part$ of
                $Nothing \to return\ ()$   -- no
                  -- attribute b, partition index mx
                $Just\ (b, mx) \to$            -- yes
                      -- rules of Section 5.1
                  let $dir\_b$ | $even\ mx = Inh$
                               | $odd\ mx\ \ = Syn$
                      $part$   | $dir\_b \equiv dir = mx$
                               | $dir\_b \not\equiv dir = mx + 1$
                   in $wrap\_up\ a\ part$   -- assign
    where $wrap\_up\ a\ part$ = do   -- actual assignment
          $assign\_partition\ a\ part$
          $preds$  $\leftarrow predecessors\ ids\ a$
            -- some preds might now be assignable
          $forM\_\ preds\ (assign\ ids)$
```

Figure 22. Function *partition* assigns every attribute to a partition for constructing interfaces.

7.3 Extending implementations of Kastens' algorithm

In this section we have given an outline of a Haskell implementation of the AOAG algorithm together with the definition of some of its most important functions. Existing implementations of Kastens' algorithm can easily be extended and turned into an algorithm for scheduling LOAGs with a small number of changes.

Kastens' algorithm can detect three types of cycles depending on the graph in which it is encountered: D_P, ID_P or ED_P. The first change is to obtain the non-induced intra-visit dependencies (candidates) from the interfaces. The second change is to ensure that the main function of the algorithm returns failure when it discovers a cycle in D_P or ID_P. The third change is to add the set of augmenting dependencies (ADS) as an additional parameter to the main function. The augmenting dependencies are then added to D_P (as shown in Section 6.1).

The main function can now call itself recursively and use ADS as an accumulating parameter. If the initial call (with an empty ADS) encounters a cycle in D_P or ID_P then the input AG is not LOAG. If it encounters a cycle in ED_P then the input AG is not

```
-- try candidates one-by-one until the first result
explore   :: [Vertex]   → Graph s → Graph s
             → Graph s → Graph s → [Edge]
             → AOAG s (Maybe (Graph s))
explore c dp idp ids itfs =
    -- candidates are swapped ivds and part of the cycle
  explore'  ∘ filter (∈ [(f, t) | f ← c, t ← c])
            ∘ concatMap (depOccs ∘ swap)
where
  explore' :: [Edge] → AOAG s (Maybe (Graph s))
  explore' [] = return Nothing   -- no more candidates
  explore' (a : as) = do
    idpC  ← copy idp
    idsC  ← copy ids
    ifC   ← copy itfs
    let backtrack = explore c dp idpC idsC ifC as
    mEs ← insertCandidate a idp   -- see Section 7.2
    case mEs of    -- is ID_P cyclic?
      Nothing  → backtrack   -- yes
      Just es  → do          -- no
        reschedule ids es   -- rectify interfaces
        (ivds, ni_ivds) ← intras itfs
        mc              ← oagtest dp ivds
        case mc of    -- is there another cycle in ED_P?
          Nothing → return (Just itfs)   -- no
          Just c  → do    -- yes, select candidates
            mres ← explore c dp idp ids itfs ni_ivds
            case mres of    -- did the candidates help?
              Nothing → backtrack             -- no
              Just res → return (Just res)    -- yes
```

Figure 23. Function *explore* executes the backtracking algorithm for automatically selecting augmenting dependencies.

OAG but we can try to find augmenting dependencies that prove it is AOAG/LOAG. Whenever a cycle is encountered in a call to the main function, the main function should behave as follows:

- If the cycle is in D_P or ID_P it returns failure.
- If the cycle is in ED_P we use the candidates obtained from the interfaces. We extend ADS with one of the candidates to get ADS' and use it as an argument in a recursive call. If this call returns failure we add another candidate to ADS instead (backtracking step). If failure is returned while there are no more candidates to try, we return failure as well. If the call returns a result (for example graph ED_P), we return this result. In this case ADS' proves that the input AG is an AOAG/LOAG and visit-sequences can be constructed from the (non-cyclic) ED_P constructed by the successful call.

Calling the main function recursively is very inefficient as the graphs D_P and ID_P are recalculated at every call with only a relatively small number of extra edges. That is why we have chosen to propagate the changes in graph ID_P directly using *insertCandidate* and adjust the interfaces accordingly using *reschedule* as shown in the definition of *explore* in Figure 23.

```
reschedule :: Graph s → [Edge] → AOAG s ()
reschedule ids es = forM_ es fix_edge
where fix_edge :: Edge → AOAG s ()
      fix_edge (f, t) = do
        Just oldf ← partition_ix f
        Just oldt ← partition_ix t
        dirf      ← direction f
        dirt      ← direction t
        let newf | oldf < oldt = oldt + dist
                 | otherwise   = oldf
            dist | dirf ≢ dirt = 1
                 | otherwise   = 0
        unless (oldf ≡ newf) $ do
            -- replace the previous assignment
          reassign_partition f newf
          preds ← predecessors f
            -- some predecessors might need to be fixed
          forM_ (map (flip (,) f) preds) fix_edge
```

Figure 24. Function *reschedule* makes sure that all edges are pointing in the right direction, given a list of edges that are the result of adding a certain fake dependency.

8. Discussion

8.1 Theoretical contributions

This paper presents an algorithm of exponential complexity for finding a static evaluation order of any LOAG, where others have given algorithms of polynomial complexity for recognising a subset of the LOAGs [8, 12, 13]. We argue that backtracking is rare for practical AGs and that our algorithm is therefore efficient in practice.

This paper explains why Kastens' algorithm is incomplete: it does not solve the combinatorial problem of finding the right sets of visits for every interface simultaneously.

AGs that are not in LOAG form a class that can be divided into two based on whether these AGs satisfy the LOAG preconditions. The preconditions can be tested in polynomial time, thus AGs that do not satisfy the preconditions are efficiently recognised. AGs that are not in LOAG but do satisfy the LOAG preconditions can be constructed. This is shown by Natori et al. [12]. These AGs will force our algorithm to perform an exhaustive search of all candidates. We have not encountered any real-world examples of such AGs to test our algorithm with. It is interesting to see if we can generate large examples, either from scratch or by altering existing AGs, to examine the runtime of our algorithm further.

8.2 Practical contributions

The build process of the Utrecht Haskell Compiler compiles many AGs, some of which, including the main AG, require augmenting dependencies to be added in the source code to find a static evaluation order. Adding the right augmenting dependencies is tedious work, demanding insight in the scheduling process and the occasional trial-and-error. We have tested the AOAG algorithm on the AGs encountered in the UHC and compiled them successfully, in runtimes comparable with the original implementation of Kastens' algorithm. The AOAG algorithm compiles the main AG in 13 seconds, automatically selecting 10 augmenting dependencies, while our old implementation of Kastens' algorithm takes 17 seconds with 24 augmenting dependencies that were manually added to the source code while the UHC was developed. Using the same 24 augmenting dependencies the AOAG algorithm takes 6 seconds to compile, showing that roughly 7 seconds are required to evaluate the effects of the automatically selected augmenting dependencies.

The following numbers give an indication of the size of UHC's main AG. It consist of:

- 30 non-terminals.
- 134 productions.
- 1332 attributes (44.4 per non-terminal!).
- 9766 direct dependencies.

Even though the main AG contains many complicated dependency patterns, none of these patterns form a counter-example to our augmenting dependency selection mechanism and no backtracking is required.

We have tested our algorithm on all AGs contained in the UHC, AGs of other projects within Utrecht University and simple examples encountered in literature. We would have liked to test with more real-world AGs. Unfortunately there is no default syntax for writing AGs and we currently cannot easily test examples that have not been written in syntax that the UUAGC accepts.

The AOAG algorithm can be used as a complete compiler for LOAGs and as a tool for finding augmenting dependencies. These augmenting dependencies can be added to the source code manually. It has been implemented as a replacement for Kastens' algorithm in the latest version of the UUAGC and can be found on Hackage[8].

8.3 Future and relevant work

Other alternatives to Kastens' algorithm are found in literature. For example, Natori et al. introduce class OAG* [12] with an alternative way of constructing the interfaces. The Eli system contains a variant of Kastens' algorithm for recognising a superclass of OAG [9]. Pennings describes an algorithm for chained scheduling, relying on alternative induced dependency graphs [13]. These approaches lead to polynomial runtime algorithms for detecting membership of a subclass of LOAG (and a superclass of OAG). We have not implemented these approaches to compare them with the AOAG algorithm, although it would be interesting to see whether these algorithms are sufficient to compile the UHC without augmenting dependencies.

Adding an augmenting dependency might lead to a cycle that can only be solved by yet another augmenting dependency, influencing the runtime of the algorithm. AGs can be constructed that require an arbitrary number of augmenting dependencies, while a different choice in the first step works immediately. We have not investigated whether taking this into account can improve the runtime of the algorithm for large AG descriptions.

A common approach for solving NP hard problems is the use of a so-called SAT solver. The SAT problem is the standard NP complete problem [5] and even though no polynomial time algorithms for solving the problem are known, there exist many heuristic solvers that work well in many practical cases [4]. In future work we encode the LOAG scheduling problem as a SAT problem and use an existing SAT solver to solve the scheduling problem. Such approach may not only lead to a solution that works better in practice, but also allows for extra constraints to be put on the resulting schedule. Such extra constraints may be used to optimise the schedule even further.

Pennings' chained scheduling algorithm finds an order that is better suited for evaluators generated in purely functional programming languages. In an AG compiler generating both imperative and purely functional code, it is beneficial to have multiple options available. We have been able to express constraints in our SAT formulation for minimising the number of visits in the interfaces. It would be interesting to find other optimisations that we can express with constraints, and to investigate what the effects of these optimisations are on the time and space efficiency of the generated evaluators.

[8] http://hackage.haskell.org/package/uuagc

8.4 Conclusion

Finding a static evaluation order for AGs is hard. We have shown that we can find an order for all LOAGs, by automatically selecting augmenting dependencies from a set of candidates with a backtracking algorithm. Although the algorithm we presented is exponential in theory, experiments with the Utrecht Haskell Compiler show that it is efficient for practical full-sized AG descriptions and does not require backtracking.

References

[1] H. Alblas. Attribute evaluation methods. In *Attribute Grammars, Applications and Systems*, volume 545 of *Lecture Notes in Computer Science*, pages 48–113. Springer Berlin / Heidelberg, 1991. ISBN 978-3-540-54572-9.

[2] G. V. Bochmann. Semantic evaluation from left to right. *Communications of the ACM*, 19(2):55–62, Feb. 1976.

[3] J. Bransen, A. Middelkoop, A. Dijkstra, and S. D. Swierstra. The Kennedy-Warren algorithm revisited: ordering Attribute Grammars. In C. Russo and N.-F. Zhou, editors, *Practical Aspects of Declarative Languages*, volume 7149 of *Lecture Notes in Computer Science*, pages 183–197. Springer Berlin / Heidelberg, 2012.

[4] K. Claessen, N. Een, M. Sheeran, N. Sörensson, A. Voronov, and K. Åkesson. Sat-solving in practice. *Discrete Event Dynamic Systems*, 19(4):495–524, Dec. 2009. ISSN 0924-6703.

[5] S. A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, pages 151–158, New York, NY, USA, 1971. ACM.

[6] A. Dijkstra. *Stepping Through Haskell*. PhD thesis, Utrecht University, 2005.

[7] J. Engelfriet and G. Filè. Simple multi-visit attribute grammars. *Journal of computer and system sciences*, 24(3):283–314, 1982.

[8] U. Kastens. Ordered attributed grammars. *Acta Informatica*, 13:229–256, 1980.

[9] U. Kastens, P. Pfahler, and M. T. Jung. The eli system. In *Proceedings of the 7th International Conference on Compiler Construction*, CC '98, pages 294–297, London, UK, UK, 1998. Springer Berlin / Heidelberg.

[10] K. Kennedy and S. K. Warren. Automatic generation of efficient evaluators for attribute grammars. In *Proceedings of the 3rd ACM SIGACT-SIGPLAN symposium on Principles on programming languages*, POPL '76, pages 32–49, New York, NY, USA, 1976. ACM.

[11] D. E. Knuth. Semantics of context-free languages. *Mathematical systems theory*, 2(2):127–145, June 1968.

[12] S. Natori, K. Gondow, T. Imaizumi, T. Hagiwara, and T. Katayama. On eliminating type 3 circularities of ordered attribute grammars. In D. Parigot and M. Mernik, editors, *Second Workshop on Attribute Grammars and their Applications, WAGA'99*, pages 93–112, Amsterdam, The Netherlands, March 1999. INRIA rocquencourt.

[13] M. C. Pennings. *Generating Incremental Attribute Evaluators*. Ph.D. thesis, Computer Science, Utrecht University, November 1994.

[14] T. W. Reps and T. Teitelbaum. *The Synthesizer Generator Reference Manual*. Texts and monographs in computer science. Springer Berlin / Heidelberg, 3rd edition, 1989. ISBN 978-3-540-96910-5.

[15] J. Saraiva. *Purely Functional Implementation of Attribute Grammars*. PhD thesis, Utrecht University, 1999.

[16] S. D. Swierstra, P. R. A. Alcocer, and J. Saraiva. Designing and implementing combinator languages. In *Advanced Functional Programming*, volume 1608 of *Lecture Notes in Computer Science*, pages 150–206. Springer Berlin / Heidelberg, 1999.