Utrecht University

Master of Science Thesis

June 26, 2014

Fine-tuned Scheduling of Linear Ordered Attribute Grammars

Author:
L.T. van Binsbergen
ICA-3132315

Daily Supervisor:
J. Bransen
Secondary Supervisors:
Dr. A. Dijkstra
Prof. Dr. J. T. Jeuring

Abstract

Attribute Grammars (AGs) are a formalism for defining tree-based computations. Trees are extended with attributes representing results and parameters of such computations. A programmer using AGs does not have to worry about the order in which the attributes are *evaluated*, this task is delegated to an AG compiler. AGs are typically used for writing compilers, as most compilers perform a number of static analyses that should be efficiently interleaved. By relying on an AG compiler to perform this task, a programmer can focus on the crucial aspects of the analyses.

Linear Ordered Attribute Grammars (LOAGs) form the largest class of AGs for which a *static evaluation order* is determined. For LOAGs strict, small-sized evaluators are generated that are assumed to be the most efficient evaluators of all AG classes.

Finding a static evaluation order for an AG, thus determining whether it is an LOAG, is an NP-complete problem. In order to find a static evaluation order for LOAGs, Kastens' algorithm (1980) is commonly used. However, this polynomial runtime algorithm only works for a subset of the LOAGs, the Ordered Attribute Grammars (OAGs). The algorithm optimistically schedules the evaluation of incompatible attributes together.

LOAGs are transformed to OAGs using *fake dependencies*: manual additions to the source code with the sole purpose of forcing the AG compiler to separate the evaluation of certain attributes. Finding the right combination of fake dependencies is tedious work, often forcing programmers to resort to trial and error. Especially for large compilers this approach is undesirable.

In order to find a static evaluation order for the Utrecht Haskell Compiler (UHC), 24 fake dependencies are required. The tedious work of finding these fake dependencies is the main motivation for this thesis. We show that, even though the problem is NP-complete, static evaluation orders can be found for all LOAGs in acceptable runtimes, presenting a Haskell implementation of two new algorithms.

The first algorithm is inspired by the use of fake dependencies. The algorithm calculates the constructions produced by Kastens' algorithm and uses them to find a set of candidates: dependencies that, when added to the source code, *might* lead to a static evaluation order. Since their correctness is not certain, their selection is implemented as a backtracking procedure. The algorithm requires no backtracking to find an evaluation order for the UHC, gathering a set of 10 fake dependencies. We argue that backtracking will be rare for practical AGs.

After a thorough examination of multiple characterisations of LOAGs, a minimal decision problem is formulated. This formulation translates directly into a Boolean formula, for which a satisfying assignment represents an evaluation order. A SAT-solver is used to find a satisfying assignment. The result is an algorithm finding an evaluation order for the UHC in 10 seconds *without* fake dependencies, while Kastens' algorithm requires 25 seconds *with* a set of fake dependencies.

With a static evaluation order, an AG compiler can optimise this order with respect to user-defined criteria. Kastens' algorithm finds an evaluation order that is efficient for evaluators generated in imperative languages, while Pennings (1994) introduced *chained scheduling* to find orders more suitable for evaluators generated in purely functional languages. We argue that the SAT-based algorithm presented in this thesis can be extended to produce schedules efficient for either programming paradigm, by providing user-defined optimisations.

Contents

Chapter 1

Introduction

1.1 Why Attribute Grammars Matter

An Attribute Grammar (AG) is a formalism for defining tree-based computations.

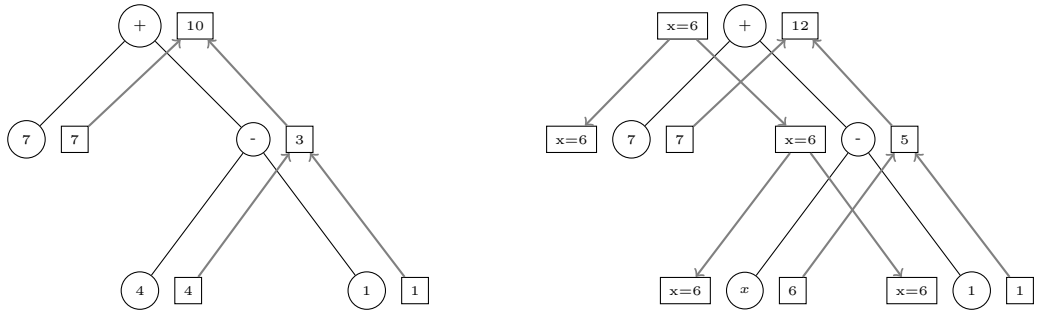

Figure 1.1: Trees depicting arithmetic expressions (7+(4-1)) and (7+(x-1)). The evaluation result is displayed to the right and the environment to the left of subexpressions.

Consider the tree representation of two arithmetic expressions in Figure 1.1 and the code fragment for evaluating arithmetic expressions in Figure 1.2. To evaluate expressions, a programmer typically writes a set of functions to evaluate all subtrees. One function is required for every datatype and every function consists of multiple cases, one for every constructor of the datatype. A call from one such function to another is called a visit to a node of the tree. The order in which visits are used executed determines the treewalk of the computation.

Defining tree-based computations this way complicates adding (even simple) extensions to the computation. Every change forces the programmer to reconsider the visits. Figure 1.2 underlines all the locations a change is needed for adding an environment as parameter. It is easy to forget a change or to make a mistake, as most changes are trivial while others are vital for the result. The visits changed as the environment is required as a parameter.

Extra problematic is producing multiple results with the same computation, reusing the original treewalk. To maintain the same number of visits, results are *tupled*[HITT97, Swi05]. For example, calculating a pretty-printed interpretation of the expression, in which
Plus (*Term* (*Nat* 2)) (*Term* (*Nat* 3)) is displayed as "(2+3)", is shown in Figure 1.3.

By writing large computations without tupling, a programmer maintains elegance at the price of efficiency. Without tupling, a separate visit is required for every result, producing a bigger

```
data Expr = Plus      Expr Expr
          | Min       Expr Expr
          | Term      Term
data Term = Nat       Int
          | Minus     Term
          | Var       String
type Env  = [(String, Int)]

evalExpr :: Expr → Env → Int
evalExpr (Plus e1 e2)  env = (evalExpr e1 env) + (evalExpr e2 env)
evalExpr (Min e1 e2)   env = (evalExpr e1 env) − (evalExpr e2 env)
evalExpr (Term term)   env = evalTerm term env

evalTerm :: Term → Env → Int
evalTerm (Nat n) env   = n
evalTerm (Minus t) env = −(evalTerm t env)
evalTerm (Var id) env  =
  case lookup id env of
    Nothing → error ("Variable " ++ id ++ " used, but not declared")
    Just v  → v
```

Figure 1.2: Example of evaluating simple arithmetic expressions with variables in Haskell.

```
evalExpr :: Expr → Env → (String, Int)
evalExpr (Plus e1 e2) env = let (pp1, v1) = evalExpr e1 env
                                (pp2, v2) = evalExpr e2 env
                            in ("(" ++ pp1 ++ "+" ++ pp2 ++ ")", v1 + v2)
```

Figure 1.3: Example of tupling the results of multiple computations.

treewalk. In an AG description, the definitions of multiple results are given independent of each other. The task of ensuring that results are computed efficiently is delegated to the AG compiler. AGs combine the best of both approaches.

AGs and compiler writing

Large interleaved computations are often required in the implementation of compilers. A compiler typically incorporates multiple static analyses to diagnose potential problems in source code, helping programmers build better software. To make compiler writing easier, AGs enable the separate definition of static analyses. AGs are said to assign meaning to sentences of a context-free language[Knu68]. As such, AGs are useful to define the semantics of programming languages.

Analysing treewalks

As noted before, the visits defined by the evaluation functions determine a treewalk. In the example, the treewalk is top-down: at no point is the result computed for one subexpression passed down as an argument to another subexpression. Whenever multiple calls have to be

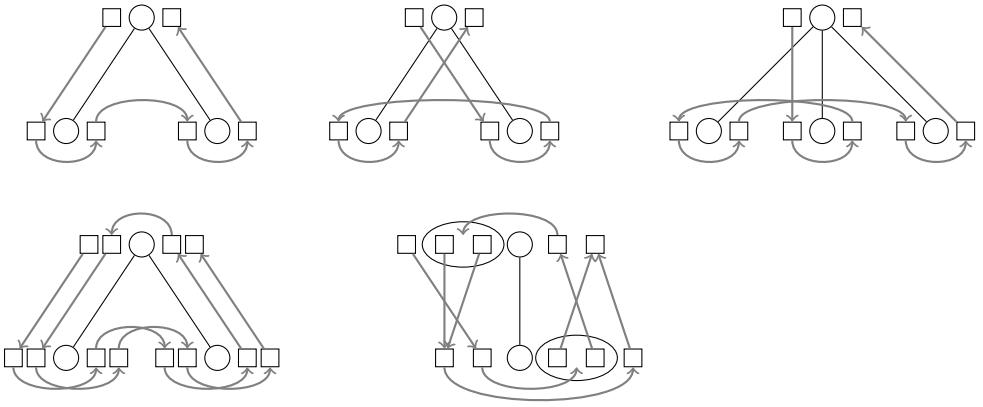

Figure 1.4: Treewalks with single and multiple visits per nodes. A square is an attribute and a circle a tree node. An arrow depicts the flow of information from one attribute to another. In order to get the information of a child node a visit has to be made. The last treewalk shows a visit that uses multiple attributes and a visit that produces multiple attributes. The attributes inside an oval are tupled.

made (in case of *Plus* or *Minus*) the order in which the calls are executed is arbitrary (and could be executed in parallel).

Adding more results and parameters to the computation requires rethinking the visits and the resulting treewalk. Finding a small treewalk for interleaved computations is not easy. Moreover, a terminating treewalk is not possible when the evaluation of some results are mutually dependent. Figure 1.4 shows some different types of treewalks.

AGs are used to describe tree-based computations at a higher level of abstraction, abstracting away from the problem of finding an efficient interleaving of multiple results. The attributes are the 'boxes' added to the tree, representing either the result of a computation, as a *synthesized* attribute, or a parameter of a computation, as an *inherited* attribute. To program with AGs is to: define the tree of interest, attach a set of attributes to every node of the tree and define how every attribute is computed from its surrounding attributes. An AG compiler is able to inform the programmer of any missing attribute definitions.

AGs are typically implemented as a Domain Specific Language (DSL). The code that executes the computation is generated from a high level AG description by an AG compiler. The description contains attribute definitions written in the syntax of a certain host language. AGs can also be implemented as embedded DSLs (EDSLs) in a host language directly[VSS09].

An AG compiler can use multiple approaches to determine a treewalk for the interleaved computations described in the grammar. This thesis focus on the following approach: "Find for every node in the tree a set of visits, specifying for every visit which inherited attributes it receives and synthesized attributes it produces, such that it is possible to execute each visit strictly[1]". The set of visits found for every node provides an *interface* between the context and the children of that node.

Note that with Haskell's lazy evaluation, scheduling interleaved computations is not a problem. If a treewalk is possible, it will be found at runtime, otherwise a 'loop' is reported. An AG compiler producing code for a lazy language can delegate this responsibility to the compiler of the host language. However, in many systems, e.g. large compilers, strict evaluation is desired.

[1]If a visit is executed strictly, every attribute it requires needs to be completely evaluated before execution and every attribute it produces is completely evaluated before the next visit is executed.

1.1.1 Programming with Attribute Grammars

Attribute Grammars relieve programmers of the following tasks:

- Proving the computation contains no mutually dependent results whenever strict evaluation is desired.
- Determining which treewalk is used to compute all results whenever strict evaluation is desired.
- Finding all locations where a change is required when a computation is extended.
- Generating boilerplate code for distributing and combining values.

With AGs it is possible to separate computations into separate code fragments, improving maintainability and reusability of the code. Figure 1.5 shows an AG description for evaluating arithmetic expressions of the previous example. Note how adding variables and an environment is written as a separate fragment. The syntax for writing AGs is explained in Chapter 2.

High level programming

The Attribute Grammar formalism enables an alternative way of programming. It allows programmers to think only in terms of tree fragments and these fragments directly associate with a visual representation thus allowing a more intuitive and spatial understanding of computations over a tree.

AGs can reside in any programming environment because AG *evaluators* suitable for use by a specific programming environment can be generated from an AG description.

Evaluators are functions that take a tree (conforming the syntactic rules described in the grammar) as input, together with the values for inherited attributes of the root node, and return the values of the synthesized attributes of the root node. Evaluators are generated for a *host language*, targeted to fit in the chosen programming environment. The definitions of attributes are usually written in the same language. AG compilers should then be able to distinguish, in the right-hand sides of attribute definitions, references to other attributes from the native syntax. In the example of Figure 1.5 the host language is Haskell and '@' is used to distinguish attribute references[2].

An example of an AG compiler is the Utrecht University Attribute Grammar Compiler (UUAGC) [3]. It can be used to generate code for the functional programming languages Haskell, Clean and OCaml. The syntax in this thesis is the syntax accepted by the UUAGC.

Imperative settings

Programming with AGs is a form of declarative programming as the order in which attribute definitions are given is irrelevant to the computation an AG describes. If strict evaluation of an AG is possible, an AG compiler is able to generate code for non-declarative languages. As such, AGs enable declarative programming in imperative settings.

[2] Attribute definitions can also be defined in a specific AG expression language which is then translated into code compatible with the generated evaluator.

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

```
  -- part 1, simple arithmetic expressions
data Expr | Plus      e1 : Expr e2 : Expr
          | Min       e1 : Expr e2 : Expr
          | Term      t  : Term
data Term | Nat       n : Int
          | Minus     t  : Term
attr Expr Term    -- syntax for declaring attributes
  syn val : Int
  -- defining the semantic rules for every production rule
sem Expr  | Plus      lhs.val = @e1.val + @e2.val
          | Min       lhs.val = @e1.val - @e2.val
          | Term      lhs.val = @t.val   -- can be automatically generated
sem Term  | Nat       lhs.val = @n
          | Minus     lhs.val = -@t.val
  -- part 2, adding variables to expressions
data Term | Var       id : String   -- extra constructor for non-terminal Term
type Env = [(String, Int)]
attr Expr Term
  inh env : Env   -- extra (inherited) attribute containing the environment
sem Term  | Var
  lhs.val = case lookup @id env of    -- defining attribute val for the new constructor
              Nothing -> error ("Variable " ++ @id ++ " used, but not declared")
              Just v  -> v
sem Expr  | Plus      e1.env = @lhs.env   -- can be automatically generated
                      e2.env = @lhs.env   -- can be automatically generated
          | Min       e1.env = @lhs.env   -- can be automatically generated
                      e2.env = @lhs.env   -- can be automatically generated
          | Term      t.env  = @lhs.env   -- can be automatically generated
sem Term  | Minus     t.env  = @lhs.env   -- can be automatically generated
```

Figure 1.5: AG description for evaluating arithmetic expressions. It shows the incremental construction of datatypes and computations. The syntax is explained in Section 2.

1.2 Reasons not to use Attribute Grammars

Restrictions

Attribute Grammars relieve a programmer of the work of specifying the evaluation of computations, at the same time restricting the ways in which such computations can be expressed.

Firstly, certain low level optimisations (potentially leading to significant performance benefits) are not always possible, depending on the combination of host language and compilers used.

Secondly, the programmer might know an evaluation order that is preferred over the order the AG compiler finds. After all, an evaluation order can be optimal in multiple regards: runtime, memory usage, ordering of specific attributes, compatibility with host language/compiler, etc.

In general, a programmer is expected to have more information about the problem he is working with than a compiler and he can use this knowledge to optimise his code to some degree.

Programming environments

Attribute Grammars often rely on an implementation in one (or more) host languages. In a typical programming environment multiple languages are used simultaneously, each with their own set of tasks, and an AG system does not always integrate properly in a multi-language environment. To fully integrate, an AG compiler might be required to generate code for a large number of the languages in the environment, depending on how it is used.

There is no globally recognised default AG compiler, able to generate evaluators for a large set of languages, with its own IDE (Integrated Development Environment), a large set of different scheduling algorithms and ways for programmers to specify how optimise the generated code.

1.3 Motivation

This thesis investigates all aspects of scheduling Linear Ordered Attribute Grammars (LOAGs), a class of AGs that allow the generation of simple, efficient and strict evaluators. LOAGs are particularly interesting because they allow to find a linear evaluation order at compile time.

The main motivation for this thesis is the problem of finding a static evaluation order for scheduling the Utrecht Haskell Compiler (UHC)[Dij05].

1.3.1 Fake dependencies in the Utrecht Haskell Compiler

The UHC consists of a large number of AG descriptions. The "main AG" describes many static analyses. To optimise the generated code, the AG has been adjusted to comply with all restrictions imposed on LOAGs. The assumption is that, by writing and compiling the main AG as an LOAG, the most efficient evaluator is generated for it (compared to other AG classes). The main AG is very large indeed. It consists of:

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

Writing the main AG as a LOAG is not enough to schedule it as an LOAG. Although LOAGs are very popular in AG literature, no algorithm for scheduling all members of the class has been described. The absence of an algorithm for LOAGs is most likely explained by the fact that recognising LOAGs is an NP-complete problem and no polynomial runtime algorithm can exist, unless $P = NP$.

Instead, the main AG is compiled using Kastens' algorithm[Kas80], designed for the class OAG, a strict subclass of LOAG. LOAGs that are not OAG (not recognised by Kastens' algorithm) can still be compiled by Kastens' algorithm using *fake dependencies*. This 'trick' is popular in literature and will be discussed thoroughly in Chapter 4.

The UHC's main AG is not an OAG. By a somewhat tedious process of trial and error, a set of fake dependencies has been found that enables Kastens' algorithm to compile it. This solution is fragile, however, as additional attributes might interfere with the current set of fake dependencies, causing unnecessary cycles. Some of these cycles are solved by removing fake dependencies, others by adding more. At the start of november 2013, 24 fake dependencies were present in the source code of the main AG, added for the sole purpose of being able to schedule the AG using Kastens' algorithm. That special syntax has been added to the UUAGC for adding fake dependencies in an AG description more easily, is an indication of the importance of fake dependencies. Although the approach is successful, we set finding an algorithm to compile all LOAGs, without the need for fake dependencies, as the goal of this thesis.

By the same motivation a purely functional variant of the Kennedy-Warren algorithm[KW76] has been developed and implemented in the UUAGC[BMDS12]. The algorithm is designed to compile AG descriptions of the class Absolutely Non-Circular Attribute Grammars (ANCAGs, described in section 3.2). Since ANCAG is a strict superclass of LOAG, it can not assume all of the restrictions imposed on LOAGs. As a result, it does not find a static evaluation order. Although the generated evaluators for this class are guaranteed to be strict, the precise evaluation order is not know at compile time. If the evaluation order is known statically the compiler is able to perform certain optimisations. This thesis addresses this topic briefly.

1.4 Thesis Overview

In Chapter 2 Attribute Grammars are defined, first by example, then formally. The chapter provides an interesting and very simple example of an LOAG that can not be compiled by Kastens' algorithm. Chapter 2 also introduces the visual style used to depict AGs.

Chapter 3 investigates the classes ANCAG, LOAG and OAG, focusing on the difference between LOAG and OAG. Multiple definitions of LOAG are discussed. Each provides a different viewing angle for looking at the problem of scheduling LOAGs. These definition inspire the algorithms presented in subsequent chapters.

Chapter 4 builds on the notion of *fake dependencies* to recognise LOAGs. The effects of fake dependencies on the scheduling procedure of Kastens' algorithm are analysed. The result of the analysis is a way to determine which fake dependencies are *candidates* for scheduling an AG. Kastens' algorithm is extended to select from these candidates automatically, using a backtracking strategy. Since the problem of scheduling LOAGs is NP-complete, selecting the right candidate at all times must be hard. However, it turns out that no backtracking is required for the main AG and all other AGs tested. A Haskell implementation of the algorithm is given.

Chapter 5, explores possibilities for defining an algorithm for LOAGs that does not use the notion of fake dependencies. The purpose is to find an algorithm that solves the problem more *directly*, dealing with the specific difficulties of the problem, and more *generally*, such that it is able to produce non-arbitrary evaluation orders. The result is a decision problem described as a

satisfiability problem. To do this, a general approach for ruling out cycles in graphs using SAT-solvers is given. For every input AG, a Boolean formula is produced. An AG is an LOAG if and only if the formula constructed for the AG is satisfiable. Using the domain specific knowledge gained from preceding chapters, the Boolean formula is minimised, enabling very large AGs to be recognised by a SAT solver quickly. The result is an algorithm capable of scheduling large LOAGs quickly. Chapter 5 provides the proof of correctness of the algorithm, as well as a Haskell implementation.

Chapter 6 discusses the minimal structures required to generate evaluators for LOAGs, completing the pipeline from AG description to evaluator. Since an algorithm for LOAGs produces an static evaluation order at *compile time*, a compiler implementing an LOAG algorithm can optimise the evaluators it produces directly. Therefore we define a number of possibly desired optimisations. Implementing these optimisations is left as future work.

Related work is discussed briefly in Chapter 7. All contributions, results, remaining future work and conclusions are given in Chapter 8.

Chapter 2

Attribute Grammar Notation

2.1 Label-Value Attribute Grammar

This section introduces AGs, AG notation and dependency graphs using an interesting example of an AG that will be referred to throughout the thesis. It demonstrates two patterns common in compiler writing, that are not trivially scheduled together. The example, referred to as LABEL-VALUE, is taken from Bransen et al.[BMDS12].

2.1.1 Tree grammar

A binary tree carrying integers in its leafs can be represented by the following context-free grammar:

data `Tree` | ***Leaf*** *val* : *Int*
| ***Bin*** *l* : *Tree* *r* : *Tree*

Figure 2.1: Label-Value grammar

A non-terminal `Tree` is defined, with two productions ***Leaf*** and ***Bin***. Production ***Leaf*** states that `Tree` can be derived from a terminal of type `Int`. Production ***Bin*** states that `Tree` can also be derived from two subtrees. We say that ***Leaf*** is a production with a node of type `Tree` as a parent and a terminal of type `Int` as a child, while ***Bin*** is a production with a node of type `Tree` as a parent and two child nodes of type `Tree`. All nodes in a production rules are called *fields*.

Identifiers are associated with all fields of each production rule, in order to refer to fields in attribute definitions. A parent node is identified by *lhs* (left-hand side). When a field, identified by x, is a terminal or non-terminal T, we say that x is of *type* T, denoted by $x : T$. Additionally we say that the production rule identified by ***Leaf*** is a constructor function taking one argument of type *Int* and that ***Bin*** is a constructor function taking two arguments, both of type `Tree` and that both constructor functions return a value of type `Tree`.

Terminal symbols are recognized by the fact that they do not have production rules associated with them (terminals are underlined in the AG notation). The terminal of type `Int` of production ***Leaf*** is identified by *val*. The left and right subtree of production ***Bin*** are identified by l and r, respectively.

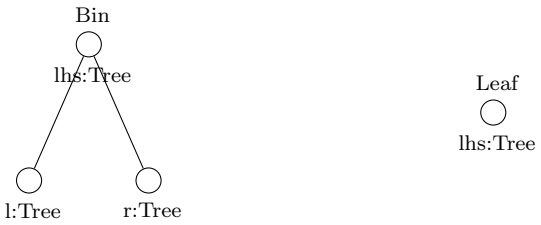

Figure 2.2: The production graphs of productions *Bin* and *Leaf*.

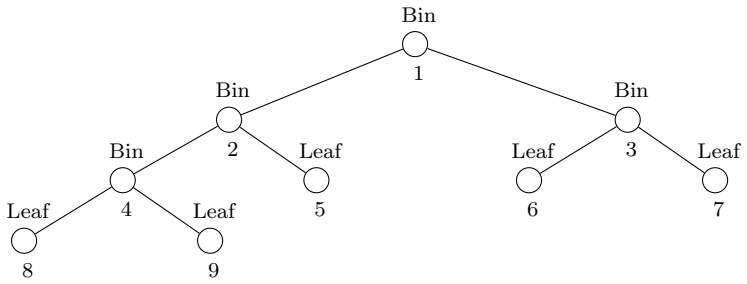

Figure 2.3: A possible derivation tree of the LABEL-VALUE AG description in Figure (2.1).

Context-free grammars describe formal languages, with terminals and non-terminals as their alphabet. From the alphabet valid sentences of the grammar are derived from grammar rules. Productions describe these rules and are also called *derivation rules*. Recognising a piece of source code as a valid sentence of the grammar is called *parsing*. The result of a successful parse is a *derivation tree* or *parse tree*. The productions of LABEL-VALUE are presented visually in Figure 2.2. A possible derivation tree is shown in Figure 2.3.

2.1.2 Attribute *vals*

Declaration For any valid derivation tree of LABEL-VALUE, we wish to list all its values in a right-to-left fashion. To achieve this, we assign an attribute, identified by *vals*, of type $[\underline{Int}]$, to non-terminal `Tree`. The attribute is accumulative: every leaf of the tree adds its value to the list. As such, the attribute needs to be *chained*: it is both inherited and synthesized. Figure 2.4 shows the attribute declaration for non-terminal `Tree`.

attr `Tree` **inh** *vals* : $[\underline{Int}]$
syn *vals* : $[\underline{Int}]$

Figure 2.4: Label-Value grammar with attribute *vals*

An attribute is declared, using the keyword **attr**, for a (list of) non-terminal(s). Each attribute is declared as either **inh** or **syn** together with its name and type. The name of an attribute is referred to as an *attribute identifier*. An attribute is uniquely determined by the combination: non-terminal × attribute identifier × direction (inherited or synthesized).

Semantic domain In order to *complete* this computation (see Definition 2.9 for the definition of a complete AG) *semantic function definitions* must be given. Together they form the *semantic domain*. Semantic function definitions define how every attribute is calculated in every context (production). Whenever we speak of attributes at the level of a production, we speak of *attribute occurrences*. Attributes at the level of a derivation tree are referred to as *attribute instances*. The evaluation process performed by an AG evaluator is called *decoration* of a derivation tree. It are attribute instances that are evaluated during decoration.

Eight occurrences of *vals* exists in LABEL-VALUE. An inherited and synthesized occurrence at the parent of both productions and at the two children of production `Bin`. The synthesized occurrences of the parents and the inherited attributes of the children are the *output* occurrences and require a definition. Output occurrences 'leave' production rules, while the remaining *input occurrences* 'enter' productions. At production `Leaf` we insert the leafs value to the accumulative result it inherits. At `Bin` the result from the parents context is passed on to the right subtree, the result of the right subtree is passed on to the left subtree, the result of the left subtree contains all values of the subtrees in the right order and is *returned* (passed on upwards to the parents context). We are writing the function definitions in Haskell, using Haskell's list operator (:).

```
sem Tree
  | Leaf
    lhs.vals = @val : @lhs.vals
  | Bin
    r.vals   = @lhs.vals
    l.vals   = @r.vals
    lhs.vals = @l.vals
```

Figure 2.5: Label-Value grammar with attribute *vals*

The left-hand side of a semantic function definition determines for which attribute occurrence the definition is given by that equation. An attribute occurrence is uniquely identified by the combination: production × field × attribute identifier × direction. When an occurrence is at the left-hand side of a semantic function definition it is an output occurrence, while occurrences at the right-hand side are input occurrences[1]. As such, the direction of attribute occurrences can be determined. The right-hand side of the equations contain references to attribute occurrences and terminals prefixed with the @-symbol to distinguish them from Haskell syntax.

An attribute occurrence in the left-hand side *depends* on all the occurrences in the right-hand side. We depict such dependencies in a *dependency graph*. The dependency graph for computing *vals* of LABEL-VALUE is shown in Figure 2.6. The dependency graphs do not actually contain dependencies, instead they represent the opposite: the flow of information from attribute to attribute. There is an edge $(a \rightarrow b)$ if a is in the right-hand side of some semantic function definition of which b forms the left-hand side.

2.1.3 Attribute *label*

We extend the semantics of the binary trees by assigning an identifier or *label* to every leaf. This is achieved by adding another chained attribute, *label*, to the grammar. The attribute, of type `Int`, will be used as a seed value to assign labels of type `Int` to all leafs. Every time we use the seed it will be increased. In order for the labels to be unique we must never use the

[1]This is an assumption in the UUAGC that this thesis uses abundantly.

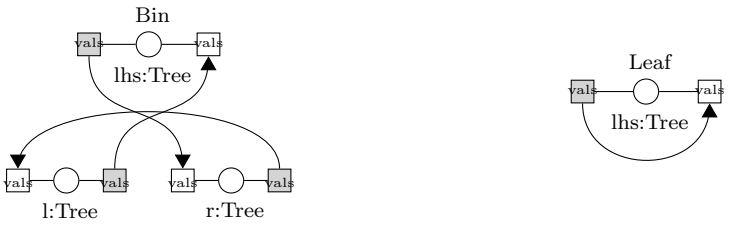

Figure 2.6: Dependency graph of the AG description shown in Figure 2.5.

same seed twice. To prevent this, the attribute is sent from node to node exactly like attribute *vals*. Except this time, we distribute the attribute in the other direction (left-to-right). Because it is sent in the other direction, the two computations (VALUE and LABEL) can not be tupled and separate visits have to be used. In Chapter 4 is shown that Kastens' algorithm attempts to schedule the two attribute in the same visit. Figure 2.7 shows the attribute declarations and semantic function definitions required for computing the labels.

```
attr Tree inh label : [Int]
          syn label : [Int]
sem Tree
  | Leaf
    loc.label = @lhs.label
    lhs.label = @lhs.label + 1
  | Bin
    l.label   = @lhs.label
    r.label   = @l.label
    lhs.label = @r.label
```

Figure 2.7: Label-Value grammar with attributes *vals* and *label*

The label of a leaf is assigned to a *local attribute*, identified by **loc**, allowing us to refer to the label in other semantic function for production `Leaf`. Local attributes are typically used to prevent code duplication. Assigning a value to a local attribute has the same effect as adding a new terminal to the production. Figure 2.8 shows the dependency graph of LABEL-VALUE.

2.2 Formal definitions

This section gives the formal definitions of the concepts introduced in the previous section together with other concepts required throughout this thesis. The given definitions are largely based on work by Saraiva [Sar99] and Kastens[Kas80]. This thesis focuses on the classical (first-order) Attribute Grammars first introduced by Knuth[Knu68]. We define Context-Free Grammars, Attribute Grammars as an extension to Context-Free Grammars, partial orders and linear orders. Chapter 3 introduces the classes of AGs relevant to this thesis.

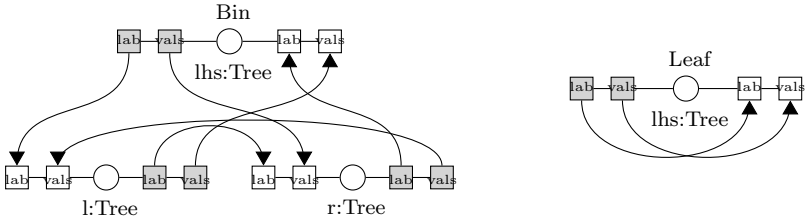

Figure 2.8: Dependency graph of the AG description in Figure 2.7.

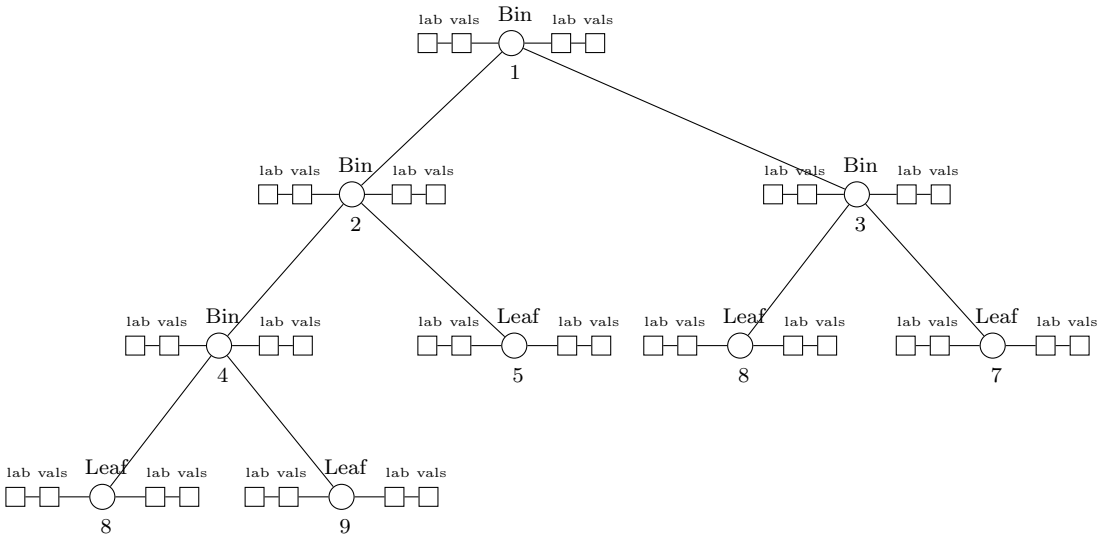

Figure 2.9: The derivation tree of Figure 2.3 with attribute instances attached to its nodes.

2.2.1 Context-Free Grammars

Definition 2.1 (Context-Free Grammar) A *Context-Free Grammar* (CFG) is a triple $G = \langle V, P, S \rangle$, where V is a finite non-empty set of symbols: the *vocabulary*. $V = \Sigma \cup N$ is partitioned into a set of terminal symbols Σ, the *alphabet*, and a non-empty set of non-terminal symbols N. P is a set of *productions*. Each production $p \in P$ is of the form $p : X_{p,0} \rightarrow X_{p,1}, X_{p,2}, \ldots X_{p,|p|}$, with $X_{p,0} \in N$. The i-th symbol of the production p, $X_{p,i} \in V$, is a *symbol occurrence*. $\mid p \mid$ is the number of symbols in production p. $X_{p,0} = lhs(p)$ is the *left-hand side* of p and each $X_{p,i}$ with $i > 0$ is an element of the *right hand side* of p, i.e. $X_{p,i} \in rhs(p)$ with $0 < i \leqslant \mid p \mid$. S is the starting symbol of the grammar. ■

Context-free grammars describe how we can generate a sentence of a language by deriving a composition of symbols. Terminal and non-terminal symbols, referred to as nodes, form the set of symbols which is the vocabulary of a language. *Production rules* show how we can derive from a single non-terminal, independent of its surrounding symbols or *context*, a sentence of the language, possibly containing other non-terminals that enable further derivation. These other non-terminals form the right-hand side of the production rule and we will refer to them as *children* or *child nodes*. Conversely the single non-terminal of the left-hand side is called the *parent node*.

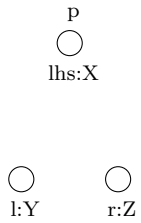

Figure 2.10: Graphic representation of the production rule $p : X \to Y \to Z$. The parent node is labelled *lhs* and the child nodes l and r.

Parent and child nodes of a production p are referred to as the *fields* of p. Every field is an *occurrence* of some non-terminal $X \in N$. Figure 2.10 shows a production p of non-terminal X with two children l and r, that are occurrences of non-terminal Y and Z, respectively.

There is a strong correspondence between non-terminals and datatypes in a functional programming language such as Haskell[PJea03]. A production rule p can be seen as a constructor for the datatype X, for which holds that $X = lhs(p)$, taking the symbols $S \in rhs(p)$ as arguments.

A relation $\Rightarrow$ between α and β claiming the existence of a derivation of β from α is valid if and only if, for all $\alpha = \alpha_1 A \alpha_2$, $\beta = \alpha_1 \gamma \alpha_2$ there exists a production $(p : A \to \gamma) \in P$, with $\alpha \in V^+$ and $\beta, \gamma, \alpha_1, \alpha_2 \in V^*$.
We use this relation in the definition of a context-free language:

Definition 2.2 (Context-Free Language) The *context-free language* generated from grammar $G = \langle V = \Sigma \cup N, P, S \rangle$ is the set $\mathcal{L}(G) = \{\ \mu \in \Sigma^* \mid S \Rightarrow^* \mu\ \}$. ■

The language generated by grammar $G = \langle V, P, S \rangle$ is the set of all sentences that can be derived, using zero or more steps validated by the productions rules in P, from the starting symbol S.

The steps that derived a sentence from the starting symbol form a *derivation tree* or *parse tree*. We say that a node k in a derivation tree is an instance of some non-terminal $X \in N$. We say that k was derived or constructed by a production p for which holds that $lhs(p) = X$. Note that there might be more than one production $p \in P$ with $X = lhs(p)$, although there is only one p used for deriving k.

2.2.2 Attribute Grammars

Attribute Grammars are an extension of CFGs. Every non-terminal has a set of *attributes* assigned to it. Attributes at production level are called *attribute occurrences*. Every attribute a assigned to non-terminal X has an occurrence o at field f if and only if f is an occurrence of X. Values of attributes are defined in terms of terminal symbols and attributes available in their *context*: the production rules in which they reside. The value of an attribute a is thus defined separately for all occurrences of a. Each attribute a assigned to a non-terminal X is either *inherited*, its occurrences are defined in productions in which the occurrences of X are child nodes, or *synthesized*, its occurrences are defined in productions of which the parent is an occurrence of X. Inherited attributes represent top-down computations, while synthesized attributes represent bottom-up computations.

Attributes at the level of derivation trees are called *attribute instances*. Every attribute a assigned to non-terminal X has an instance i at node K if and only if K is an instance of X. Figure 2.9 shows a possible derivation tree of Label-Value with its attribute instances.

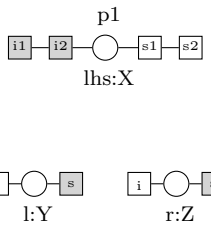

Figure 2.11: A production rule with attributes occurrences assigned to its nodes.
Inherited occurrences are placed at the left-hand side of a node, synthesized occurrences at the right-hand side.

Definition 2.3 (Attribute Grammar) An *Attribute Grammar* (AG) is a triple $\langle G, A, D \rangle$, where $G = \langle V = \Sigma \cup N, P, S \rangle$ is a context-free grammar. A is the set of all *attributes* $(X \cdot a)$, with $X \in N$ and a an *attribute identifier*. $X \cdot a$ denotes that the attribute identified by a is assigned to non-terminal X. $D = (T, \mathcal{T}, E)$ is the *semantic domain* of the AG, where T is the collection of all the attributes types, $\mathcal{T}$ is a function that assigns a type from T to an attribute, i.e. $\mathcal{T}(X \cdot a) \in T$. E is the set of *semantic function definitions* $(X_{p,i} \cdot a, \lambda)$, providing a definition λ for *attribute occurrence* $(X_{p,i} \cdot a)$. ■

Set A can be partitioned in sets representing, for each $X \in N$, the attributes of X:

$$A_N(X) = \{\ X \cdot a \ \mid\ X' \cdot a \in A, X' = X\ \} \tag{2.1}$$

Set $A_N(X)$ is partitioned further into the sets of inherited and synthesized attributes of X, $AI_N(X)$ and $AS_N(X)$ respectively. When an attribute is both inherited and synthesized we call it a *chained attribute*.

For each production $p \in P$ we gather all the *attribute occurrences*.

$$A_P(p) = \{\ X_{p,i} \cdot a \ \mid\ X_{p,i} \in (\{lhs(p)\} \cup rhs(p)),\ X = X_{p,i},\ X \cdot a \in A_N(X)\ \} \tag{2.2}$$

Two attribute occurrences are *clones* if they are different occurrences of the same attribute.

Definition 2.4 (Clones) Attribute occurrences $(X_{p,i} \cdot a)$ and $(X_{q,j} \cdot b)$ are *clones* if and only if $X_{p,i} = X_{q,j}$, $p \neq q \vee i \neq j$ and $a = b$. The set of all clones of $(X_{p,i} \cdot a)$ is defined as:

$$\begin{aligned} clones(X_{p,i} \cdot a) = \{\ X_{q,j} \cdot b \ \mid\ q \in P,\ X_{q,j} \in (\{lhs(q)\} \cup rhs(q)),\ X_{p,i} = X_{q,j} = X, \\ (X \cdot b) \in A,\ a = b,\ p \neq q \vee i \neq j\ \} \end{aligned} \tag{2.3}$$

■

Two attribute occurrences are *siblings* if they are different occurrences at the same field:

Definition 2.5 (Siblings) Attribute occurrences $(X_{p,i} \cdot a)$ and $(X_{q,j} \cdot b)$ are *siblings* if $p = q$, $i = j$ and $a \neq b$. ■

Definition 2.6 (Clone pair) A pair of attribute occurrences (a', b') is a *clone-pair* of (a, b) if a' is a clone of a, b' is a clone of b and (a', b') is a pair of siblings. ■

Every production rule of the CFG is extended with *semantic function definitions*, defining how to compute the value of an associated attribute occurrence. These definitions form the set E.

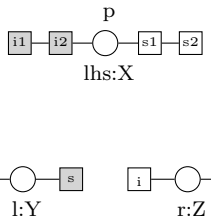

Figure 2.12: Input occurrences have a white background, output occurrences have a gray background.

Set E is partitioned into subsets $E_P(p)$, for each $p \in P$, denoting the semantic functions defined at production p. Furthermore, we define for every production the set of *input occurrences*:

$$\begin{aligned} O_{in}(p) = &\{\ X_{p,0} \cdot a \ \mid\ X \cdot a \in AI_N(lhs(p))\ \} \\ &\cup \{\ X_{p,i} \cdot a \ \mid\ X_{p,i} \in rhs(p),\ X = X_{p,i},\ X \cdot a \in AS_N(X))\ \} \end{aligned} \tag{2.4}$$

The remaining attribute occurrences $A_P(p) \ni X_{p,i} \cdot a \notin O_{in}(p)$, are the *output occurrences* of p:

$$\begin{aligned} O_{out}(p) = &\{\ X_{p,0} \cdot a \ \mid\ X \cdot a \in AS_N(lhs(p))\ \} \\ &\cup \{\ X_{p,i} \cdot a \ \mid\ X_{p,i} \in rhs(p),\ X = X_{p,i},\ X \cdot a \in AI_N(X))\ \} \end{aligned} \tag{2.5}$$

AG descriptions

An AG *description* is a source text from which an AG compiler obtains all the information of an AG. The notation in which AG descriptions are written in this thesis has been introduced in Section 2.1. The notation consists of three parts: a set of datatype definitions and their constructors (CFG), a set of attributes assigned to every datatype (non-terminal) and a set of semantic function definitions for every constructor (production rule). Every datatype is a possible start symbol of the CFG. The UUAGC accepts this notation and checks whether the description is complete and normalised (defined soon). If it is not, the UUAGC either warns the programmer or reports an error. If it is, the UUAGC applies certain preprocessing steps and calls one of set of a code generating procedures (which one is determined by the programmer). What is means for an AG description to be complete and normalised is defined next.

Given a function $\mathcal{P}_\lambda$ to obtain from a semantic function definition λ the attribute occurrences used in λ, we can define the set of occurrences on which $X_{p,i} \cdot a$ *depends*.

$$SF_P(X_{p,i} \cdot a) = \{\ X_{p,j} \cdot b \ \mid\ (X_{p,i} \cdot a, \lambda) \in E_P(p),\ X_{p,j} \cdot b \in \mathcal{P}_\lambda(\lambda),\ X_{p,j} \cdot b \in O_{in}(p)\ \} \tag{2.6}$$

The occurrences in $SF_P(p)$ have to be input occurrences of p. Hence, we only consider AG descriptions that are written in Bochmann normal form.

Definition 2.7 (Bochmann Normal Form) An AG description is written in *Bochmann Normal Form* or *BNF*, if only input occurrences are used are in the right-hand side of semantic functions definitions, i.e. $\forall (p \in P, (X_{p,i} \cdot a) \in A_P(p))\ (SF_P(X_{p,i} \cdot a) \cap O_{out}(p) = \emptyset)$ ■

We restrict AG descriptions further, by saying that all descriptions must be *normalised*.

Definition 2.8 (Normalised Attribute Grammar description) An AG description is *normalised* if it is written in Bochmann Normal Form and if only output occurrences have semantic function definitions, i.e. $\forall(p \in P, (X_{p,i} \cdot a) \in O_{in}(p))\ (\neg\exists\lambda(X_{p,i} \cdot a, \lambda) \in E_P(p))$ ■

An AG description is complete if for all p, all output occurrences of p have some definition λ in $E_P(p)$.

Definition 2.9 (Complete Attribute Grammar description) A *complete AG description* is a description for which holds that $\forall(p \in P, X_{p,i} \cdot a \in O_{out}(p))\ (\exists\lambda(X_{p,i} \cdot a, \lambda) \in E_P(p))$. ■

Definition 2.10 (Well-Typed Attribute Grammar description) A *well-typed AG description* is a description for which holds that, given a function $\mathcal{T}_\lambda$, which assigns a type to every semantic function definition λ, $\forall((X_{p,i} \cdot a, \lambda) \in E)\ (\mathcal{T}_\lambda(\lambda) = \mathcal{T}(X_{p,i} \cdot a))$ ■

AG evaluators

The code generation procedures in the UUAGC generate a function Φ, called an *evaluator*, that executes the semantics defined in the AG description.

Evaluator Φ receives two parameters δ and $\mathcal{I}$. Parameter δ is a valid derivation tree of the CFG. If R is an instance of non-terminal X and the root node of δ, then $\mathcal{I}$ must contain values for all inherited attributes of R, i.e. $AI_N(X)$.

Evaluator Φ returns a set $\mathcal{S}$ that contains values for all synthesized attributes of R, i.e. $AS_N(X)$. To do so, Φ evaluates all attributes of all nodes of δ. This process is called *decoration*[2].

During decoration, Φ performs a treewalk τ over δ by executing a sequence of *visits*. At every point of its execution Φ is currently performing visit v to a node K. Besides evaluating instances of attribute occurrences of the production p, used to derive K, Φ might perform a visit to either child of p or suspend visit v and continue with the visit v' (of the parent of K) from which the visit v originated (if K is the root of δ, decoration is completed by suspending v). If K is an instance of non-terminal X, we say that visit v is a pair $\mathcal{I} \times \mathcal{S}$, with $\mathcal{I} \subseteq AI_N(X)$ and $\mathcal{S} \subseteq AS_N(X)$, where $\mathcal{I}$ represents parameters (attributes it receives) and $\mathcal{S}$ represents results (attributes it returns) of the visit. When visit v is the i-th visit to K we subscript v with i, i.e. $v = (\mathcal{I}_i, \mathcal{S}_i)_i$.

In this thesis we consider algorithms that generate evaluators for AG descriptions α, capable of decorating all possible derivation trees of the CFG defined in α. The algorithms can therefore only rely on information in the description: the evaluator is computed statically.

2.2.3 Orders

Definition 2.11 (Partial order) A *partial order* is a binary relation $\sqsubseteq$, with the properties:

- $a \sqsubseteq a$ (*reflexivity*)
- If $a \sqsubseteq b$ and $b \sqsubseteq a$ then $a = b$ (*antisymmetry*)

[2]Not all attributes of all nodes of a derivation tree have to be evaluated in order to evaluate the synthesized attributes of the root node. The exception have been left out of the definition for simplicity. Which attributes do not have to be evaluated is discussed in Chapter 6.

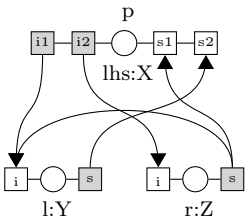

Figure 2.13: A dependency graph for a production.

- If $a \sqsubseteq b$ and $b \sqsubseteq c$ then $a \sqsubseteq c$ (*transitivity*)

■

Definition 2.12 (Linear order) A *linear order* or *total order* is a binary relation $\leqslant$, with the properties:

- $a \sqsubseteq b$ or $b \sqsubseteq a$ (*totality*)
- If $a \sqsubseteq b$ and $b \sqsubseteq a$ then $a = b$ (*antisymmetry*)
- If $a \sqsubseteq b$ and $b \sqsubseteq c$ then $a \sqsubseteq c$ (*transitivity*)

■

Every linear order is a partial order, as totality implies reflexivity. We say that $(a \sqsubset b) \in O$ if and only if $(a \sqsubseteq b) \in O$ and $(b \sqsubseteq a) \notin O$.

2.2.4 Dependency graphs

Definition 2.13 (Dependency graph) A *dependency graph* is a graph $G = \langle V, E \rangle$ where each vertex $v \in V$ is either an attribute or an attribute occurrence and an edge from a to b, i.e. $(a \to b) \in E$, denotes that attribute (occurrence) b depends on attribute (occurrence) a. The edges in G are called *dependencies*. ■

The direction $\to$ in edge $(a \to b)$, denoting that attribute b depends on an attribute a, corresponds to the *information flow* between attributes (occurrences). Figure 2.13 shows an example of a dependency graph.

Chapter 3

Attribute Grammar Classes

All possible Attribute Grammar descriptions from the class AG. We have restricted this class by saying that we only consider AG descriptions that are *normalised*. By placing further restrictions other AG classes are defined. This chapter introduces the AG classes of interest in this thesis: the Absolutely Non-Circular Attribute Grammars (ANCAGs), Linear Ordered Attribute Grammars (LOAGs), Partitionable Attribute Grammers (PAGs), Ordered Attribute Grammars (OAGs) and Arranged Orderly Attribute Grammars (AOAGs).

AG classes are sets of AG descriptions. Using subset and equality relation we demonstrate the hierarchy between classes: OAG $\subset$ PAG = AOAG = LOAG $\subset$ ANCAG $\subset$ AG.

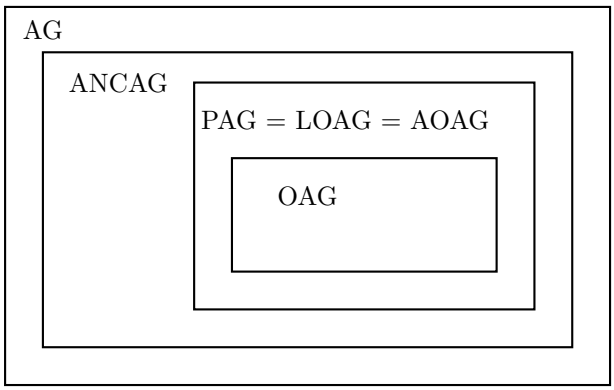

Figure 3.1: Schematic hierarchy of the AG classes relevant to this thesis.

Each of the classes is introduced by their distinctive properties, i.e. the properties an AG should satisfy to be a member of that class. These properties determine what type of *evaluator* we can generate for the AG descriptions in a particular class.

Algorithms for testing membership of classes AOAG and LOAG are the topics of Chapters 4 and 5 and are the main result of this thesis.

3.1 Lazy evaluation

The definition of an evaluator (page 20) allows cyclic evaluators. This is not necessarily a problem when the evaluator is executed in a lazy evaluation scheme. If evaluation is possible, the correct

evaluation order is found at runtime. If evaluation is not possible, the evaluator will (report a) loop. Lazy evaluators can be generated as catamorphisms using folds and algebras[SAS99].

To prevent cyclic evaluators, a static analysis on the dependencies of AGs is performed. If a cycle is detected, it is reported and no evaluator is generated. If no cycle is detected, a *strict* evaluator is generated. A static analysis for detecting dependency cycles also prevents the generation of cyclic evaluators that *can* be executed using lazy evaluation: the static analysis is pessimistic.

Detecting dependency cycles can be done in multiple ways, each way defining their own subclass of AG descriptions. In the next section we define the class of Absolutely Non-Circular Attribute Grammars.

3.2 Absolutely Non-Circular Attribute Grammars

Absolutely Non-Circular Attribute Grammars, introduced by Kennedy and Warren[KW76], form the largest class of AG descriptions for which strict evaluators can be generated. An AG is cyclic *directly* if it has an attribute occurrence that is defined, possibly through a sequence of other occurrences, in terms of itself. An AG can also contain *induced* cycles. Induced cycles manifest themselves as cycles spanning attribute instances of nodes of a derivation tree, such that the instances are instances of attribute occurrences of multiple production rules. Both direct and induced cycles might occur in one derivation tree but not in the other. Therefore we have to be pessimistic when looking for cycles at compile time.

Dependency graphs are used to detect cycles. The next section shows how to calculate a dependency graph from the semantic function definitions for any AG, given the sets O_{out} (equation 2.5) and SF_P (equation 2.6) defined in section 2.2.2.

3.2.1 Direct dependencies

For each production p we introduce the *direct dependency graph* D_P and draw an edge between occurrence a and occurrence b if there is a semantic function defined at p which defines b in terms of a.

$$D_P(p) = \{\ (X_{p,i} \cdot a \to X_{p,j} \cdot b)\ \mid\ X_{p,j} \cdot b \in O_{out}(p),\ X_{p,i} \cdot a \in SF_P(X_{p,j} \cdot b)\ \} \tag{3.1}$$

Graph $D_P(p)$ can only contain edges to inherited attributes of children of p and edges to the synthesized attributes of the root of p: the output occurrences of p. Note that by the use of SF_P it is implied that $X_{p,i} \cdot a$ must be an input occurrence of p. From the definition of D_P it follows that all AGs we consider are normalised. We refer to a cycle in D_P as a *type 1 cycle*[RT89, Sar99, NGI+99]. For AG descriptions written in Bochmann normal form no type 1 cycles can occur.

3.2.2 Induced dependencies

Direct dependencies in one production, following directly from the semantic function definitions, influence dependencies in other productions, which requires us to define *induced dependencies*. A graph containing induced dependencies is called an *induced dependency graph*. To check for paths in a graph G we check for an edge in its *transitive closure* G^+. The induced dependency graph is used to find the induced cycles mentioned before.

Kennedy & Warren define i/o (input/output) graphs for each non-terminal and use them to define the induced dependency graph DG_P. They say that $(a \to b) \in IO_X(X)$ when there is a path from inherited occurrence a to synthesized occurrence b in the graph graph $DG_P^+(p)$, $\forall(p \in P)\ (X = lhs(p))$, in which a and b are occurrences of the parent of p. The induced dependency graph $DG_P(p)$ is constructed by adding to $D_P(p)$, all the edges from the i/o graphs of the non-terminals of which the children of p are occurrences[KW76].

$$DG_P(p) = D_P(p) \cup \{\ (X_{p,i} \cdot a \to X_{p,i} \cdot b)\ \mid\ X_{p,i} \cdot a \in A_P(p),\ X_{p,i} \cdot b \in A_P(p),\ i > 0,\ X = X_{p,i},\ (X \cdot a \to X \cdot b) \in IO_X(X)\ \} \tag{3.2}$$

$$IO_X(X) = \{\ (X \cdot a \to X \cdot b\ \mid\ X \cdot a \in AI_N(X),\ X \cdot b \in AS_N(X),\ p \in P,\ X = lhs(p),\ (X_{p,0} \cdot a \to X_{p,0} \cdot b) \in DG_P^+(p)\ \} \tag{3.3}$$

Definition 3.1 (Absolutely Non-Circular Attribute Grammar) An AG is an *Absolutely Non-Circular Attribute Grammar* or *ANCAG*, if and only if the induced dependency graph DG_P is cycle free. ■

An acyclic graph DG_P is a precondition for scheduling ANCAGs. If the precondition is not met, the Kennedy-Warren algorithm is not able to guarantee an evaluation order exists. If the precondition is met, a strict evaluation of all attributes is possible. The algorithm finds all the visits that, based on the knowledge of the AG description, might be required during the decoration of all possible derivation trees. At runtime the largest possible visit will be chosen as a next step, based on the knowledge which attributes have been evaluated so far (thus evaluating as much attribute instances as possible at every step). The algorithm guarantees that there is an action available to the evaluator in any possible situation the evaluator 'might find itself in'. The exact order in which the attribute instances are evaluated is not know at compile time. To find this order statically, we need to restrict AG descriptions further.

3.3 Linear Ordered Attribute Grammars

By restricting AGs further we gain more information on the AGs that can be exploited by evaluators generated for these AGs. Just as the restrictions for ANCAGs guarantee that its evaluators can be strict, the restrictions on the class of *Linear Ordered Attribute Grammars* (LOAGs) guarantee that for any derivation tree of the grammar a total evaluation order for its attribute instances can be found statically. This restriction demands that any evaluator is able to evaluate the attribute instances of a node K by performing the same sequence of visit to child f of K, *irrespective* of the production used to derive f. Evaluators for LOAGs are strict and efficient. The static evaluation order is implicitly encoded in the evaluators, such that their executions follows a linear order.

Definition 3.2 (Linear Ordered Attribute Grammar) An AG $\langle G = \langle \Sigma \cup N, P, S \rangle, A, D \rangle$ is a *Linear Ordered Attribute Grammar* or *LOAG*, if and only if for any possible *derivation tree* δ of G, there exists a linear order $LO(\delta)$ and linear orders $\forall(X \in N)\ (LO(X))$, such that:

- If a and b are attribute occurrences and b depends on a in D then $(a \sqsubset b) \in LO(\delta)$ (*feasibility*)

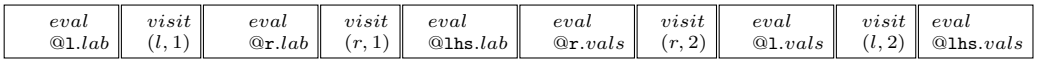

Figure 3.2: Possible visit-sequence for production *Bin* of LABEL-VALUE.

Figure 3.3: Possible visit-sequences for production *Leaf* of LABEL-VALUE.

- If a and b are attributes of non-terminal X and $(a \sqsubset b) \in LO(X)$ then $(a \sqsubset b) \in LO(\delta)$ ($LO(X)$ is *respected*)

■

We call the problem of deciding whether an AG is linear ordered, the problem of *linear orderedness*. Engelfriet and Filè[EF82] proved that linear orderedness is *NP-complete*: no polynomial runtime algorithm for deciding whether an AG is an LOAG exists, when $P \neq NP$. They showed that the problem of Boolean *satisfiability* is an instance of linear orderedness.

We investigate the problem of linear orderedness in this section to find intuitions for its NP-hardness. First of all we need to prove that a linear order exists for any derivation tree of the grammar. To do this, we define *visit-sequences* for productions and *interfaces* for non-terminals.

3.3.1 Visit-sequences

A *visit-sequence* is a sequence of instructions used by the evaluators generated for LOAGs. One or more visit-sequences are generated for every production. An evaluator assigned with the task of evaluating the attribute instances of a non-terminal instance K derived by production p, performs the actions from the v-th visit-sequence of p, whenever it is performing the v-th visit of the evaluator to K. The possible instructions in a visit-sequence are:

- $eval(X_{p,i} \cdot a)$: evaluate the attribute instance of occurrence $(X_{p,i} \cdot a) \in O_{out}(p)$.
- $visit(f, v_f)$: descend to child node f of K for the v_f-th time and proceed decoration at f by executing the v_f-th visit-sequence of the production used to derive f.
- $suspend(v)$: end the v-th visit to K and proceed decoration at the parent of K. If K is the root node, decoration is completed.

We typically speak of a set of visit-sequences for every production, although by using the suspend-instruction only a single visit-sequence is required. The suspend-instructions 'glue' visit-sequences together. We decide to ignore suspend-instructions and only use them to informally refer to the action an evaluator performs when execution a suspend-instruction.

For every visit to a node K there should be a corresponding visit-sequence for the production that was used to derive K. Since a root node has no parent, and only one visit will be performed to it (evaluator application), a root node will have only one visit-sequence. Examples of visit-sequences for the LABEL-VALUE grammar are given in figures 3.2 and 3.3.

Plan tree

Say that we have a set of visit-sequences for all $p \in P$ and a derivation tree δ. If we replace every node K in δ by the visit-sequences for the production that was used to derive K we obtain a *plan*

tree[Pen94]. We can find a linear order on all the attribute instances of δ, that are instances of occurrences with an eval-instruction in one of the visit-sequences, using the plan tree. Starting from the left-most instruction of the visit-sequence at the root of the plan tree, we traverse all visit-sequences from left to right. If a visit-instruction is encountered, continue with left-most instruction of the visit-sequences corresponding to the visit-instruction. If all instructions of a visit-sequence s have been examined, then continue with the visit-instruction of the parent corresponding to the visit-sequence s. Traverse the entire plan tree like this and the order in which eval-instructions are encountered is the linear order on attribute instances.

Complete and well-defined visit-sequences

A visit-sequences s for production p, corresponding to a visit $v = (\mathcal{I}_i, \mathcal{S}_i)_i$, is *complete* if s contains an eval-instruction for every attribute in $\mathcal{S}_i$, i.e. $\forall(X \cdot a \in AS_N(lhs(p)))\ (eval(X_{p,0} \cdot a) \in s)$.

Say that a visit-sequence s for production p *contains* attribute occurrence a if there is an eval-instruction for a in s or if there is a visit $(\mathcal{I}_i, \mathcal{S}_i)_i$ with $a \in \mathcal{S}_i$. Visit-sequence s is *well-defined* if for every attribute occurrence b contained in s and depending on occurrences a, all occurrences a are contained in s such that the corresponding eval or visit-instruction is executed before the eval or visit-instruction of b in s (note that this is an inductive definition).

Complete and well-defined visit-sequences are used to find a *feasible* linear order on the attribute instances of any derivation tree (first part of Definition 3.2 of LOAGs). The order is obtained from a plan tree as explained above.

The next step is to show what is needed for the linear order to *respect* a certain set of linear orders, one for every non-terminal (second part of Definition 3.2 of LOAGs). In the next section we will define *interfaces* from which orders on the attributes of every non-terminal are obtained. How to generate evaluators, and the visit-sequences they require, is the subject of Chapter 6.

3.3.2 Interfaces

A process p that is currently decorating a derivation tree *relays* control to another process q when executing a *visit*-instruction. The child-process q will, in turn, *return* control to p by executing a *suspend*-instruction. Between these two instruction, exactly one visit-sequence s had its instructions performed by the child process. Say that visit-sequence s corresponds to the $visit(K, v)$-instruction, where K is an instance of non-terminal X. Between $visit(K, v-1)$ and $visit(K, v)$ some set $\mathcal{I} \subseteq AI_N(X)$ is evaluated. The values of the attributes in $\mathcal{I}$ are made available as parameters to the process q. By executing visit-sequence s, process q evaluates some set $\mathcal{S} \subseteq AS_N(X)$. The pair $(\mathcal{I}, \mathcal{S})$ uniquely defines a visit to K. This visit can be seen to describe the communication between process p and q. We say that all visits of K form an *interface* of K.

The definition of LOAGs determines that every visit to a node K should be made irrespective of which production was used to derive K. We conclude that for LOAGs every instance (and therefore every occurrence) of a non-terminal $X \in N$ needs to have the same interface. To make sure all direct dependencies of all occurrences of X are reflected in the interface for X, we define an alternative induced dependency graph $ID_S(X)$ in the next section. First we define interfaces formally using ID_S.

Definition 3.3 (Interface) A partitioning P_N of the attributes of an non-terminal X is an *interface of non-terminal* X, when for each visit $(I_i, S_i)_i \in P_N(X)$ holds that $I_i \subseteq AI_N(X)$ and $S_i \subseteq AS_N(X)$. The interface is *complete* when all attributes of X are part of the partition, i.e. $\bigcup_{(I_i,S_i)_i \in P_N(X)}(I_i) = AI_N(X)$ and $\bigcup_{(I_i,S_i)_i \in P_N(X)}(S_i) = AS_N(X)$. An interface P_N is *well-defined*, if for each dependency $(X \cdot a \to X \cdot b) \in ID_S(X)$ holds that: if $(X \cdot b) \in (I_i \cup S_i)$ for

some visit $(I_i, S_i)_i \in P_N(X)$, then $(X \cdot a) \in (I_j \cup S_j)$ for some visit $(I_j, S_j)_j \in P_N(X)$, with $j \leqslant i$ when $(X \cdot b) \in S_i$ and $j < i$ when $(X \cdot b) \in I_i$. ■

3.3.3 Induced dependencies

To construct interfaces that are complete and well-defined we must use an alternative version of an *induced dependency graph* (alternative to DG_P for ANCAGs, from equation 3.2): if there is a path between siblings $X_{q,j} \cdot a$ and $X_{q,j} \cdot b$ in the induced dependency graph of some production q, then we need an induced dependency $a' \to b'$ for every clone-pair (a', b') of (a, b).

The following definition is used by Kastens[Kas80]:

$$ID_P(p) = D_P(p) \cup \{\ (X_{p,i} \cdot a \to X_{p,i} \cdot b)\ \mid\ q \in P,\ (X_{q,j} \cdot a \to X_{q,j} \cdot b) \in ID_P^+(q),\ X_{p,i} = X_{q,j}\ \} \tag{3.4}$$

We refer to cycles in ID_P as *type 2 cycles*. Just like graph DG_P for the Kennedy-Warren algorithm, an acyclic graph ID_P serves as a precondition for an AG to be an LOAG. By testing for type 2 cycles first, a large group of AGs is ruled out immediately.

The induced dependency graphs DG_P and ID_P are different in the following aspects. Kastens' method:

1. considers paths from synthesized to inherited attribute occurrences.
2. considers paths in child nodes.
3. adds induced dependency not only to child nodes, but also to parent nodes.

We conclude that $DG_P \subseteq ID_P$, that a cycle in DG_P is a cycle in ID_P and that if ID_P is cycle free, so should DG_P. An AG with a cycle free induced dependency graph ID_P is therefore a member of the class ANCAG.

Non-terminal dependencies

For every non-terminal X we gather all dependencies between occurrences of X in ID_P and add them to the induced dependency graphs for non-terminals, ID_S (S for symbol). Clone-pairs should share their dependencies ($a \to b$, $b \to a$, or a and b are independent), for it is these dependencies that must be respected by a well-defined interface.

$$ID_S(X) = \{\ (X \cdot a \to X \cdot b)\ \mid\ \forall (p \in P)\ ((X_{p,i} \cdot a \to X_{p,i} \cdot b) \in ID_P(p),\ X = X_{p,i})\ \} \tag{3.5}$$

3.3.4 Intra-visit dependencies

Given a set of complete and well-defined interfaces $P_N(X)$, one for every $X \in N$, we can generate a set complete and well-defined visit-sequences for every production. The visit-sequences must respect the interfaces. In other words, the visit-sequences can contain a visit-instruction for a visit v to a field of type X, if and only if v is an element of the interface of X, i.e. $v \in P_N(X)$.

A complete procedure to construct visit-sequences from interfaces will be given in Chapter 6. For now it suffices to say that every visit-sequence s for production p, implementing a visit $v = (\mathcal{I}_i, \mathcal{S}_i)_i$, must contain an eval-instruction for all attributes in $\mathcal{S}$ (to be complete). In order

to be well-defined, s must also contain every attribute occurrence on which the occurrences in $\mathcal{S}$ depend. Say that $s_r \in \mathcal{S}_i$ and $(s_1 \to s_r) \in D_P(p)$, where s_1 is a synthesized attribute of a child node in p. In order to be well-defined, s must contain the visit-instruction v that produces s_1. If $v = (\{i_1\}_i, \{s_1, s_2\}_i)_i$ then v requires the value of i_1 and produces the value of s_2 as well as the value of s_1. Any attribute depending on s_2 will now also depend on i_1, even if $(i_1 \to s_2) \notin D_P(p)$. By constructing an interface with visit v we have introduced a new dependency! Such *intra-visit dependencies* are produced only by fixing the interfaces.

The intra-visit dependencies might lead to new cycles. If a combination of interfaces results in cycles with intra-visit dependencies then the visit-sequences generated using these interfaces can not be well-defined. Therefore we need to take intra-visit dependencies into account when constructing the interfaces.

$$\begin{aligned} IVD(X) =& \{\ X \cdot a \to X \cdot b \mid \forall((I_i, S_i)_i \in P_N(X)),\ X \cdot a \in I_i,\ X \cdot b \in S_i\ \} \\ & \cup \{\ X \cdot a \to X \cdot b \mid \forall((I_{i+1}, S_{i+1})_{i+1}, (I_i, S_i)_i \in P_N(X)),\ X \cdot a \in S_{i+1},\ X \cdot b \in I_i\ \} \end{aligned} \tag{3.6}$$

3.3.5 Compatible interfaces

To prevent cycles caused by intra-visit dependencies we introduce the notion of compatible interfaces.

Definition 3.4 (Compatible interfaces) A set of interfaces $\{\ P_N(X) \mid \forall X \in N\ \}$ contains *compatible* interfaces if the direct dependency graph extended with the intra-visit dependencies imposed by the interfaces is acyclic, i.e. the graph $D_P(p)[P_N(X_{p,0}), \ldots, P_N(X_{p,|p|})]$ is cycle free, with

$$\begin{aligned} & D_P(p)[P_N(X_{p,0}), \ldots, P_N(X_{p,|p|})] = \\ & \quad D_P(p)\ \cup \\ & \quad \{\ X_{p,i} \cdot a \to X_{p,i} \cdot b \mid 0 \leqslant i \leqslant |\ p\ |,\ X = X_{p,i},\ (X \cdot a \to X \cdot b) \in IVD(X)\ \} \end{aligned} \tag{3.7}$$

■

3.3.6 Generating interfaces from direct dependencies

So far we have looked at the problem of linear orderedness bottom-up and concluded that to show that a feasible linear order exists between the attribute instances of any derivation tree it suffices to show that we can generate complete, well-defined and compatible interfaces. In this section we show how to construct complete and well-defined interfaces from the direct dependencies.

From the direct dependency graph D_P an induced dependency graph ID_P is constructed using Equation 3.4. The induced dependency graph guarantees that all clone-pairs share the same dependency information. From ID_P the graph $ID_S(X)$ is constructed for every non-terminal $X \in N$ using Equation 3.5. Graph $ID_S(X)$ contains all the dependencies that well-defined interfaces need to take into account. In graph $ID_S(X)$ there are certain attributes that have no outgoing edges. We call these attributes *carefree* as no other attributes depend on them. All paths in $ID_S(X)$ must end in a carefree attribute as there are no cycles (a precondition for LOAGs). The graph of all paths that end in the same carefree attribute we call a thread.

Definition 3.5 (Thread) A *thread* of non-terminal X is the set of paths in graph $ID_S(X)$ that end in the same carefree attribute. A *carefree attribute* is an attribute on which no other

attributes depend. If t is a carefree attribute, then the set of paths ending in t is called the *thread of attribute t*. ■

Every thread of t in $ID_S(X)$ can be transformed into an interface, as follows:

- If t is a synthesized attributes, assign it to set S_1. Assign it to I_1 otherwise. Proceed by assigning every attribute a for which holds that $(a \to t) \in ID_S(X)$, as follows:
  - If all attributes b for which holds that $(a \to b) \in ID_S(X)$ are assigned, then find the highest i for which $b \in (I_i \cup S_i)$. If $b \in I_i$ and a is inherited then assign a to I_i, if a is synthesized then assign a to S_{i+1}. If $b \in S_i$ and b is synthesized then assign a to S_i, if a is inherited then assign a to I_i. Proceed by assigning all attributes that point to a.

The sets I_i and S_i form the i-th visit of interface constructed from the thread. By applying this procedure to every thread of X, a set of disjoint interfaces is obtained for X. Together the interfaces are complete, as every attribute in $ID_S(X)$ is either carefree, or part of a path leading to a carefree attribute. The interfaces are also well-defined, as all the dependencies from $ID_S(X)$ have been respected. Figure 3.5 shows how we depict interfaces graphically.

Figure 3.4 shows the progress we have made in determining the required constructions to guarantee a linear evaluation order for any derivation tree. There is one missing link. How can we construct a set of interfaces $P_N(X)$, one for every $X \in N$, from a set of interfaces for X, one for every thread of X, such that the interfaces P_N are compatible?

We argue that this step is the difficult step, causing the NP-hardness of deciding linear orderedness. Fixing the interface for a non-terminal X influences the set of possible interfaces for other non-terminals Y as the intra-visit dependencies imposed by the interface for X might induce new dependency paths that have to be taken into account in the interface of Y. Otherwise X and Y might not be *compatible*.

3.3.7 Partitionable Attribute Grammars

The notion of complete, well-defined and compatible interfaces allows us to define a new class of AGs, namely the *Partitionable Attribute Grammars*. In section 3.4 we see that Kastens' algorithm is an approximation algorithm for recognising PAGs.

Definition 3.6 (Partitionable Attribute Grammar) An AG $\langle G = \langle \Sigma \cup N, P, S \rangle, A, D \rangle$ is a *Partitionable Attribute Grammar* or *PAG* if and only if we can construct *compatible*, *complete* and *well-defined* interfaces, such that for every production $p : X_{p,0} \to X_{p,i}, \ldots, X_{p,|p|}$the graph $D_P(p)[P_N(X_{p,0}), \ldots, P_N(X_{p,|p|})]$ is cycle free. ■

Interfaces impose a partial order on the attributes and since every linear order is a partial order, it is implied that LOAG $\subseteq$ PAG. In fact, by assuming AGs written in Bochmann Normal Form, the two classes are equal[Pen94].

3.4 Ordered Attribute Grammars

Kastens provides seven steps for determining whether an AG is of the class of *Ordered Attribute Grammars* or *OAG* (a strict subclass of PAG)[Kas80]. The graphs D_P, ID_P and ID_S, defined earlier in this chapter, are the results of the first three steps. We continue with the fourth step, in which interfaces are derived from ID_S. This step inspired the more general procedure for generating interfaces from threads described previously in Section 3.3.6.

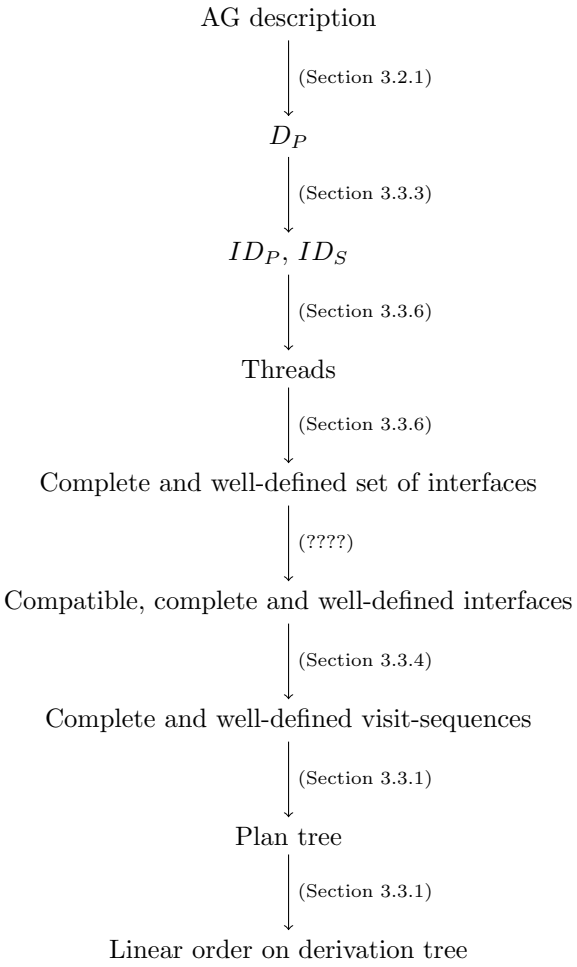

Figure 3.4: The constructions we use to guarantee a linear order on attribute instances of any derivation tree. Every arrow is attached with the section number in which the transition of one construction to another is defined.

3.4.1 Step 4 - Disjoint set partitioning P_S

A disjoint set partitioning $P_S(X, k)$ is created of all attributes of a non-terminal X. The partitions are indexed by k. The attributes are assigned to the partitions such that the partitions with an even index contain only inherited attributes partitions with an uneven index contain only synthesized attributes. The partitioning relates directly to an interface. If an attribute a is assigned to partition $P_S(X, k)$, a is part of the i-th visit with $i = \lfloor (k+1)/2 \rfloor$ if k is even then $a \in I_i$, if k is uneven then $a \in S_i$.

$P_S(X)$ is defined such that it respects the dependencies in $ID_S(X)$ by only adding an attribute a to $P_S(X, k)$ if each of the attributes that depend on a is member of a set $P_S(X, k')$ where $k' \leqslant k$. In other words, all dependencies in $ID_S(X)$ have to point from left to right in the graphical representation of the interface (see Figure 3.5). Partition $P_S(X, 1)$ contains only synthesized carefree attributes.

The attributes are assigned to the 'earliest' partition. That is, if of all attributes depending on a, the attribute b is assigned to the partition with the highest index k', then a is assigned

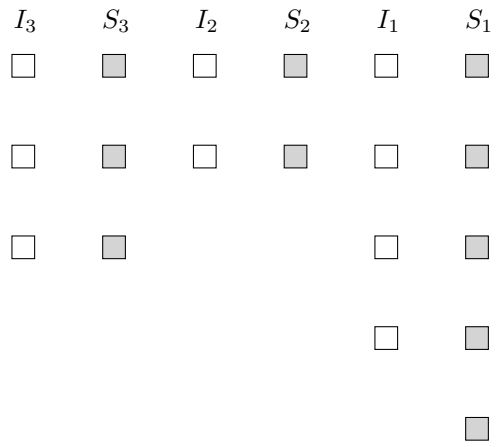

Figure 3.5: A graphical representation of an interface. Every gray column shows the synthesized attributes of a visit, while a white columns shows the inherited attributes of a visit. The intra-visit dependencies implied by this interface are all dependencies we can draw from attributes in a column to the attributes of its right neighbour.

to partition $P_S(X, k')$ if a and b are both synthesized or inherited and a assigned to partition $X_{X,k}(X, k' + 1)$ otherwise. In other words, for every attribute $(X \cdot a)$ there is a dependency $(X \cdot a \to X \cdot b) \in ID_S(X)$ that does not 'skip' a column in the graphical representation of the interface.

$$P_S X, 1) = \{\ X \cdot a \mid X \cdot a \in AS_N(X),\ \neg\exists(X \cdot b)\ ((X \cdot a \to X \cdot b) \in ID_S^+(X))\ \} \tag{3.8}$$

$$\begin{aligned} P_S(X, 2n) = \{\ X \cdot a \mid\ & X \cdot a \in AI_N(X),\ \forall(X \cdot b \in A_N(X), n' \leqslant 2n)\ (\\ & (X \cdot a \to X \cdot b) \in ID_S^+(X) \Rightarrow X \cdot b \in P_S(X, n'))\ \} \\ & \setminus \cup_{i=1}^{2n-1} P_S(X, i) \end{aligned} \tag{3.9}$$

$$\begin{aligned} P_S(X, 2n+1) = \{\ X \cdot a \mid\ & X \cdot a \in AS_N(X),\ \forall(X \cdot b \in A_N(X), n' \leqslant 2n+1)\ (\\ & (X \cdot a \to X \cdot b) \in ID_S^+(X) \Rightarrow X \cdot b \in P_S(X, n'))\ \} \\ & \setminus \cup_{i=1}^{2n} P_S(X, i) \end{aligned} \tag{3.10}$$

3.4.2 Step 5 - Completion

The set partitioning defined in the previous step computes a single interface for every non-terminal by implicitly merging the interfaces constructed for individual threads (the missing link of Figure 3.4). It zips the interfaces together, such that the i-th visit of every interface ends up in the i-th visit of the merged interface. The result is an interface that contains the minimum amount of visits (will be explained in Section 6.4.1). Merging interfaces in this way is optimistic, as it does not take the intra-visit dependencies implied by the interfaces into account. *The resulting interfaces are not guaranteed to be compatible.* They might form a cycle in combination with the direct dependencies at production level such that no well-defined visit-sequences can be constructed.

Whether the intra-visit dependencies cause cycles is tested with the *completion* of ID_S: Let m_X be the maximum number of partitions in $P_S(X)$, i.e. $\forall(X, m' > m_X)\ (P_S(X, m') = \emptyset)$, the

intra-visit dependencies calculated from P_S are:

$$IVD(X) = \{\ (X \cdot a \to X \cdot b)\ \mid\ X \cdot a \in P_S(X,k),\ X \cdot b \in P_S(X,k-1),\ 2 \leqslant k \leqslant m_X\ \} \quad (3.11)$$

The completion of ID_S is then defined as:

$$D_S(X) = ID_S(X) \cup IVD(X) \quad (3.12)$$

It follows that by completing ID_S using P_S we have created a dependency graph D_S which contains all direct, induced and intra-visit dependencies.

3.4.3 Step 6 - Extended dependency relation

Kastens defines the *extended dependency relation* ED_P, to test whether the interfaces represented in P_S are compatible.

$$ED_P(p) = D_P(p) \cup \{\ (X_{p,i} \cdot a \to X_{p,i} \cdot b)\ \mid\ X_{p,i} = X,\ (X \cdot a \to X \cdot b) \in D_S\ \} \quad (3.13)$$

We refer to cycles in ED_P as *type 3 cycles*.

3.4.4 Step 7 - OAGs

Definition 3.7 (Ordered Attribute Grammar) An *Ordered Attribute Grammar* or *OAG* is an AG for which if and only if ED_P is cycle free. ■

Kastens' algorithm was shown not to solve the problem of *linear orderedness* by counter-examples: AGs that were not part of the class OAG but for which compatible, complete and well-defined interfaces can be constructed. Such examples are found in [NGI+99] and in Section 4.1. The Label-Value AG description given in Section 2.1 is also an LOAG/PAG that is not OAG.

3.4.5 Step 8 - AOAGs

Kastens provided an eight step to his algorithm:

Definition 3.8 (Arranged Orderly Attribute Grammar) An AG is an *Arranged Orderly Attribute Grammar* or *AOAG* if and only if it is recognized as an *ordered attribute grammar* by extending its set of direct dependencies with a set *augmenting dependencies* or *fake dependencies*, called *ADS*. ■

Fake dependencies serve only the purpose of forcing the algorithm in choosing different interfaces that are be compatible. The dependencies in *ADS* can be added to source text of the AG description manually, as shown in the next section, or added to D_P directly as if they correspond to semantic function definitions. Deciding which dependencies to add to *ADS* in order to decide whether an arbitrary AG is LOAG is a combinatorial problem. In Chapter 4 a method for selecting fake dependencies is described, resulting in a backtracking algorithm for automatically finding fake dependencies.

It is easy to see that AOAG is a characterisation of the class LOAG. If an AG is LOAG, a linear order on the attribute occurrences of all productions exists. This order is trivially transformed into a set of dependencies *ADS*.

Adding fake 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 expression. Simply add the attribute a of the fake 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`. This method is being used frequently in practice, for example in [NGI+99, BMDS12, RT89]. A difficulty of this approach is that the resulting AG may not be well-typed. Consider the LABEL-VALUE example (Figure 2.7) again, this time written with a fake dependency:

```
data Tree | Leaf val : Int
          | Bin  l   : Tree r : Tree
attr Tree
     inh  label : [Int]
     inh  vals  : [Int]
     syn  label : [Int]
     syn  vals  : [Int]
sem Tree
  | Leaf
    lhs.vals  = if True then@val : @lhs.vals
                        else @lhs.label
    loc.label = @lhs.label
    lhs.label = @loc.label + 1
  | Bin
    r.vals    = @lhs.vals
    l.vals    = @r.vals
    lhs.vals  = @l.vals
    l.label   = @lhs.label
    r.label   = @l.label
    lhs.label = @r.label
```

Figure 3.6: Full AG description of LABEL-VALUE, with fake dependency `Leaf`.`lhs`.*label* → `Leaf`.`lhs`.*vals*

The fake dependency `Leaf`.`lhs`.*label* → `Leaf`.`lhs`.*vals* has been added to the semantic function definition of `Leaf`.`lhs`.*vals*. The AG description of Figure 3.6 will be recognized as an OAG, proving that the AG description from Figure 2.7 is AOAG.

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

$$D'_P(p) = ADS \cup \{\ (X_{p,i} \cdot a \to X_{p,j} \cdot b)\ \mid\ X_{p,j} \cdot b \in O_{out},\ X_{p,i} \cdot a \in SF_P(X_{p,j} \cdot b)\ \} \qquad (3.14)$$

Chapter 4

AOAG Algorithm

This chapter describes an algorithm that computes, for any LOAG, a set of *augmenting* or *fake dependencies*, called ADS, that enables Kastens' algorithm to schedule the grammar. To find this set, Kastens' algorithm is extended to find a set of *candidates*. A backtracking procedure selects from the candidates until a complete set ADS is found. This chapter describes which dependencies are considered as candidates and presents a Haskell implementation of the backtracking algorithm. Additionally, we investigate in which cases candidates are incorrect and require backtracking. We expect that backtracking will be rare for practical AG descriptions.

The chapter is organised as follows: first an AG description of the running example of this chapter is given. From the description we obtain a set of complete and well-defined interfaces after constructing D_P, ID_P and ID_S first (see Figure 3.4). The interfaces are not *compatible*, hence ED_P is cyclic. We then show how a fake dependency can be added to the AG description to force Kastens' algorithm to find compatible interfaces. It is explained how fake dependencies can be found and an algorithm for finding them automatically is presented in the final section.

4.1 iModule

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. It tells us 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 4.1 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 4.1: A simple module defined with iMODULE

```
data Module | Module h  : [TySig] b    : Body
data Body   | Body   ds : [Dat]   fs   : [Fun]
data TySig  | TySig  id : FunId   ty   : [TyId]
data Dat    | Dat    id : TyId    cons : [[TyId]]   -- list of types for every constructor
data Fun    | Fun    id : FunId   def  : FunDef
```

Figure 4.2: Description of the abstract syntax of iModule.

```
attr Module inh ts  : [TyId]                  -- Types from the module's context
            syn ex  : [(FunId, FunDef)]       -- The exported function definitions
            syn err : Bool                    -- Whether the module is valid
            syn ts  : [TyId]                  -- All types used by the module
attr Body   inh ss  : [FunId]                 -- Functions declared in the header
            inh ts  : [TyId]                  -- Types from the module's context
            syn ex  : [(FunId, FunDef)]       -- The exported function definitions
            syn err : Bool                    -- Whether unavailable types where used
            syn ts  : [TyId]                  -- All types used by the module
```

Figure 4.3: Attribute declarations for non-terminals `Module` and `Body`.

We consider two static analyses for iModule, written in a single LOAG. Firstly, we verify that all the type signatures and datatype definitions rely only on types that are available. Secondly, the set exported function definitions is computed. Figure 4.2 defines the abstract syntax of iModule, while Figure 4.3 declares the required attributes for non-terminals `Module` and `Body`.

4.1.1 Semantic functions

With the attributes in place, it is now possible to define the semantics of the iModule system. Figure 4.4 shows dependency graphs for productions ***Module*** and ***Body***. The productions that induce the *induced dependencies* between occurrences of children h, ds, and fs are not shown.

Available Types

First we verify that all the type signatures and datatype definitions use only available types. Types are made available through either:

- the context of the module (for example type literals such as Int, defined in the Prelude), entering ***Module*** as attribute ts(inh).
- datatype definitions in the module's body (attribute ts(syn) of child b).

Child ds in ***Body*** uses the types arriving from the modules context to verify whether its own datatype declarations use only available types (`ds`.$ts \rightarrow$ `ds`.err) and passes the set of types it makes available upwards to the context of the module (`ds`.ts(inh) $\rightarrow$ `ds`.ts(syn) $\rightarrow$ `lhs`.ts(syn)) and to the module's header (`b`.ts(syn) $\rightarrow$ `h`.ts), enabling the header to verify whether its type signatures are valid (`h`.$ts \rightarrow$ `h`.err).

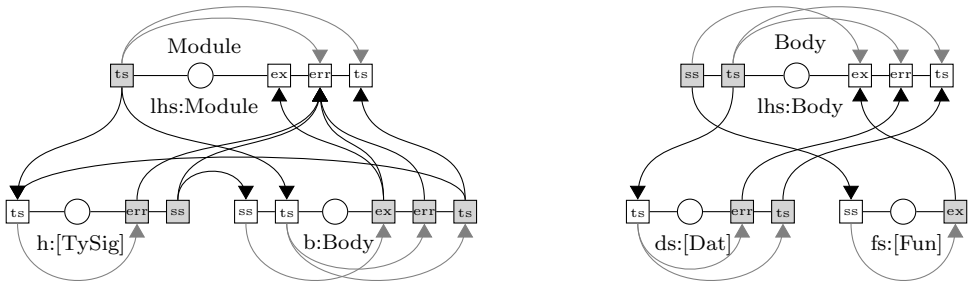

Figure 4.4: Induced dependency graphs for productions *Module* and *Body*.

Exported Definitions

The secondary goal is to construct the list of exported definitions and to test whether this list is complete. The signatures produced by the header are passed downwards into the body of the module ($\texttt{h}.ss \rightarrow \texttt{b}.ss$). *Body*'s fs (function definitions) uses the signatures arriving from the header to produce a list of exports ($\texttt{fs}.ss \rightarrow \texttt{fs}.ex$), by returning only the definitions that are required according to the module's interface. The exports are passed upwards to the module declaration ($\texttt{fs}.ex \rightarrow \texttt{lhs}.ex$). Production *Module* reports an error when the set of *exports* is incomplete or when b or h reports an error ($\texttt{b}.ex \rightarrow \texttt{lhs}.err$, $\texttt{h}.ss \rightarrow \texttt{lhs}.err$, $\texttt{b}.err \rightarrow \texttt{lhs}.err$, $\texttt{h}.err \rightarrow \texttt{lhs}.err$).

Specifying the arrows

We have given a high-level description of a procedure that returns, given a set of types available in context, the definitions of the functional interface and decides whether the module is valid with respect to our rules. The procedure can operate on every possible module defined within our system. For actually generating an evaluator, executing this procedure, our description is too high-level. We need to define the arrows explicitly. Fortunately, the arrows that are required for logistics only (e.g. $\textit{Body}.\texttt{lhs}.ts \rightarrow \textit{Body}.\texttt{ds}.ts$ and $\textit{Module}.\texttt{b}.ex \rightarrow \textit{Module}.\texttt{lhs}.ex$) are generated by the UUAGC. With this feature, we are only concerned with giving the semantic function definitions that are crucial to the semantics we described.

Figure 4.5 adds the crucial semantic function definitions to iModule. The semantic function definitions for the attributes of lists of non-terminal are generated by the UUAGC. For example, the equation for $[\texttt{Dat}].\texttt{lhs}.ts$, of type $[\underline{\text{TyId}}]$, is generated using $(+\!\!+)$ to combine results from individual `Dat`-elements and $[\,]$ as a base element (corresponding to Haskell's monoid instance for lists). The equation for Boolean attribute *err* will be generated using $(\vee)$ and *False*.

Just like the example computations of the introduction (Figure 1.5 the two different computations (static analyses) described above can be written completely independent of each other. Deciding how they are combined, is the task of the algorithm discussed in this chapter. We shall see that Kastens' algorithm tries to schedule the evaluation of the header's synthesized attributes *err* and *ss* in a single visit, while the synthesized attributes *err*, *ex* and *ts* of the body are also evaluated in a single visit. This combination leads to a *type 3 cycle*. The AG described above *is* orderable, however, by adding fake dependencies.

We explained our AG in a step-by-step fashion, which is an indication that an order exists. Although we explained the AG with (parts of) an evaluation order in mind, the semantic function definitions do not enforce that type signatures and datatypes are verified *before* the signatures

```
sem Module | Module
  lhs.err = ¬ (all (∈ (map fst @b.ex)) @h.ss) ∨ @h.err ∨ @b.err
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 Fun | Fun
  lhs.ex  = if (@id ∈ @lhs.ss) then [(@id, @def)] else []
```

Figure 4.5: The crucial semantic function definitions computing the semantics for iMODULE.

are used to construct the list of exports. This separation is made in our explanation and turns out to be required for the AG to be ordered.

The next section shows that Kastens' algorithm fails to schedule the AG for iMODULE and we see how a fake dependency enforces that type signatures and datatypes are verified before the exported definitions are collected. How to automatically find the required fake dependencies is shown in Section 4.3.

4.2 Scheduling iModule

4.2.1 Threads

From ID_S we compute a set of threads (Section 3.3.6). Figure 4.6 shows the threads and interfaces produced by Kastens algorithm for non-terminals [TySig] and Body.

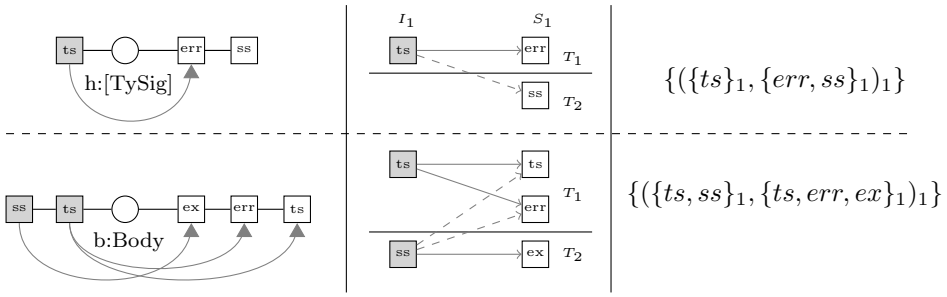

Figure 4.6: Graph ID_S, threads (separated by black horizontal line) and their interfaces, and the visits obtained by merging the interfaces for [TySig] and Body. The gray lines are dependencies in ID_S while the dashed lines are inter-thread dependencies.

4.2.2 Inter-thread dependencies

The decision how to merge the interfaces constructed from threads is the crucial choice that constitutes the NP-hardness of scheduling LOAGs. Different combinations of visits in the interfaces require different *intra-visit dependencies*, possibly contradicting the direct dependencies.

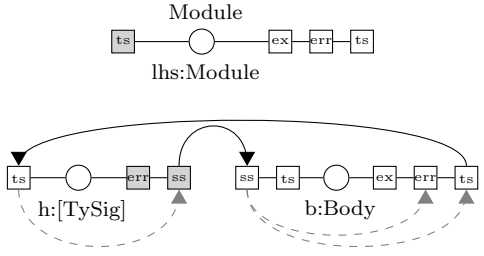

Figure 4.7: Adding the inter-thread dependencies to D_P results in a cycle.

The *inter-thread dependencies* from attributes assigned to one thread to attributes assigned to another thread (dashed gray arrows in Figure 4.6), are exactly those intra-visit dependencies that are *not* part of ID_S. Inter-thread dependencies (`Body`.$ss \rightarrow$ `Body`.ts) and ([`TySig`].$ts \rightarrow$ [`TySig`].ss) implied by these interfaces result in a type 3 cycle (`h`.$ts \rightarrow$ `h`.$ss \rightarrow$ `b`.$ss \rightarrow$ `b`.$ts \rightarrow$ `h`.ts, see Figure 4.7) in the extended dependency relation ED_P, showing that the AG for iMODULE is *not* an OAG.

However, by replacing the calculated interface for non-terminal `Body` with $\{(\{ts\}_2, \{ts, err\}_2)_2, (\{ss\}_1, \{ex\}_1)_1\}$ we find that the AG is an PAG. With this interface the separation of verifying types first and collecting exports from signatures second is made explicit. The next section investigates fake dependencies and shows which fake dependency leads to this interface.

4.2.3 Fake dependencies

In order find compatible interfaces for iMODULE we can either, consider alternative ways of merging the interfaces constructed from threads, or accept the way the interfaces are merged and force different threads instead. The latter is achieved using fake dependencies. The former is done in related work (see Sections 7.1 and 7.2).

The goal of a fake 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, with a different set of intra-visit dependencies to be considered in graphs D_S and ED_P. Moreover, we know that the fake dependency should prevent at least one of the inter-thread dependencies from appearing in that set, to prevent the type 3 cycle. It therefore seems sensible to consider only the reverses of inter-thread dependencies as fake dependencies. Firstly, because it immediately prevents the inter-thread dependency itself and secondly because it can not lead to a type 2 cycle in a obvious way (although it must be possible, as the problem is NP-hard). To support these claims we investigate *candidates* more formally in the next section.

By choosing inter-thread dependency ($a \rightarrow b$) as a fake dependency and add its reverse to ID_S. It forces that a and b become part of the same thread (the threads are merged), such that b is evaluated before a according to the new interface. In our example, Kastens' algorithm can be forced to generate interface $\{(\{ts\}_2, \{ts, err\}_2)_2, (\{ss\}_1, \{ex\}_1)_1\}$ for non-terminal `Body` by adding the fake dependency `b`.ts(syn) $\rightarrow$ `b`.ss. Figure 4.8 shows how the fake dependency is used to force different threads for non-terminal `Body`, relocating ts(inh) and ts(syn) to I_2 and S_2 respectively (in the graphs representing threads no arrows are allowed to point from right to left, according to the procedure described in Section 3.3.6).

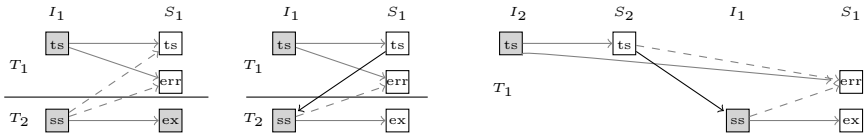

Figure 4.8: Choosing fake dependency $ts(\text{syn}) \rightarrow ss$ results in a different interface.

4.3 Selecting fake dependencies

In this section we zoom in on a set of *sensible candidates* by providing lemmas which are used for ruling out dependencies. We can not hope to find strong enough lemmas to guarantee that our selection of candidates is complete, while ADS is calculated in polynomial time.

Besides assuming that every AG is normalised we make another assumption.

Say that AG α contains type 3 cycle(s) and x and y are dependency between attribute occurrences such that α with $ADS = \{x, y\}$ is an OAG. We assume that α with $ADS = \{x\}$ is either an OAG, or contains a type 3 cycle that *will* be solved by adding y to ADS. In other words: it does not matter whether fake dependencies are used at the same time or consecutively in any order.

LOAG $\subset$ ANCAG

An ADS is *feasible* for AG α if α extended with the dependencies in ADS contains no type 2 cycle. We therefore rule out the reverse of every dependency in ID_P. Additionally, adding a dependency already in ID_P has no effect.

Lemma 4.1. *Any sensible set of fake dependencies will* not *contain an dependency* $(X_{p,i} \cdot a \rightarrow X_{p,j} \cdot b)$ *if* $(X_{p,i} \cdot a \rightarrow X_{p,j} \cdot b) \in ID_P$ *or* $(X_{p,j} \cdot b \rightarrow X_{p,i} \cdot a) \in ID_P$.

It follows that no dependency that is the reverse of an dependency in D_P is considered a sensible candidate ($D_P \subseteq ID_P$), preventing all possible type 1 errors. Note that Lemma 4.1 does not guarantee that all type 2 errors are prevented.

Part of the cycle

When we are a looking for an ADS we are, so we can suppose, testing a class of AGs that is a proper superclass of OAG. Otherwise we have a sufficient ADS for all its members: the empty set. Additionally we assume that any ADS-finding procedure is preceded by a check for type 2 cycles (recall that the absence of type 2 cycles is a precondition for AGs to be LOAG, Section 3.3.3).

All AGs under consideration will then have no type 2 errors and at least one type 3 error and any ADS should satisfy the property that, by using it, at least one dependency in every cycle of ED_P will be replaced by its inverse[Kas80]. Say that $a \rightarrow b$ is an dependency in a type 3 cycle, instead of adding dependency candidate $(b \rightarrow a)$ to ADS, to enforce that $(a \rightarrow b)$ no longer occurs in ED_P, we can also add any clone-pair of (b, a). Adding any of these dependencies will cause $(b \rightarrow a) \in ID_P$, ID_S and ED_P. Since the result of adding any dependency of a group of clone-pairs will be the same, we can decide to use $b \rightarrow a$ directly. Then consider an dependency $c \rightarrow d$, such that adding it to ADS will result in a path $b \longrightarrow a$. Besides forcing that $(b \rightarrow a) \in ID_P$, adding $c \rightarrow d$ might also enforce some other dependency $(e \rightarrow f) \in ID_P$. When $e \rightarrow f$ is the reverse of an edge in some other type 3 cycle, adding edge $c \rightarrow d$, might solve two cycles at once. However, using the assumption at the start of this section, these side-effects do

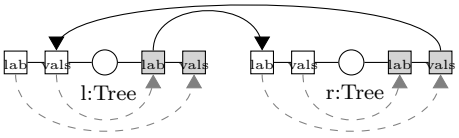

Figure 4.9: The type 3 cycle discovered by Kastens' algorithm for LABEL-VALUE.

not influence feasibility and the procedure we propose will find $e \to f$ at a later stage. Adding fake dependencies solving multiple cycles at once is a nice enhancement. The goal of this section is limit the set of sensible candidates, such that the backtracking procedure's search space is limited. We do not consider this option, as it seems hard to find such dependencies efficiently.

Lemma 4.2. *Any sensible set of fake dependencies will contain only edges that are the reverse of some edge in a cycle in ED_P.*

The LABEL-VALUE example of section 2.1, is shown to be an AOAG/LOAG in Section 3.4.5 by adding a fake dependency $\texttt{Leaf.lhs}.label \to \texttt{Leaf.lhs}.vals$. The type 3 cycle that renders LABEL-VALUE $\notin$ OAG is given in Figure 4.9.

Although fake dependency $\texttt{Leaf.lhs}.label \to \texttt{Leaf.lhs}.vals$ works, it is not a sensible candidate as it is not part of the cycle. However, its clone-pair $\texttt{Bin.r}.label \to \texttt{Bin.r}.vals$ is. In turn, this dependency is not a candidate as it is not a reversed dependency of a dependency in the cycle. Still it works, as it implies $\texttt{Bin.l}.lab \to \texttt{Bin.l}.vals$ through transitivity. It is clear that a fake dependency might work for different reasons. As noted before, we decide to consider only candidates which remedial effects are direct effects instead of side-effects.

Trimming the cycle

So far we know that all candidates will be the reverse of some edge in a cycle in ED_P and that all candidates must not be a part of D_P (subset of ID_P). It follows that, looking at the definition of ED_P (equation 3.13, page 32), the edge must be in the set

$$\begin{aligned}\{\ (X_{p,i} \cdot b \to X_{p,i} \cdot a)\ \mid\ & X_{p,i} = X,\ (X \cdot a \to X \cdot b) \in D_S, \\ & X_{p,i} \cdot a \in cycle(ED_P(p)),\ X_{p,i} \cdot b \in cycle(ED_P(p))\ \}\end{aligned} \tag{4.1}$$

Many edges in D_S originate, through ID_S, from ID_P and Lemma 4.1 says that the reverses of these edges are not candidates. We can therefore only consider intra-visit dependencies.

Lemma 4.3. *Any sensible set of fake dependencies will only contain the reverse of dependencies in the set $(D_S \setminus ID_S)$, i.e. $(IVD \setminus ID_S)$.*

The set $(IVD \setminus ID_S)$ is exactly the set of inter-thread dependencies.

Theorem 4.1. *Any sensible set of fake dependencies will only contain the reverse of* inter-thread dependencies.

Note that all inter-thread dependencies satisfy the normalisation property.

4.3.1 On the correctness of candidates

We argue that if there exists some set of fake dependencies that prove an AG is ordered, then we can prove the AG is ordered using only sensible candidates according to Theorem 4.1. However,

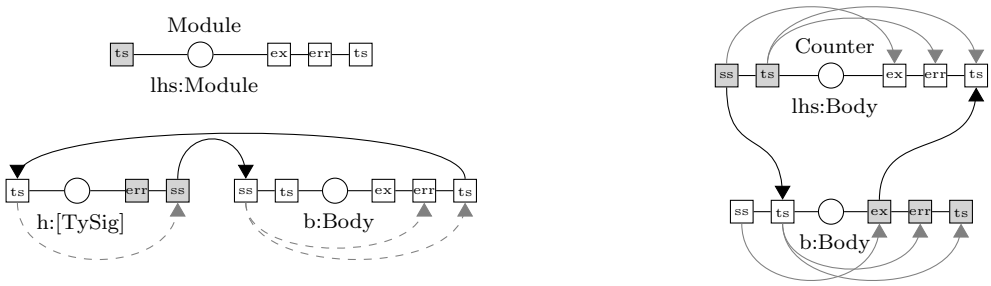

Figure 4.10: Choosing (*Module*.b.ts(syn) → *Module*.b.ss(inh)) as a fake dependency results in a unresolvable cycle in production *Counter*, as it implies both (*Counter*.lhs.ts(inh) → *Counter*.lhs.ex(syn)) and (*Counter*.b.ex(syn) → *Counter*.b.ts(inh)).

linear orderedness is NP-hard while we can compute candidates efficiently. Therefore the set of all candidates must contain 'incorrect choices': it contains dependencies that result in type 2 cycles. Type 1 cycles can not occur as all candidates are normalised, while type 3 cycles will be resolved by yet another fake dependency. Let ID_P be the induced dependency graph before selecting candidates. We examine what kind of dependencies produce cycles in ID'_P, the induced dependency graph we obtain by adding $(a \to b)$ as a fake dependency:

1. An edge $(b \to a) \in ID_P$ causing a cycle in ID'_P, with the new edge $(a \to b)$.
2. An edge $(b' \to a') \in ID_P$ causing a cycle with $(a' \to b') \in ID'_P$, imposed as a induced dependency from the new edge $(a \to b)$ because (a', b') is a clone-pair of (a, b).
3. An edge $(c \to d) \in ID_P$ causing a cycle with $(d \to c) \in ID'_P$ imposed as a induced dependency by the new edge $(a \to b)$ through clone-pairs *and* transitivity.
4. Both $(c \to d)$ and $(d \to c)$ are induced by $(a \to b)$.

In the first three cases, the fake dependency is not a sensible candidate. In the fourth case, there must be an edge from an inherited to a synthesized attribute inducing an edge from a synthesized to an inherited attribute (or vice versa). Such a *switch* can occur between attributes of the same non-terminal. Consider inherited attributes i_1, i_2; synthesized attributes s_1, s_2; and sensible candidate $(X_{p,i} \cdot s_1 \to X_{p,i} \cdot i_1)$. All candidates respect the normalisation property, thus $X_{p,i}$ must be a child node of p $(i > 0)$. An edge $(X_{p,j} \cdot i_2 \to X_{p,j} \cdot s_2)$ can only be implied by our candidate if dependencies $(X_{p,j} \cdot i_2 \longrightarrow X_{p,i} \cdot s_1)$ and $(X_{p,i} \cdot i_1 \longrightarrow X_{p,j} \cdot s_2)$ exist. These dependencies must be induced (hence $i = j$), since $X_{p,i} \cdot s_1$ is an input occurrence while $X_{p,i} \cdot i_1$ is an output occurrences. It is possible that candidate $(X_{p,i} \cdot s_1 \to X_{p,i} \cdot i_1)$ induces a dependency contradicting a dependency induced by $(X_{p,j} \cdot i_2 \to X_{p,j} \cdot s_2)$.

Backtracking in iModule

From the cycle in Figure 4.7 we extract candidates (`b`.ts(syn) → `b`.ss) and (`h`.ss → `h`.ts) that both succeed in altering the calculated interfaces such that iMODULE is recognized as an OAG. We can observe the switch pattern in non-terminal `Body` when we add a new production *Counter* (Figure 4.10) and select candidate (`Body`.ts(syn) → `Body`.ss). It implies both (`lhs`.ts(inh) → `lhs`.ex) and (`b`.ex → `b`.ts(inh)) in *Counter*. By adding the production *Counter* to non-terminal `Body` (figure 4.10) we have created a counter-example to our candidate selection. Graph ID_P

becomes cyclic and backtracking is required for the algorithm to choose ($\texttt{h}.ss \rightarrow \texttt{h}.ts$) from the candidates instead.

The production `Counter` is highly artificial and its addition renders the AG to be ill-typed (not well-typed). Backtracking is required only for these kinds of constructions. We therefore expect that backtracking steps in the AOAG algorithm will be rare for practical AG descriptions.

4.4 Implementation

We have implemented our algorithm using a monad, *AOAG s a*, built on top of monads ST and Error. The main function of the algorithm is show in Figure 4.11.

```
aoag :: Graph s → AOAG s (Maybe (Graph s))
aoag dp = do                                -- given direct dependency graph D_P
  (idp, ids)    ← induced dp                -- induced dependencies, Section 3.3.3
  thread_ifs    ← schedule ids              -- set of interfaces for each thread, Section 3.3.6
  interfaces    ← merge thread_ifs          -- merging interfaces for threads, Section 4.2.1
  ds            ← edges interfaces          -- intra-visit dependencies, Section 3.3.4
  itd           ← inters threads            -- inter-thread dependencies, Section 4.2.2
  mc            ← oagtest dp ds             -- definition 3.7
  case mc of    -- did oagtest report a cycle in ED_P?
    Nothing → return (Just interfaces)      -- recognised as OAG
    Just c  → explore c dp idp threads itd  -- use fake dependencies
```

Figure 4.11: The main function of the AOAG algorithm, performing the steps of Kastens' algorithm and exploring fake dependency candidates if necessary.

Figure 4.12 shows the backtracking function for exploring all fake dependency candidates. The dependency graphs (of type *Graph s*) are STRefs that point to the graph hidden in state, behind the usual graph operations. Since we use pointers we have to make actual copies of graphs ID_P and *threads* to allow backtracking. Function *insertCandidate* inserts a fake dependency a to the induced dependency graph in such a way that all the effects of choosing a as a candidate are reflected in the graph, by adding all induced dependencies imposed by a. It returns the new dependencies between attributes (at non-terminal level) required by this insert. When the candidate leads to a type 2 cycle backtracking is performed.

4.5 Conclusions

4.5.1 Results

Although the algorithm we presented is exponential in theory, it turns out to be efficient for realistic full-sized attribute grammars in practice.

The AOAG algorithm compiles the main AG of the UHC in 35 seconds, automatically selecting 10 fake dependencies, while it needs 25 seconds using 24 fake dependencies originally in the source code. Even though the main AG contains many complicated dependency patterns, none of these patterns contained a counter-example to our fake dependency selection mechanism (like the one in Figure 4.10), so no backtracking is performed. As happens to be the case for all LOAGs we have tested. We therefore expect that backtracking is very rare for practical AGs.

```
  -- edgeOccs generates all occurrences of an edge between attributes
edgeOccs :: Edge    → [Edge]
  -- try candidates one-by-one until the first result
explore    :: [Vertex]     → Graph s → Graph s → Graph s
                           → [Edge] → AOAG s (Maybe (Graph s))
explore c dp idp threads =
    -- candidates should be swapped and part of the cycle
  explore' ∘ filter (∈ [(f, t) | f ← c, t ← c]) ∘ concatMap (edgeOccs ∘ swap)
where explore' :: [Edge] → AOAG s (Maybe (Graph s))
      explore' [] = return Nothing   -- no more candidates
      explore' (a : as) = do
        idpC      ← copy idp
        threadsC ← copy threads
        let backtrack = explore c dp idpC threadsC as
        cOrEs     ← insertCandidate a idp   -- explained above
        case cOrEs of                -- is there a type 2 cycle?
          Left c → backtrack         -- yes
          Right es → do              -- new attribute dependencies
            reschedule threads es    -- recompute the threads, Figure 4.8
            interfaces ← merge threads   -- new interfaces
            ds         ← edges  interfaces
            itd        ← inters threads
            mc         ← oagtest dp ds    -- test new interfaces
            case mc of               -- is there a cycle in ED_P this time?
              Nothing → return (Just interfaces)   -- no, AOAG
              Just c  → do           -- yes, try the new candidates
                mres ← explore c dp idp threads itd
                case mres of     -- did the extra candidates help?
                  Nothing → backtrack            -- no, lets backtrack
                  Just res → return (Just res)   -- yes, AOAG
```

Figure 4.12: A backtracking procedure for finding a set ADS. If a selected candidate renders the AG unorderable, then the next candidate is tried. If no candidates remain the AG was unorderable to begin with.

The AOAG algorithm can be used both as a tool for deciding membership of LOAG, required in an evaluator generator for LOAGs, and as a tool for automatically finding fake dependencies.

4.5.2 Open questions

Adding a fake dependency might lead to a cycle that can only be solved by yet another fake dependency. Influencing the runtime of the algorithm. AGs can be constructed that require an arbitrary number of fake dependencies (by repeating some construction an arbitrary number of times), while a different choice in the first step leads to a schedule directly. It remains to be investigated whether taking this into account can improve efficiency of the algorithm for realistic AGs. Furthermore, some fake dependencies might solve multiple cycles at once. Although we expect that they are neither very common in large AGs nor easy to find, selecting these candidates improves the runtime of the algorithm. In general, we have not experimented with heuristics

to decide which candidate to try first. It is interesting to see the effects such heuristics on the runtime of the algorithm and on the size of the set ADS.

4.5.3 LOAG Approach

The next chapter improves on the AOAG algorithm by defining an algorithm that solves that schedules LOAGs more directly, instead of making and correcting some optimistic choices. The motivation for setting this goal is threefold: we wish to give an algorithm that is not optimistic, we wish to improve on the runtime of the AOAG algorithm and we wish to give an algorithm that is able to take different optimisation criteria into account.

Chapter 5

LOAG Algorithm

In this chapter we give the second algorithm for determining whether an AG is of the class LOAG. The algorithm computes a Boolean formula that, when shown to be *satisfiable*, guarantees a linear order exists. Any SAT-solver can be used to find a satisfying assignment of the variables in the formula. Interfaces are constructed from the assignment. We show how the Boolean formula is constructed, prove that the formula finds valid interfaces and show how the size of the Boolean formula can be reduced to make it easier to generate and solve.

5.1 SAT approach

5.1.1 Problem definition

Multiple characterisations of the class LOAG have been explored in Chapter 3. From the definitions and intuitions given in that chapter we obtain a minimal decision problem for determining whether an AG is linear ordered.

From Chapter 3 we know that it suffices to construct complete, well-defined and compatible interfaces to show a linear order exists for any derivation tree. Complete and well-defined interfaces are easily generated from $ID_S(X)$ (Section 3.3.6). Generating compatible interfaces is the real challenge. A set of intra-visit dependencies has to be taken account for every interfaces. The combination of the different intra-visit dependencies from the different interfaces might result in cycles at production level. If each direct dependency graph remains acyclic with the intra-visit dependencies added to it, the interfaces are compatible.

The AOAG algorithm finds compatible interfaces by making an optimistic choice and correcting this choice when a cycle is discovered. The algorithm presented in this chapter approaches the problem from another angle. It finds a set of intra-visit dependencies first, ensuring that they do not lead to cycles at production level. Complete, well-defined and compatible interfaces can be constructed from a set of the intra-visit dependencies that is total: the interfaces constructed from them impose only intra-visit dependencies that were already in the set. By selecting $(i \to s)$ or $(s \to i)$ for every pair $i \in AI_N(X) \times s \in AS_N(X)$, with $X \in N$, every possible intra-visit dependency is considered. We can thus formulate the problem of linear orderedness as follows:

Definition 5.1 (Boolean linear orderedness) An AG $\langle\langle \Sigma \cup N, P, S\rangle, A, D\rangle$ is of the class LOAG if we can add a dependency $i \to s$ or $s \to i$ for every pair $i \times s$, with $i \in AI_N(X)$ and $s \in AS_N(X)$ for every $X \in N$, such that the direct dependency graphs extended with these

dependencies are cycle free. ■

The following idea for an algorithm follows from the above definition: introduce a variable $x_{i,s}$ for every pair $i \times s$, where $x_{i,s} = T$ implies $i \to s$ and $x_{i,s} = F$ implies $s \to i$. Then impose a constraint on the variables associated with every possible cycle in the dependency graphs, ruling out the combination of variable assignments that lead to this cycle. A valid combination of assignments for all variables is a solution to our problem and we can extract the required interfaces from it. We have thus described linear orderedness as an instance of the problem of Boolean satisfiability.

Definition 5.2 (Boolean Satisfiability Problem) The *Boolean satisfiability problem* (*SAT-problem*) is to determine, for a given Boolean formula, whether it is satisfiable, i.e. there is a variable assignment that satisfies the formula. ■

Definition 5.3 (Boolean Formula) A *Boolean formula* or *Boolean expression* is a set of Boolean variables combined by conjunction ($\wedge$), disjunction ($\vee$) and negation ($\neg$) operators. ■

Definition 5.4 (Conjunctive Normal Form) A Boolean formula is in *Conjunctive Normal Form* (*CNF*) if it is conjunction of clauses, where a *clause* is disjunction of terms. Every term is a variable or a negation of a variable. ■

Every Boolean formula can be transformed into CNF[1].

By restricting the formula of a SAT-problem to CNF we have a very natural representation of the problem. Every clause imposes a constraint and every constraint must hold simultaneously. Furthermore, every constraint is formed by a set of terms of which at least one must hold. Most SAT-solvers[2] expect formulas in CNF and provide an interface for users to add clauses as constraints iteratively (as opposed to forcing the user to construct the Boolean formula in its entirety and providing it to the solver at once).

5.1.2 Ruling out cycles

Adding clauses

We can rule out cycles using clauses as follows. If there is a cycle c, containing the edges $a_1 \to b_1, a_2 \to b_2, \ldots, a_n \to b_n$, by adding a clause $\neg var(a_1, b_1) \vee \neg var(a_2, b_2) \vee \ldots \vee \neg var(a_n, b_n)$ for all edges $a \to b$ for which $var(a, b)$ is defined, given that $var(a, b) = x_{a,b}$ if a is inherited and b is synthesized, $var(a, b) = \neg x_{a,b}$ if a is synthesized and b is inherited or $var(a, b)$ is undefined otherwise. In the next section we examine how all the cycles can be ruled out with this approach.

Problem graph

In our problem definition an assignment to a variable represents the direction of the edge between two vertices. We observe a similarity with the relation between undirected and directed graphs. An undirected graph $G = \langle V, E \rangle$ is represented by a directed graph $G' = \langle V', E' \rangle$, by taking both

[1] Web-application WolframAlpha easily transforms arbitrary Boolean formulas into CNF, among many other things: http://www.wolframalpha.com/

[2] A SAT-solver is software for automatically finding satisfying assignments for SAT-problems, if possible.

$a \to b \in E'$ and $b \to a \in E'$ if a and b are connected in G. We depict the connection between two vertices a and b in undirected graph G by saying $(a \leftrightarrow b) \in E$.

We consider the set of variables as edges in an undirected graph. We refer to this graph as the *problem graph*. The problem graph for production p has a vertex for every input and output occurrence of p. The edges in the problem graph are either edges between siblings or between non-siblings. All possible sibling pairs of input $\times$ output occurrences are added to the graph (they are the variables from definition 5.1). We collect these pairs in the set $S_P(p)$.

$$\begin{aligned} S_P(p) =& \{\ (X \cdot i, X \cdot o)\ \mid\ (X \cdot o) \in AS_N(lhs(p)),\ (X \cdot i) \in AI_N(lhs(p))\ \} \cup \\ & \{\ (X \cdot i, X \cdot o)\ \mid\ (X \cdot o) \in AI_N(X),\ (X \cdot i) \in AS_N(X),\ X \in rhs(p)\ \} \end{aligned} \tag{5.1}$$

Direct dependencies are added to the problem graph by assigning a new variable (adding a new undirected edge) and fixing its value (we know its direction). If a direct dependency relates to a sibling edge already part of the problem graph (it is a dependency between an input and output occurrence of the same field), there is already a variable for that edge and we fix its value.

Definition 5.5 (Problem graph for productions) The *problem graph for production p* is an undirected 2-edge-colored[3] bipartite graph $P_P(p) = \langle U \cup V, \mathcal{D} \cup \mathcal{S} \rangle$, with $U = O_{in}(p)$, $V = O_{out}(p)$ and

$$\begin{aligned} \mathcal{D} = \{\ a \leftrightarrow^{\rightarrow} b\ \mid\ & (a \to b) \in D_P(p), \\ & (a, b) \notin S_P(p),\ (b, a) \notin S_P(p)\ \} \end{aligned} \tag{5.2}$$

$$\begin{aligned} \mathcal{S} =& \{\ a \leftrightarrow b\ \mid\ (a, b) \in S_P(p),\ (a \to b) \notin D_P(p),\ (b \to a) \notin D_P(p)\ \} \cup \\ & \{\ a \leftrightarrow^{\rightarrow} b\ \mid\ (a, b) \in S_P(p),\ (a \to b) \in D_P(p)\ \} \cup \\ & \{\ a \leftrightarrow^{\leftarrow} b\ \mid\ (a, b) \in S_P(p),\ (b \to a) \in D_P(p)\ \} \cup \end{aligned} \tag{5.3}$$

Every edge $(a \leftrightarrow b)$ corresponds to a Boolean variable $x_{a,b}$. There exists only one variable for every edge: either $x_{a,b}$ exists, with $x_{b,a} = \neg x_{a,b}$, or $x_{b,a}$ exists, with $x_{a,b} = \neg x_{b.a}$. Whenever $(a \leftrightarrow^{\rightarrow} b) \in (\mathcal{D} \cup \mathcal{S})$ then $x_{a,b} = T$ (or $x_{b,a} = F$) and whenever $(a \leftrightarrow^{\leftarrow} b) \in (\mathcal{D} \cup \mathcal{S})$ then $x_{a,b} = F$ (or $x_{b,a} = T$). $S_P(p)$ is the set of all sibling pairs of p: Problem graph $P_P(p)$ is bipartite as every edge in $\mathcal{D}$ is from an input to output occurrence (under the assumption that the input AG is normalised) and no edge in $\mathcal{S}$ is from input to input or output to output occurrence. ■

Definition 5.6 (Cycle in undirected graph) A *cycle in an undirected graph* $G = \langle V, E \rangle$ is a set $c = \{v_1, \ldots, v_n\} \subseteq V$ such that for every $1 \leqslant i < n$, $(v_i \leftrightarrow v_{i+1}) \in E$ and $v_1 = v_n$. The edges in c, denoted by $c^{\leftrightarrow}$, are exactly the edges $(v_i \leftrightarrow v_{i+1})$. The length of cycle c is the number of edges in c, i.e. $\mid c^{\leftrightarrow} \mid = n - 1$. ■

Definition 5.7 (Directed cycle in problem graph) A cycle $c = \{v_1, \ldots, v_n\}$ in an (undirected) problem graph $P = \langle V, E \rangle$ is *directed*, if and only if every variable associated with every edge $(a \leftrightarrow b) \in c^{\leftrightarrow}$ has a value assigned to it, i.e. $(a \leftrightarrow^{\rightarrow} b) \in E$ or $(a \leftrightarrow^{\leftarrow} b) \in E$, and the assignments form a directed path $v_1 \leftrightarrow^{\rightarrow} \ldots \leftrightarrow^{\rightarrow} v_n$ or $v_1 \leftrightarrow^{\leftarrow} \ldots \leftrightarrow^{\leftarrow} v_n$. ■

[3] A k-edge-colored graph is a graph with k different types of edges.

Assignment graph

An assignment graph is a directed subgraph of a problem graph, such that it reflects only the assignments made for every variable of the problem graph.

Definition 5.8 (Assignment graph for productions) The *assignment graph for production* p is a directed 2-edge-colored bipartite subgraph, $P_P^{\rightarrow}(p) = \langle U \cup V, \mathcal{D}' \cup \mathcal{S}' \rangle$, of problem graph $P_P(p) = \langle U \cup V, \mathcal{D} \cup \mathcal{S} \rangle$, with

$$\mathcal{D}' = \{\ a \to b \ \mid\ (a \leftrightarrow^{\rightarrow} b) \in \mathcal{D}\ \} \tag{5.4}$$

$$\begin{aligned} \mathcal{S}' = &\{\ a \to b \ \mid\ (a \leftrightarrow^{\rightarrow} b) \in \mathcal{S}\ \} \cup \\ &\{\ b \to a \ \mid\ (a \leftrightarrow^{\leftarrow} b) \in \mathcal{S}\ \} \end{aligned} \tag{5.5}$$

An assignment graph $P_P^{\rightarrow}(p)$ is *complete* if there is an edge $(i \to o)$ or $(o \to i)$, $\forall (i, o) \in S_P(p)$. ■

Whenever we have a complete set of assignments for the variables $\mathcal{S}$ of a problem graph we can extract a complete assignment graph from it. It follows from the definition of assignment graphs, that an assignment graph for production p is a subgraphs of the problem graph of p and is therefore cycle free whenever the problem graph of p contains no directed cycles.

Definition 5.9 (Cycle in directed graph) A *cycle in a directed graph* $G = \langle V, E \rangle$ is a set $c = \{v_1, \ldots, v_n\} \subseteq V$ such that for every $1 \leqslant i < n$, $(v_i \to v_{i+1}) \in E$ and $v_1 = v_n$. The edges in c, denoted by $c^{\rightarrow}$ are exactly the edges $(v_i \to v_{i+1})$. The length of c is the number of edges in c, i.e. $\mid c^{\rightarrow} \mid = n - 1$. ■

We have constructed our problem graph in such a way that there are no possible assignments that lead to cycles of length < 3. There is no variable $x_{a,b}$ with $a = b$ (there are no self edges) and thus no cycles of length 1. Secondly, there are no two variables $x_{a,b}$ and $x_{a',b'}$ that both represent the edge $a \leftrightarrow b$, therefore we can not have cycles of length 2. The absence of such variables is an invariant we must preserve when adding new variables to our problem.

A cycle $v_1 \leftrightarrow v_2 \leftrightarrow \ldots \leftrightarrow v_n \leftrightarrow v_1$ in the problem graph might result in two possible (directed) cycles in the assignment graph, $v_1 \leftarrow v_2 \leftarrow \ldots \leftarrow v_n \leftarrow v_1$ and $v_1 \to v_2 \to \ldots \to v_n \to v_1$. We can rule out both cycles by saying that only assignments in which at least one edge points to the right and at least one edge points to the left are valid. We achieve this by adding the clauses $var(v_1, v_2) \vee \ldots \vee var(v_n, v_1)$ and $\neg var(v_1, v_2) \vee \ldots \vee \neg var(v_n, v_1)$ to our problem, where var is defined as before (Section 5.1.1).

Since we assume every AG is normalised, the edges from D_P in a cycle can not be adjacent (or there is an edge pointing away from an output occurrence or towards an input occurrence). Every pair of direct dependencies in a cycle is therefore either connected through a path that contains an other direct dependency or through a path containing only edges between siblings. Since intra-visit dependencies (and the sibling edges considered in the assignment graphs) are either from input to output occurrence or vice versa, this path of sibling edges must be of odd length (see Figure 5.1).

Lemma 5.1. *No cycle in assignment graph* $P_P^{\rightarrow} = \langle U \cup V, \mathcal{D} \cup \mathcal{S} \rangle$ *has adjacent edges from* $\mathcal{D}$.

Lemma 5.2. *Every path of edges from* $\mathcal{S}$*, in every cycle of assignment graph* $P_P^{\rightarrow} = \langle U \cup V, \mathcal{D} \cup \mathcal{S} \rangle$ *with some edges from* $\mathcal{D}$*, is of odd length.*

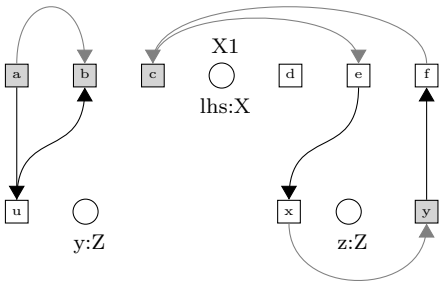

Figure 5.1: In a normalised AG there are no cycles with adjacent direct dependencies and no cycles with even length paths of sibling edges that also contain direct dependencies. The dependencies $u \rightarrow b$ and $e \rightarrow x$ are not allowed.

Enumeration

A naive approach to rule out all possible cycles in the assignment graph would be te enumerate all possible cycles resulting from any possible variable assignment. This goal can be formulated as 'find all minimal cycles of length > 2 in the problem graph'.

Finding the minimal subcomponents (of size > 2) is not easy. No efficient algorithm for it has been found in literature and Tarjan's algorithm which is used to find maximal connected components is not easily adjusted for our purposes. We can, however, construct the problem graph in such a way that the only cycles we have to prevent are cycles of length 3. For this we use the notion of *chordal graphs*.

5.2 Transitivity and chordal graphs

Definition 5.10 (Chordal Graph) An undirected graph is *chordal* if and only if every cycle of length > 3 has a *chord*, an edge that is not part of the cycle but connects two vertices on it. A chordal graph is also called *triangulated* as it consists only of (combinations of) triangles. ■

Definition 5.11 (Triangulated problem graphs) An undirected bipartite 2-edge-colored problem graph $P = \langle U \cup V, \mathcal{D} \cup \mathcal{S} \rangle$ is *triangulated* or *chordalised* by a set of edges $\mathcal{C}$ if $P^{\Delta} = \langle U \cup V, \mathcal{D} \cup \mathcal{S} \cup \mathcal{C} \rangle$ is a chordal graph. There is no guarantee that the edges in $\mathcal{C}$ maintain the bipartite property. Hence, P^{Δ} is an undirected 3-edge-colored graph. ■

We can show that if a problem graph is chordal, adding the clauses for all triangles, such that all assignments leading to directed cycles in that triangle are prevented, suffices to rule out all directed cycles of length > 3.

Theorem 5.1. *Any complete assignment for a chordal problem graph that leads to no directed cycles of length 3, leads to no directed cycles of length* > 3.

Proof. By induction. Base case $n = 3$ follows from assumption "there are no directed cycles of length 3". Step case: *show that if the problem graph contains no directed cycles of length* $\leqslant n$ *then it contains no directed cycles of length* $n + 1$. By contradiction.

Step case, by contradiction. Say that $c = \{v_1, \ldots, v_{n+2}\}$ is an undirected cycle of the triangulated problem graph $P^{\Delta} = \langle V, E \rangle$, with $\mid c^{\leftrightarrow} \mid = n + 1$. Since P^{Δ} is chordal, there must be a

chord $\{v_a \leftrightarrow v_b\}$, with $i \leqslant a < b < n+2$. The chord splits the cycle in two subcycles c_1 and c_2. There are two ways in which c forms a directed cycle: either there is a path $v_a \longleftrightarrow^{\rightarrow} v_b$ through c_1 and a path $v_b \longleftrightarrow^{\rightarrow} v_a$ through c_2 or a path $v_a \longleftrightarrow^{\leftarrow} v_b$ through c_1 and a path $v_b \longleftrightarrow^{\leftarrow} v_a$ through c_2. In the former case $(v_a \leftrightarrow^{\rightarrow} v_b) \in E$ (or there is a directed cycle of length $\leqslant n$ in c_1 or the assignment is incomplete) which leads to a directed cycle of length $\leqslant n$ in c_2. In the latter case $(v_a \leftrightarrow^{\leftarrow} v_b) \in E$ (or there is a directed cycle of length $\leqslant n$ in c_1 or the assignment is incomplete) which leads to a directed cycle of length $\leqslant n$ in c_2.

Both cases contradict the assumption of the induction hypothesis. □

We can view chords as sharing information about the presence of paths through one subcycle with adjacent subcycles. Figure 5.2 shows that all possible assignments ruling out cycles of length 3 rule out cycles of length 4.

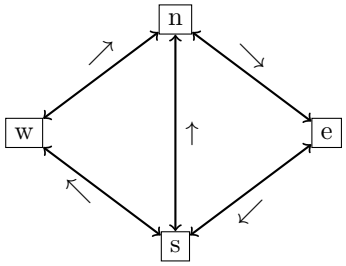

Figure 5.2: A cycle of length 4 is ruled out by ruling out all its subcycles of length 3. When $s \leftrightarrow^{\rightarrow} w$ and $w \leftrightarrow^{\rightarrow} n$ are assigned, $s \leftrightarrow^{\rightarrow} n$ is implied. When $s \leftrightarrow^{\rightarrow} n$ is assigned, $n \leftrightarrow^{\rightarrow} e$ and $e \leftrightarrow^{\rightarrow} s$ can not be assigned simultaneously.

5.2.1 Constructing a chordal graph

A different characterisation of a chordal graph uses the notion of a perfect elimination order:

Chordal Graph Characterisation 1. *An undirected graph G is* chordal *if and only if it has a* perfect elimination order. *A perfect elimination order is an ordering $v_1, \ldots, v_n$ of the vertices of G such that in the graph $G[v_1, \ldots, v_i]$, $\forall(i)$ $(1 \leqslant i \leqslant n)$, the vertex v_i is* simplicial. *A vertex v is called simplicial in a graph G if the neighbourhood of v forms a connected component in G. The graph $G[v_1, \ldots, v_i]$ is the induced subgraph of G containing only the vertices $v_i, \ldots, v_i$ and the edges between these vertices.*

We can use the notion of a perfect elimination order to transform any problem graph into a chordal graph. Consider the following procedure:

1. While the graph still contains vertices:
   (a) Select a vertex v from the problem graph in some arbitrary way.
   (b) For every pair (a, b) of unconnected vertices in the neighbourhood of v:
      i. Add the edge $(a \leftrightarrow b)$ to $\mathcal{C}$.
      ii. Associate a single new variable with the new edge $(a \leftrightarrow b)$.
   (c) Remove v, and all edges that concern v, from the graph.

It is easy to see that the original problem graph $P_P(p)$ is triangulated by $\mathcal{C}$ into chordal graph $P_P^{\Delta}(p)$. The order in which we removed the vertices from the graph follows a perfect elimination order. We say that by fixing the order in which we remove the vertices, we have chosen an elimination order proving that $P_P^{\Delta}(p)$ is chordal. The chosen elimination order greatly influences the numbers of clauses and variables added to our problem. This topic will be discussed in Section 5.3. By adding variables (chords) this way we maintain the invariant that no variable represents a self edge and no two variables represent the same edge.

5.2.2 Enumerating all clauses

It follows from Theorem 5.1 that it suffices to prevent assignments that lead to cycles of length 3 by adding clauses for all triangles in P_P^{Δ}. The procedure for transforming the problem graph into a chordal graph, given in the previous section, encounters all the triangles the triangulated problem graph contains. We can therefore extend the procedure to add all clauses to our SAT-problem right away. We replace step (b) by:

(b) For every pair (a, b), with $a \neq b$ of vertices in the neighbourhood of v:

i. If a and b are unconnected, add $a \leftrightarrow b$ to $\mathcal{C}$ and introduce a new variable $x_{a,b}$.

ii. Consider the variables $x_{v,a}$, $x_{a,b}$ and $x_{b,v}$.

iii. Rule out the two directed cycles possible in undirected cycle $\{a, b, v\}$, by adding the clauses $x_{v,a} \vee x_{a,b} \vee x_{b,v}$ and $\neg x_{v,a} \vee \neg x_{a,b} \vee \neg x_{b,v}$ to the SAT-problem.

A Haskell implementation for adding all required clauses is given in Figure 5.3. Note that some of the edges in a triangle already have a value assigned to it if it represents a direct dependency. We know from Lemma 5.1 that this can hold for a single edge in a triangle only. When such an edge exists, one clause has been satisfied already (the corresponding variable is True) and the other will consist out of two variables instead of three (the corresponding variable can never be True). Since all the clauses added to the problem consist of three variables or less, the problem is a 3-satisfiability instance. 3-satisfiability is one of Karp's 21 NP-complete problems[Kar72].

Definition 5.12 (3-satisfiability) The problem of *3-satisfiability* (*3-SAT*) is a special instance of the Boolean satisfiability problem, requiring the Boolean formula to be in CNF and every clause to be of length at most 3. ■

5.2.3 Constructing problem graphs

The problem graphs are graphs at production level, in which vertices represent attribute occurrences. Every field of a production is an occurrence of a non-terminal for which we have to construct the same interface. The variables can therefore be created at non-terminal level and shared amongst the clone-pairs.

Following the definition of the problem of Boolean linear orderedness we begin by associating a variable $x_{i,s}$ with every edge $i \leftrightarrow s$ for every pair of inherited and synthesized attributes of a non-terminal.

$$S_N(X) = \{\ (X \cdot i, X \cdot s)\ \mid\ (X \cdot i) \in AI_N(X),\ (X \cdot s) \in AS_N(X)\ \} \tag{5.6}$$

When constructing the problem graph for a production p, all the edges between inherited and synthesized occurrences of the fields of p (siblings) will already have a variable associated

```
-- given a sat problem and a problem graph, add all required clauses to
-- the sat problem such that any satisfying assignment will not
-- produce cycles of length > 3, returning the set C
-- required for triangulating the problem graph.
noCycles :: Sat → Graph → { Chord } → IO { Chord }
noCycles sat graph oldchords | empty graph = return oldchords
                             | otherwise   = do
  -- select the next vertex from the elimination order,
  v          ← selectVertex graph
  -- its neighbours
  vns        ← neighbours v graph
  -- and all pairs that v forms a triangle with.
  let pairs = [(a, b) | a ← ns, b ← ns, a ≢ b]
  -- add the required clauses for every triangle, finding necessary chords
  newchords ← sequence [noTriangle sat graph a b v | (a, b) ← pairs]
  -- recursively call noCycles after removing v from the problem graph
  noCycles sat (removeVertex v graph) ((concat newchords) union oldchords)
  where   -- check if there is already an edge a ↔ b
    noTriangle :: Sat → Graph → Vertex → Vertex → Vertex → IO { Chord }
    noTriangle sat graph a b v = do
          if (¬ (edgeExists (a, b) graph))   -- check if (a, b) exists
            then do
              x ← newVar sat (a, b)          -- if not, create a new var
              addEdge (a, b) x graph         -- and add it to the graph
              ruleMeOut                      -- prevent direct cycles
              return [(a, b)]                -- return the new chord
          else ruleMeOut ≫ return []
      where ruleMeOut = ruleOut sat (var (v, a) graph)
                                    (var (a, b) graph)
                                    (var (b, v) graph)
    ruleOut :: Sat → Var → Var → Var → IO ()
    ruleOut sat x y z = do addClause sat [x, y, z]
                           addClause sat [¬ x, ¬ y, ¬ z]
```

Figure 5.3: Haskell function for finding a perfect elimination order, enumerating all triangles.

```
-- Given the current sat problem, a production and a map associating variables
-- to edges between attributes, generate the problem graph for production p
prGraph :: Sat → Pr → [(Edge, Var)] → IO ()
prGraph sat p varMap = do
    graph ← newGraph
    -- add the variables associated with all siblings of this production
    sequence [ addEdge (i, o) xio graph
             | field ← fields p
             , inhs ← AI_N(typeOf field)
             , syns ← AS_N(typeOf field)
             , let   i | field ≡ "lhs" = inhs   -- input occurrences
                       | otherwise     = syns
                     o | field ≡ "lhs" = syns   -- output occurrences
                       | otherwise     = inhs
                     xio = case lookup (i, o) varMap of
                       Nothing → error "sibling edge has no var"
                       Just v  → v
             ]
    -- fix all the variables associated with direct dependencies
    -- or add a variable if it did not yet exist and fix it
    sequence [ do when (x ≡ ⊥) $ do
                    x ← newVar (a, b)
                  assert x graph   -- fix x to a ↔→ b
             | (a → b) ∈ D_P(p)
             , let x = var (a, b) graph
             ]
    chords ← noCycles sat graph ∅
    return ()
```

Figure 5.4: Haskell function that constructs the problem graph for production p and calls noCycles.

with it (through the non-terminal which is the type of the field). We say that these variables are *shared* between fields of the same type. The initialisation of the graph is completed by adding associating a variable $x_{a,b}$ with the edge $a \leftrightarrow b$ for every edge $(a \to b) \in D_P(p)$. The variable will receive assignment True, i.e. $var(a, b) = T$ and $var(b, a) = F$. In Figure 5.4 a Haskell function is given that performs the initialisation steps for a given production and calls *noCycles* from Figure 5.3 to rule out all assignments that lead to directed cycles.

5.3 Reducing the SAT-problem

After collecting the required clauses from the problem graphs, we can rely on an external sat solver to find a satisfying assignment, completing the SAT-based algorithm. However, the following observations allow us to shrink the SAT-problem:

- Many encountered triangles are triangles of siblings. The clauses added for such triangles can be shared between fields of the same type. We can rule out such triangles at non-terminal level first, preventing a duplication of clauses. This allows us to forget about

triangles between two or more siblings at production level.

- Since we assume the input AG is normalised, no direct dependencies are connected and no cycle can have two or more adjacent direct dependencies. This assumption guarantees that no cycles with two or more direct dependencies are encountered.
- Just as variables for inherited-synthesized sibling pairs are shared between fields of the same type, newly added chords (inherited-inherited and synthesized-synthesized pairs) between siblings can be shared.
- Determining the elimination order greatly influences the number of clauses and variables added to the SAT-problem. For example, if the graph is chordal to begin with it has an elimination order that, if we use it, requires the addition of no extra variables in the SAT-problem. Furthermore, we can optimise the elimination order we produce with respect to the sum of the number of triangles encountered for every vertex, reducing the number of clauses added to the SAT-problem.

5.3.1 Non-terminal level

The algorithm sketched above first creates new variables for every inherited-synthesized pair of every non-terminal, then initialises the problem graph for every production and traverses it in a perfect elimination order giving us a set of edges (new variables / chords) that together with the initial edges form a chordal graph. In the meantime clauses are added that prevent any solution to the sat problem in which there is a cycle of length 3.

We extend the algorithm by moving as much work as possible to the non-terminal level. Before creating the problem graphs for all productions a problem graph is constructed and triangulated for every non-terminal. Every triangle encountered at this stage corresponds to at least one triangle at the production level (only for root nodes will it be at most one).

Definition 5.13 (Problem graph for non-terminals) The *problem graph for non-terminal* X is an undirected 2-edge-colored bipartite graph $P_N(X) = \langle U \cup V, \mathcal{D} \cup \mathcal{S} \rangle$, with $U = AI_N(X)$, $V = AS_N(X)$ and

$$\begin{aligned} \mathcal{D} = \{\ X \cdot a \leftrightarrow^{\rightarrow} X \cdot b \ \mid\ & (X \cdot a \to X \cdot b) \in D_N(X), \\ & (X \cdot a, X \cdot b) \notin S_N(X),\ (X \cdot b, X \cdot a) \notin S_N(X)\ \} \end{aligned} \tag{5.7}$$

$$\begin{aligned} \mathcal{S} = & \{\ a \leftrightarrow b \ \mid\ (a,b) \in S_N(X),\ (a \to b) \notin D_N(X),\ (b \to a) \notin D_N(X)\ \} \cup \\ & \{\ a \leftrightarrow^{\rightarrow} b \ \mid\ (a,b) \in S_N(X),\ (a \to b) \in D_N(X)\ \} \cup \\ & \{\ a \leftrightarrow^{\leftarrow} b \ \mid\ (a,b) \in S_N(X),\ (b \to a) \in D_N(X)\ \} \end{aligned} \tag{5.8}$$

Every edge $(a \leftrightarrow b)$ corresponds to a Boolean variable $x_{a,b}$. There exists only one variable for every edge: either $x_{a,b}$ exists, with $x_{b,a} = \neg x_{a,b}$, or $x_{b,a}$ exists, with $x_{a,b} = \neg x_{b.a}$. Whenever $(a \leftrightarrow^{\rightarrow} b) \in \mathcal{D} \cup \mathcal{S}$ then $x_{a,b} = T$ (or $x_{b,a} = F$) and whenever $(a \leftrightarrow^{\leftarrow} b) \in \mathcal{D} \cup \mathcal{S}$ then $x_{a,b} = F$ (or $x_{b,a} = T$). $S_N(X)$ is the set of all sibling pairs of X (equation 5.6). $D_N(X)$ is the set of all direct dependencies for non-terminal X:

$$D_N(X) = \{\ X \cdot a \to X \cdot b \ \mid\ (X_p^i \cdot a \to X_q^j \cdot b) \in D_P(p),\ X_p^i = X = X_q^j\ \} \tag{5.9}$$

■

Note that the definition uses $D_N(X)$ instead of $ID_S(X)$. Propagating dependency paths through non-terminals is left to the SAT-solver, instead of by a separate graph algorithm.

Definition 5.14 (Assignment graph for non-terminals) The *assignment graph for non-terminal* X is a directed 2-edge-coloured bipartite subgraph, $P_N^{\rightarrow}(X) = \langle U \cup V, \mathcal{D}' \cup \mathcal{S}' \rangle$, of problem graph $P_N(X) = \langle U \cup V, \mathcal{D} \cup \mathcal{S} \rangle$, with $U = AI_N(X)$, $V = AS_N(X)$ and

$$\mathcal{D}' = \{\ a \to b \mid (a \leftrightarrow^{\rightarrow} b) \in \mathcal{D}\ \} \tag{5.10}$$

$$\begin{aligned} \mathcal{S}' = &\{\ a \to a \mid (a \leftrightarrow^{\rightarrow} b) \in \mathcal{S}\ \} \cup \\ &\{\ b \to a \mid (a \leftrightarrow^{\leftarrow} b) \in \mathcal{S}\ \} \cup \end{aligned} \tag{5.11}$$

An assignment graph $P_N^{\rightarrow}(X)$ is *complete* if there is an edge $(i \to s)$ or $(s \to i)$, $\forall (i, s) \in S_N(X)$. ■

Some cycles in an assignment graphs for productions are cycles that contain only vertices of a single field. When the variables associated with the edges of field f are shared amongst fields of the same type X, ruling all out cycles for X separately, rules out all the cycles spanning attribute occurrences of all fields of type X. We then no longer have to consider triangles with adjacent sibling dependencies at production level. To prove this we need additional lemmas. Figure 5.5 shows how the problem graphs for a non-terminal is initialised, such that the variable association can be shared with fields of that type.

Lemma 5.3. *If a complete assignment graph for non-terminal X is cycle free, then every path of length 3 is closed under transitivity, i.e. every path $i_1 \to s_1 \to i_2 \to s_2 \Rightarrow i_1 \to s_2$.*

Proof. From definition 5.13, together with the assumption that the direct dependencies are normalised, we know that all edges in an assignment graph for non-terminals are between inherited and synthesized attributes or vice versa (not between inherited-inherited and synthesized-synthesized pairs). Assume a complete assignment graph $P_N^{\rightarrow}(X) = \langle V, E \rangle$ and the arbitrary path $i_1 \to s_1 \to i_2 \to s_2$ as part of the graph. Every i must be inherited (or synthesized) while very s must be synthesized (or inherited). Since every inherited-synthesized pair is part of complete assignment graph $P_N^{\rightarrow}(X)$, so must either $(i_1 \to s_2)$ or $(s_2 \to i_1)$. The edge $(s_2 \to i_1)$ contradicts our assumption that the assignment graph is cycle free. We conclude that $(i_1 \to s_2) \in E$. □

Lemma 5.4. *If in an assignment graph for non-terminal X every path of length 3 is closed under transitivity, then every path of odd length is closed under transitivity.*

Proof. By induction with base case $n = 3$ (using Lemma 5.3). Step case: *Show that if every path of odd length $\leqslant n$ is closed under transitivity in an assignment graph for X then so is every path of odd length $n + 2$.* Consider arbitrary path $i_1 \to s_1 \to \ldots \to i_{k-1} \to s_{k-1} \to i_k \to s_k$ of length $n + 2$. The path $i_1 \to s_1 \to \ldots \to i_{k-1} \to s_{k-1}$ is of odd length n, thus closed under transitivity following the induction hypothesis, showing the existence of path $i_1 \to s_{k-1} \to i_k \to s_k$, which is a path of length 3 and also closed under transitivity (Lemma 5.3). We conclude edge $i_1 \to s_k$ exists. □

Definition 5.15 (Alternating cycle for assignment graphs) Cycle c in assignment graph $P_P^{\rightarrow}(p) = \langle U \cup V, \mathcal{D} \cup \mathcal{S} \rangle$ is *alternating*, if no two edges from $\mathcal{D}$ and no two edges from $\mathcal{S}$ are adjacent in c. ■

Definition 5.16 (Alternating cycle for chordal problem graphs) Cycle c in chordal problem graph $P_P^{\Delta}(p) = \langle U \cup V, \mathcal{D} \cup \mathcal{S} \cup \mathcal{C} \rangle$ is *alternating*, if no two edges from $\mathcal{D}$ and no two edges from $\mathcal{S}$ are adjacent in c. Note that an alternating cycle can contain any number of adjacent edges from $\mathcal{C}$. ■

```
ntGraph :: Sat → Nt → [(Edge, Var)] → IO [(Edge, Var)]
ntGraph sat x varMap = do
    graph ← newGraph
    -- create variables for all direct dependencies
    newVars ← sequence [ do xis ← newVar (i, s)
                            addEdge (i, s) xis graph
                            return (i, s)
                       | (a → b) ∈ D_N (x)
                       , let i | inherited (a)  = a
                               | otherwise      = b
                             s | synthesized (b) = b
                               | otherwise      = a
                       ]
    let varMap = varMap ++ newVars
    -- add the variables/edges associated with all inh/syn pairs of attributes
    newVars ← sequence [ case lookup (i, s) newVar of
                            Nothing → do xis ← newVar (i, s)
                                         addEdge (i, s) xis graph
                                         return [(i, s)]
                            Just xis → return []
                       | i ← AI_N(x)
                       , s ← AS_N(x)
                       ]
    let varMap = varMap ++ (concat newVars)
    chords ← noCycles sat graph ∅
    return varMap
```

Figure 5.5: Haskell function that constructs the problem graph for non-terminal x and calls noCycles.

Lemma 5.5. *If there is no alternating cycle in $P_P^{\rightarrow} = \langle U \cup V, \mathcal{D} \cup \mathcal{S} \rangle$ and there are no cycles spanning only edges from $\mathcal{S}$, then there are no cycles in $P_P^{\rightarrow}(p)$.*

Proof. By contradiction. Consider a non-alternating cycle c in $P_P^{\rightarrow}(p) = \langle U \cup V, \mathcal{D} \cup \mathcal{S} \rangle$. Cycle c must include some edges from $\mathcal{D}$ according to the assumptions. According to Lemma 5.2, all paths spanning only edges from $\mathcal{S}$ in c are of odd length > 1. Under the assumption that there are no cycles spanning edges from $\mathcal{S}$, all such paths are closed under transitivity, according to Lemmas 5.3 and 5.4. Hence, for every non-alternating cycle c there are some alternating cycles, contradicting the assumptions. □

Theorem 5.2. *If there are no directed alternating cycles of length 3 in a chordal problem graph $P_P^{\Delta} = \langle U \cup V, \mathcal{D} \cup \mathcal{S} \cup \mathcal{C} \rangle$ and every odd path of edges from $\mathcal{S}$ is closed under transitivity, then there are no alternating cycles in any of its assignment subgraphs.*

Proof. By induction.

Base case, $n = 4$ (smallest possible alternating cycle in assignment graph). By contradiction. Consider assignment graph $P_P^{\rightarrow}$ that is a subgraph of chordal problem graph $P_P^{\Delta} = \langle U \cup V, \mathcal{D} \cup \mathcal{S} \cup \mathcal{C} \rangle$. Any alternating cycle $c = v_1 \rightarrow v_2 \rightarrow v_3 \rightarrow v_4 \rightarrow v_1$ in $P_P^{\rightarrow}$ corresponds to an alternating directed cycle $c' = v_1 \leftrightarrow^{\rightarrow} v_2 \leftrightarrow^{\rightarrow} v_3 \leftrightarrow^{\rightarrow} v_4 \leftrightarrow^{\rightarrow} v_1$ in P_P^{Δ}. Since P_P^{Δ} is chordal, c' is split into two triangles t_1 and t_2 by some edge $(v_3 \leftrightarrow v_1)$ (or $(v_2 \leftrightarrow v_4)$). $(v_3 \leftrightarrow v_1) \in \mathcal{D}$ contradicts the normalisation assumption, $(v_3 \leftrightarrow v_1) \in \mathcal{S}$ implies $(v_1 \leftrightarrow^{\rightarrow} v_2) \in \mathcal{S}$ and $(v_2 \leftrightarrow^{\rightarrow} v_3) \in \mathcal{S}$, which contradicts the assumption that c' is alternating. Therefore $(v_3 \leftrightarrow v_1) \in \mathcal{C}$, implying that t_1 and t_2 are alternating. The assumption that there are no directed alternating cycles of length 3 implies that $(v_3 \leftrightarrow^{\leftarrow} v_1)$ or there is a directed cycle $v_1 \leftrightarrow^{\rightarrow} v_2 \leftrightarrow^{\rightarrow} v_3 \leftrightarrow^{\rightarrow} v_1$ in t_1 (or t_2). However, $(v_3 \leftrightarrow^{\leftarrow} v_1)$ implies an directed alternating cycle of length 3 in t_2 (or t_1).

Step case. By contradiction. Consider assignment graph $P_P^{\rightarrow}$ that is a subgraph of chordal problem graph $P_P^{\Delta} = \langle U \cup V, \mathcal{D} \cup \mathcal{S} \cup \mathcal{C} \rangle$. Any alternating cycle $c = v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_n \rightarrow v_1$ in $P_P^{\rightarrow}$ corresponds to an directed alternating cycle $c' = v_1 \leftrightarrow^{\rightarrow} v_2 \leftrightarrow^{\rightarrow} \ldots \leftrightarrow^{\rightarrow} v_n \leftrightarrow^{\rightarrow} v_1$ in P_P^{Δ}. Since P_P^{Δ} is chordal, c' is split into two subcycles by some edge $(v_a \leftrightarrow v_b)$. $(v_a \leftrightarrow v_b) \in \mathcal{D}$ contradicts the normalisation assumption, $(v_a \leftrightarrow v_b) \in \mathcal{S}$ implies there exists an alternating cycle of length $\leqslant n$ (as all odd paths of edges from $\mathcal{S}$ are closed under transitivity), contradicting the assumption of the induction hypothesis. Therefore $(v_a \leftrightarrow v_b) \in \mathcal{C}$, splitting c' in two alternating subcycles c'_1 and c'_2. It is implied that $(v_a \leftrightarrow^{\leftarrow} v_b)$, or there is a directed alternating cycle $v_a \leftrightarrow^{\rightarrow} \ldots \leftrightarrow^{\rightarrow} v_b \leftrightarrow^{\rightarrow} v_a$ of length $\leqslant n$ in c'_1 (or c'_2). However, $(v_a \leftrightarrow^{\leftarrow} v_b)$ implies an directed alternating cycle of length of length $\leqslant n$ in c'_2 (or c'_1). □

Ruling out all cycles at non-terminal level and making sure that all variables are properly shared amongst fields of the same type, we know that, using lemmas 5.3 and 5.4, all odd paths spanning edges in $\mathcal{S}$, in every assignment graph $P_P^{\rightarrow}(p) = \langle U \cup V, \mathcal{D} \cup \mathcal{S} \rangle$, are closed under transitivity. According to Lemma 5.5 we only have to worry about alternating cycles in $P_P^{\rightarrow}(p)$. Applying Theorem 5.2 guarantees that all alternating cycles are ruled out, by ruling out all directed alternating cycles of length 3 in the problem graph $P_P^{\Delta}(p)$ (of which $P_P^{\rightarrow}$ is a subgraph). Is is therefore safe to change the algorithm such that *pairs*, calculated in Code Fragment (5.3), computes only pairs from which alternating triangles are constructed. All alternating triangles, concerning vertex v, are found by considering:

- A pair of edges $(v \leftrightarrow d, v \leftrightarrow s)$, with $(v \leftrightarrow d) \in \mathcal{D}$ and $(v \leftrightarrow s) \in \mathcal{S}$.
- A pair of edges $(v \leftrightarrow d, v \leftrightarrow c)$, with $(v \leftrightarrow d) \in \mathcal{D}$ and $(v \leftrightarrow c) \in \mathcal{C}$.
- A pair of edges $(v \leftrightarrow s, v \leftrightarrow c)$, with $(v \leftrightarrow s) \in \mathcal{S}$ and $(v \leftrightarrow c) \in \mathcal{C}$.
- A pair of chords $(v \leftrightarrow c_1, v \leftrightarrow c_2)$, with $(v \leftrightarrow c_1) \in \mathcal{C}$ and $(v \leftrightarrow c_2) \in \mathcal{C}$.

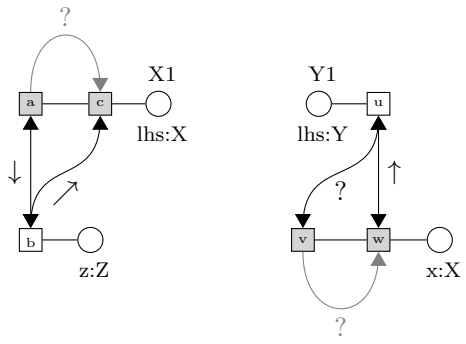

Figure 5.6: When a inherited-inherited chord is shared, a SAT-solver might need less work.

Sharing chords

We can further reduce the number of variables added to our problem by sharing all chords that have been added at non-terminal level. Without this step, a chord between siblings attributes (required to complete a triangle with a direct dependency and another chord) might lead to multiple variables. If the same chord (though at a different field) is added a second time (making a new variable), both variables might be assigned a different value. It will then take time, requiring inspections of the surrounding variables, to discover that the two variables indeed need to have the same value (to produce the same interface).

Figure 5.6 displays this problem. In this figure we see two triangles, each from a different production rule. The gray arrow indicates a chord between siblings of the same kind (both inherited). Consider the four clauses we add for these two triangles: $var(a,c) \vee var(c,b) \vee var(b,a)$, $\neg var(a,c) \vee \neg var(c,b) \vee \neg var(b,a)$, $var(v,w) \vee var(w,u) \vee var(u,v)$ and $\neg var(v,w) \vee \neg var(w,u) \vee \neg var(u,v)$. From the assignments (labels of the arrows) we know that $var(b,a) = F$ and $var(c,b) = F$ and thus that $var(a,c) = T$. If the variables of the two gray chords are shared, i.e. $var(a,c) = var(v,w)$, it is implied that $var(v,w) = T$ and thus that $var(u,v) = F$, as $var(w,u) = T$ is already assigned. Without sharing the variables we can not conclude this yet. In order to properly share the variables the procedure of Figure 5.3 has to be adjusted, such that every new chord between siblings is added to $varMap$.

5.3.2 Reducing the number of encountered triangles

The code fragment for function *noCycles* in Figure 5.3 shows a call to function *selectVertex* unaccompanied by explanation. In this section we show that the decision how to select the next vertex has a large impact on the number of clauses added to the SAT-problem.

Given a partially triangulated problem graph $P = \langle U \cup V, \mathcal{D} \cup \mathcal{S} \cup \mathcal{C} \rangle$ we can select a vertex v based on v's neighbourhood and whether edges connected to v in this neighbourhood are edges within $\mathcal{D}$, $\mathcal{S}$ or $\mathcal{C}$. The goal is to give a heuristic, for selecting the next vertex, such that in the elimination order produced by the heuristic, the numbers of triangles encountered is minimal. Knowing that only alternating triangles are considered, we expect that vertices with less edges from $\mathcal{C}$ in their neighbourhood lead to a smaller of number of triangles.

We investigated this claim by experimenting with the six different orders in which edges from three sets are combined. The results are shown in the first Table of Figure 5.7. In the row with order $(\mathcal{D}, \mathcal{S}, \mathcal{C})$ selects the vertex with the smallest number of edges in $\mathcal{D}$, when equal $\mathcal{S}$ is used for comparison and finally $\mathcal{C}$. In the second table different experiments are shown in which sets

Order	#Clauses	#Vars	Ratio
$(\mid \mathcal{D} \mid, \mid \mathcal{S} \mid, \mid \mathcal{C} \mid)$	21.307.812	374.792	57.85
$(\mid \mathcal{D} \mid, \mid \mathcal{C} \mid, \mid \mathcal{S} \mid)$	8.301.557	220.690	37.62
$(\mid \mathcal{S} \mid, \mid \mathcal{D} \mid, \mid \mathcal{C} \mid)$	12.477.519	287.151	43.45
$(\mid \mathcal{S} \mid, \mid \mathcal{C} \mid, \mid \mathcal{D} \mid)$	8.910.379	241.853	36.84
$(\mid \mathcal{C} \mid, \mid \mathcal{D} \mid, \mid \mathcal{S} \mid)$	3.004.705	137.277	21.89
$(\mid \mathcal{C} \mid, \mid \mathcal{S} \mid, \mid \mathcal{D} \mid)$	3.359.910	156.795	21.43
$(\mid \mathcal{D} \mid + \mid \mathcal{S} \mid, \mid \mathcal{C} \mid)$	12.424.635	386.323	32.16
$(\mid \mathcal{D} \mid, \mid \mathcal{S} \mid + \mid \mathcal{C} \mid)$	8.244.600	219.869	37.50
$(\mid \mathcal{D} \mid + \mid \mathcal{C} \mid, \mid \mathcal{S} \mid)$	2.930.922	135.654	21.61
$(\mid \mathcal{S} \mid, \mid \mathcal{D} \mid + \mid \mathcal{C} \mid)$	8.574.307	236.348	36.28
$(\mid \mathcal{S} \mid + \mid \mathcal{C} \mid, \mid \mathcal{D} \mid)$	3.480.866	157.089	22.16
$(\mid \mathcal{C} \mid, \mid \mathcal{D} \mid + \mid \mathcal{S} \mid)$	3.392.930	157.568	21.53
$(\mid \mathcal{C} \mid + \mid \mathcal{D} \mid + \mid \mathcal{S} \mid)$	3.424.001	148.724	23.02
$(3* \mid \mathcal{S} \mid *(\mid \mathcal{D} \mid + \mid \mathcal{C} \mid) + (\mid \mathcal{D} \mid * \mid \mathcal{C} \mid)^2)$	2.679.772	127.768	20.97

Figure 5.7: Table showing the number of clauses and variables required for solving the main AG of the UHC, selecting the next vertex in the elimination order based on different ways to compare neighbourhoods. All other minimisations of this section are active.

are considered simultaneously. The results tell us that selecting neighbourhoods with a smallest $\mid \mathcal{C} \mid$ is beneficial, especially when considered simultaneously with $\mathcal{D}$ (see row $(\mid \mathcal{D} \mid + \mid \mathcal{C} \mid, \mathcal{S})$).

If we consider how the number of edges in each of the three sets progresses as graph P grows, we observe that at the start of the triangulation process there are no chords ($\mathcal{C}$ is empty). Since set $\mathcal{S}$ contains all combinations of attributes of a field, we expect that $\mathcal{S}$ is much bigger then $\mathcal{D}$. All the edges added to the graph are added to $\mathcal{C}$. We identify two phases: the starting phase, with a none to a few chords and the end phase with a lot of chords. For comparison, there are 19.244 variables between input-output pairs and 9.766 direct dependencies.

During the starting phase the smallest number of alternating triangles is encountered for vertices with a small $\mid \mathcal{D} \mid \times \mid \mathcal{S} \mid$. While during the end phase, a large number of alternating triangles is encountered for neighbourhoods with large $\mid \mathcal{C} \mid$.

The two phases have conflicting targets, we therefore need to find a linear combination with two terms. One should dominate the start phase while the other dominates the end phase.

It is therefore that we created expression $(a* \mid \mathcal{S} \mid *(\mid \mathcal{D} \mid + \mid \mathcal{C} \mid) + (\mid \mathcal{D} \mid * \mid \mathcal{C} \mid)^2)$ and experimented with different values of a. Setting $a = 3$ appears most efficient.

This analysis is hardly complete and we expect better results are still possible.

5.4 Conclusions

We have reformulated the problem of determining whether an AG is LOAG as a minimal decision problem. The resulting formulation corresponds to a satisfiability problem and can be solved by a SAT-solver. A Boolean formula is generated for an input AG and a satisfying assignment to the variables in this formula proves the AG is an LOAG. The Boolean formula disallows all cycles that incompatible interfaces might induce. Complete, well-defined and compatible interfaces can therefore be constructed from the satisfying assignment.

For large AGs many potential cycles must be ruled out. This number is reduced using the notion of chordal graphs. We have shown that it is enough to prevent all triangles (cycles of length 3), after triangulating the dependency graphs into chordal graphs. For every triangle two clauses are added to the SAT-problem.

To reduce the number of clauses, variables of the formula can be shared for fields of the same type. Additionally, we have shown that it suffices to rule out all triangles with at most one direct dependency and at most one dependency between siblings, after all non-terminal graphs have been closed under transitivity.

The number of triangles is greatly influenced by the heuristic used to transform the graphs of the AG into chordal graphs. We have discussed multiple heuristics and compared the resulting number of clauses of the. Figure 5.8 shows the number of variables and clauses for the main AG description and ToGrin AG description of the UHC using different methods for reducing the numbers of variables and clauses.

Clause reduction (main AG)	#vars	#clauses	compiletime
Chordal graphs (Theorem 5.1)	474.727	20.140.906	26s
Sharing variables (Section 5.3.1)	175.557	11.383.773	24s
Alternating triangles (Theorem 5.2)	148.724	3.424.001	12s
Elimination order (Section 5.3.2)	127.768	2.679.772	10s
Clause reduction (ToGrin)	#vars	#clauses	compiletime
Chordal graphs (Theorem 5.1)	24.531	291.993	0.7s
Sharing variables (Section 5.3.1)	6.271	207.999	0.7s
Alternating triangles (Theorem 5.2)	5.115	46.035	0.6s
Elimination order (Section 5.3.2)	4361	34.219	0.6s

Figure 5.8: Number of variables and clauses required for scheduling the main AG and ToGrin AG from the UHC using different ways to reduce the numbers.

In the first step only problem graphs for productions are triangulated. The heuristic for choosing the next vertex in the elimination is order is fixed to $(\mid \mathcal{S} \mid + \mid \mathcal{C} \mid + \mid \mathcal{D} \mid)$. Since all edges have the same color this effectively the same as looking at the length of the neighbourhood.

In the second problem graphs for non-terminals are triangulated first and variables are shared amongst clone-pairs. Productions without children can now be ignored.

In the third step colors are added to the edges of the problem graph and only clauses are added for alternating triangles.

The final steps reduces the number of variables and clauses by improving the heuristic for selecting the next vertex in the elimination order. It is changed from $(\mid \mathcal{S} \mid + \mid \mathcal{C} \mid + \mid \mathcal{D} \mid)$ to $(3* \mid \mathcal{S} \mid *(\mid \mathcal{D} \mid + \mid \mathcal{C} \mid) + (\mid \mathcal{D} \mid * \mid \mathcal{C} \mid)^2)$. The experiments show that the reduction of variables and clauses cause a great decrease in runtime. The decrease in runtime is dominated by the time required to generate the SAT-problem instead of the time required to find a satisfying assignment. This observation implies that the runtime of the algorithm is not dominated by the external SAT-solver used to find the satisfying assignment (MiniSat).

The next chapter discusses how evaluators are generated from visit-sequences and how visit-sequences are generated from complete and compatible interfaces. Additionally, The chapter discusses how the efficiency of evaluators might be compared and a number of criteria are proposed for comparing schedules. Finally, ideas about optimising schedules taking these criteria into account are discussed.

5.4.1 Open questions

Testing preconditions

As will be discussed further in Section 8.3.1, we have been unable to test how the algorithm reacts to AGs within superclasses of LOAG. The algorithm will report that no static evaluation

order can be found for these AGs, however it is unclear how fast a SAT-solver can determine this.

Weakly chordal bipartite graphs

The number of clauses might be reduced further by using the knowledge that the program and assignment graphs are bipartite. So far only the fact that the graphs are 2-edge-colored has been used. We ran multiple experiments and proved additional lemmas using the notion of *weakly chordal* graphs. Weakly chordal graphs are defined using a different kind of elimination order that is based on edges as opposed to vertices. No decrease in the amount of clauses has been achieved. Perhaps the heuristic described in Section 5.3.2 needs a radical change as well.

Chapter 6

Generating Optimised Evaluators

For any input AG there might be zero, one or more solutions to the LOAG scheduling problem. So far we have only been interested in determining whether there are *some* solutions, not in finding specific solutions.

In this section we take a closer look at the code-generation process and present two different models for code-generation. From the models we extract certain optimisation criteria and argue about the feasibility of using our algorithms to take these criteria into account. Experiments with different implementations of some optimisations have shown promising results. We expect that with the LOAG algorithm we have given an algorithm that can be extended to take all kinds of user-defined optimisations into account. Unfortunately, due to time constraints, not all proposed optimisation criteria have been implemented as part of the LOAG algorithm. This task has been left as future work.

6.1 Models for code-generation

The efficiency of generated code relies for the greater part on the execution model of the machine on which it is supposed to run and the nature of the instructions that operate the machine. Therefore it is generally impossible to optimise the generate code, without determining and fixing the execution model. We will not go to these lengths, although the work presented in this thesis might form a starting point for such a thorough investigation.

An AG compiler can generate multiple types of evaluators. For example, it can generate an evaluator that parses syntax and performs attribute evaluation simultaneously, in a process known as *parse-time attribution*. Alternatively, a compiler can generate an incremental evaluator of which the input is a previously decorated parse tree together with a new parse tree. An incremental evaluator tries to redecorate as little as possible based on the difference between the two trees.

Instead of considering a full execution model (and a programming language for it), we compare two methods for generating evaluators using visit-sequences and take visit-sequences as the object to optimise.

6.1.1 Procedural evaluator

From a given set of visit-sequences, executable code can be generated in multiple ways. We discuss two models: a procedural model that is typically used in an imperative context and a

```
-- execute all steps in the current visit-sequence
-- found in the lookup table vss
DECORATE (K : Node, visitnr : Int) =
  -- find the visitnr-th visit-sequence of the production that constructed K
  foreach (step : VisitStep in vss[K.prod, visitnr])
    case step of
      eval (a)    → compute (K, a)    -- evaluate attribute a of K and store its result
      visit (c, i) → DECORATE (c, i)  -- visit c for the i-th time
```

Figure 6.1: An implementation of decoration in the procedural model.

functional model from which pure functions[1] can be generated. The former relies on mutable state to store attributes. The latter is beneficial in an incremental evaluator as pure functions can be memoised. The procedural evaluator requires the generation of three bits of code:

- A table in which the evaluator can look up the steps of every visit-sequence of every production.
- A procedure that performs the evaluation of a certain instance of every attribute occurrence of every production. It stores the result in memory relative to the node of the parse tree to which the attribute instance belongs. The value is stored such that the value of the attribute instance can be accessed both when the node acts as a parent of a production as well as when it acts as a child.
- A procedure that performs the actual decoration of a given parse tree. It either makes a recursive call when a visit-instruction is next in the current visit-sequence or it calls one of the aforementioned evaluation procedures in case of an eval-instruction. An example implementation of such a procedure is given in Figure 6.1.

One of the problems of this model is its greedy storage of attribute values. Attribute instances that are no longer required for decorating other parts of the parse-tree should be removed from memory to save space. This problem is addressed by Kastens[Kas91].

6.1.2 Pure evaluator

A pure evaluator can not rely on a globally mutable state. The value of an attribute instance that is required for the evaluation of some other instance needs to be passed from visit to visit explicitly. As such, the problem is not having too much information available, but too little. The pure functions that are generated in this model perform a single visit to a certain node. We therefore assume that all attribute instances evaluated in a certain visit are available within the function generated for that visit after the they have been evaluated. The problem lies with instances evaluated within visit i and required in a visit $j > i$[Sar99, Pen94].

Within the pure evaluator model, the following bits of code are generated:

- A visit function for every visit-sequence of every production of every non-terminal. It executes all of the instructions in the visit-sequence. As arguments, the visit functions receive the values of the inherited attribute instances of the visit and the values of instances,

[1] A pure function is a function that is influenced only by the values of its parameters and other than producing its return value it has no other (side)effects.

evaluated at other visit functions, it requires. It returns the values of the synthesized attributes of the visit and the values of attributes that are required as arguments of other visit-functions. To determine which additional values have to be received and returned *inter-sequence dependencies* (discussed more formally in Section 6.3.2) are calculated.

- A set of distributing functions, one for every non-terminal, that glue visit functions together by distributing the right attribute instances between them. The distributing function for a non-terminal X, applied to a node K, calls a different visit function based on the production of X that was used to construct K and the number of visits already performed to K. This visit function, in turn, applies the right distributing function whenever a child visit is required. As such, the repeated interaction between visit and distributing functions form a traversal through the parse tree.

6.2 Generating visit-sequences

In the previous section we have discussed generating code from visit-sequences. In this section we show how to generate visit-sequences from a schedule, completing the pipeline from input AG to an evaluator. The evaluator performs evaluation of attribute instances in a linear order (see Section 3.3.1 explains how visit-sequences imply such a linear order).

6.2.1 Scheduling graph

We show in this section that it suffices for any LOAG scheduling algorithm to produce complete, well-defined and compatible interfaces for all non-terminals, by generating visit-sequences from them. The interfaces are combined with the direct dependencies to form a graph, that we refer to as TD_P following Kastens[Kas80]. We informally refer (and referred) to this graph as a *schedule*. Using the intra-visit dependencies that are imposed by the interfaces (Section 3.3.4) and the direct dependency graph D_P, TD_P is defined as follows:

$$\begin{aligned} TD_P(p) = \{\ (X_p^i \cdot a \to X_p^j \cdot b) \ \mid\ & (X_p^i \cdot a \to X_p^j \cdot b) \in D_P(p)\ \} \cup \\ \{\ (X_p^i \cdot a \to X_p^i \cdot b) \ \mid\ & (X \cdot a \to X \cdot b) \in IVD(X), \\ & X_p^i \in (rhs(p) \cup \{lhs(p)\},\ X = X_p^i\ \} \end{aligned} \tag{6.1}$$

6.2.2 Computational paths

From $TD_P(p)$ it is possible to generate visit-sequences which represent linear orders on the attributes occurrence of every production. As the interfaces are well-defined the visit-sequences respect the direct dependencies.

We assume that no direct inspection of a parse-tree is possible: from outside the parse-tree only the root node is available and within the parse-tree only fields of the production, used to construct the current node, are available. Observe that with this assumption not every attribute occurrence of p requires its instances to be evaluated and can therefore be removed from $TD_P(p)$. Whenever a node x executes a visit to one of its children y, the only observable effect within the scope of x is that some synthesized attribute instances of y are now available. A similar fact holds when x is the root of the parse tree: inside the scope from which the first call to the evaluator is made, the only change observable in that scope is that the synthesized instances of root node x are now available. We conclude that for every production p, only the instances of the synthesized occurrences of the parent of the p, together with all the occurrences on which they depend, are to be evaluated.

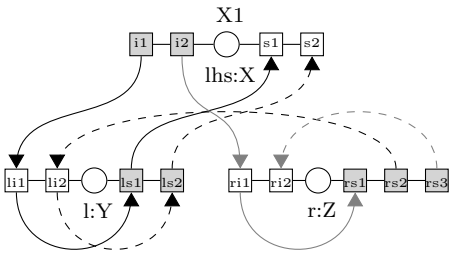

Figure 6.2: The four different computational paths shown in the graph TD_P of a single production.

Every parent node in TD_P can be seen to represent a set of visit functions. There is a function within this set for every visit to the parent: given a subset of its inherited occurrences it produces a subset of its synthesized occurrences. Every path in $TD_P(p)$ represents a computation within a visit function of the parent of p. The paths that do not start at a inherited occurrence of a parent represent a constant computation. Paths that do not end in a synthesized occurrence of a parent do not contribute to the result of the visit function. Following Pennings[Pen94], we distinguish four types of computational paths within $TD_P(p)$:

- *Diverging* computations are represented by paths that start at an inherited occurrence of the parent of p ending at an occurrence that is not a synthesized occurrence of the parent of p (gray in Figure 6.2).
- *Converging* computations are represented by paths that start at an occurrence that is not an inherited occurrence of the parent of p ending at an synthesized occurrence of the parent of p (black and dashed in Figure 6.2).
- *Unconnected* computations start and end at occurrences of children of p (gray and dashed in Figure 6.2).
- *Chained* computations start at an inherited occurrence of the parent of p ending at a synthesized occurrence of the parent of p (black in Figure 6.2).

Paths from synthesized occurrences of the parent of p to inherited occurrences of the parent of p are missing from the above separation as they do not represent computations. Instead, they determine how many visit functions p has and which occurrences belong to which visit function.

Only the instances of occurrences of chained and converging computations need to be evaluated. Unconnected and diverging computations are likely produced by programming errors. They are removed from $TD_P(p)$[2].

6.2.3 Generating visit-sequences

After removing unconnected and divergent computational paths from $TD_P(p)$, all remaining attribute occurrences require evaluation. We can therefore traverse these paths and transform every node of the path into a step of a visit-sequence. Output occurrences (Equation 2.5) are transformed in to evaluation instructions, while input occurrences (Equation 2.4) are translated into child visits. Exceptions are the inherited occurrences of the parent of p, as they are transformed into suspend instructions. The visit-sequence we obtain this way has to satisfy three conditions:

[2]A user friendly compiler should warn programmers of any attributes that are not evaluated because they are part of unconnected or diverging computations.

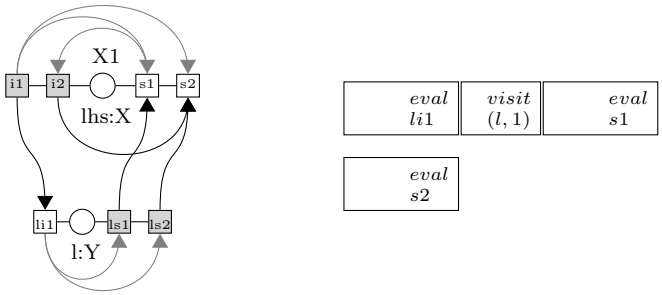

Figure 6.3: An example of schedule graph TD_P (direct dependencies are black, intra-visit dependencies are gray), together with the visit-sequences calculated from it. The dependency $ls2 \rightarrow s2$ is an inter-sequence dependency.

1. Every eval-instruction should follow the instructions (either eval instruction or child visit) on which it depends.
2. Every child visit should follow all eval-instructions of the inherited occurrences required as input of that visit.
3. Every i-th suspend instruction should follow the eval-instructions of all synthesized occurrences that form the output of the i-th visit to the parent of p.

The first condition can be met by removing nodes from the graph after they have been transformed into an instruction and transforming only nodes to eval instructions when the node has no incoming arrows (no more pending dependencies). Every child visit, although it produces one or more synthesized occurrences, should only appear once, therefore we merge all the occurrences it produces into a new *visit node*, in a process called condensing[Pen94]. By transforming the new visit node to a visit-instruction when it has no incoming arrows, it is guaranteed that all the inherited occurrences required for the visit have been evaluated (second condition). The same applies to suspend instructions, except this time we merge inherited occurrences of the parent into a single visit node (third condition). The procedure shown in Figure 6.4 computes a set of visit-sequences for $TD_P(p)$, one for every visit to the parent of p, such that the above conditions are satisfied.

6.3 Proposed optimisations

From the code generation models of Section 6.1 we extract a number of possible optimisations to judge the strength of a scheduling solution. These optimisations are generally not mutually exclusive: minimising cost with respect to one optimisation might increase costs associated with another. We do not wish to quantify the optimisations to compare different schedules empirically.

The proposed optimisations can be used as test cases to verify the feasibility of performing any optimisations with the LOAG algorithm.

6.3.1 Minimising the number of visits

The total number of visits is a good indication of the size of treewalks produced for any derivation tree. Although to know the real treewalk the derivation tree has to be known.

```
VSS (p : Production, tdp : Production → Graph) =
  m_deadends = sinks (tdp (p));    -- deadends contains possible deadends
  while (¬ (empty (deadends)))     -- remove unconnected and divergent paths
    n = deadends.pop ();
    if (¬ (field (n) ≡ parent (p)) ∧ is_synthesized (n) ∧ is_sink (n))
      remove (n, tdp (p));
      deadends = deadends ∪ preds (n);
  foreach (i : Int in visits (parent (p)))    -- initialise empty sequences
    seq [i] = { };
  for (i : Int in visits (parent (p)))    -- merge nodes that are related to condition 2
      -- obtain all the inherited occurrences of the i-th
      -- visit of the interface if the parent of p
      -- and merge them into a single node (without self-edges)
      -- it returns a new node V(f,i) where f is the field to which
      -- the visit should be made.
    occs = inherited (visit (i, parent (p)));
    condense (i, occs, True, tdp (p));
  for (c : Field in children (p))        -- merge nodes that are related to condition 3
    for (i : Int in visits (c))
        -- same as above, except for children we merge synthesized attributes
      occs = synthesized (visit (i, c));
      condense (i, occs, False, tdp (p));
  visitnr = 0;
  startpoints = sources (tdp (p))
  while (¬ (empty (startpoints)))      -- transform nodes into instructions
    n = startpoints.pop ();
    possible_sources = succs (n);
    remove (n, tdp (p));
    startpoints = startpoints ∪ (filter (is_source, possible_sources))
    case n of
      V (f, i) → if f ≡ parent (p)          -- all required occurrences of this visit
                 then visitnr++;            -- have been transformed, acts as suspend
                 else seq [i].append (Visit (f, i))   -- perform i'th visit to child f.
      α        → seq [i].append (Eval (α))   -- perform the evaluation of attribute α
```

Figure 6.4: A procedure that calculates dependency respecting visit-sequences from graph $TD_P(p)$.

A schedule with more visits generally requires a treewalk that includes more procedure calls, increasing the workload of a call or pattern matching stack, even though the number of attribute evaluations required during the decoration process remains the same.

As noted before, without fixing the execution model it is impossible to quantify the costs of a visit. We assume, however, that in any execution model there will be some overhead for function application. We therefore propose the following optimisation.

Optimisation 1. *A schedule with less visits is preferred.*

This criteria judges the total number of visits, the sum of the number of visits required for every non-terminal. This approach is sensible when no knowledge of possible derivations trees is given. In some situations a programmer might know that a certain non-terminal occurs frequently in derivations trees for his grammar. In this case it might be more beneficial to reducing the number of visits for that particular non-terminal instead of the sum over all non-terminals.

6.3.2 Minimising the number of inter-sequence dependencies

A pair of attribute occurrences $(X_p^i \cdot a, X_p^j \cdot b)$ forms an *inter-sequence dependency* if $(X_p^i \cdot a \to X_p^j \cdot b) \in D_P$ and $(X_p^i \cdot a)$ is evaluated during the u-th visit to the parent of p while $(X_p^j \cdot b)$ is evaluated during the v-th visit to the parent of p with $u < v$. As noted in Section 6.1.2, it is set of dependencies that represent a flow of information that poses a logistic difficulty in pure evaluators. By minimising the number of inter-sequence dependencies we minimise the lifetime (time required in memory) of attributes. An example of an inter-sequence dependency is given in Figure 6.3

Optimisation 2. *A schedule with less inter-sequence dependencies is preferred.*

6.3.3 Optimising incremental behaviour

An incremental evaluator typically receives as input the decorated parse tree of a previous run and detects changes in a new tree with respect to the tree that was used to produced the previous result. Whenever there is a change in the value of a attribute instance a, all the instances depending on a have to be reevaluated. Not all instances b depending on a actually require an update. Some dependencies on the path from a to b are induced by the interfaces. By minimising the lengths of all paths that protrude from an attribute occurrence, we minimise the number of reevaluations required when an instance of that attribute occurrence changes. We propose the following optimisations.

Optimisation 3. *Whenever the value of an attribute instance a is expected to change often due to modifications of the parse tree, a schedule in which a is scheduled later is preferred.*

Which attribute is expected to change often should be indicated by the programmer.
We can do the opposite for attributes that are costly to evaluate.

Optimisation 4. *Whenever an attribute b is particularly costly to evaluate, a schedule in which b is scheduled sooner is preferred.*

These optimisations can be combined.

Optimisation 5. *Whenever the value of an attribute a is expected to change often due to modifications of the parse tree and attribute b is particularly costly to evaluate, a schedule in which b is independent of a is preferred.*

6.4 Performing optimisations

Separate implementations

The proposed optimisations can be implemented inside an LOAG scheduling algorithm or as separate procedures. We chose to consider the optimisations as a separate problem. We take the following approach: given a valid schedule (a set of complete, well-defined and compatible interfaces or graph TD_P), adjust the schedule such that it is improved according to the optimisation criteria, while remaining valid.

Although this approach is not ideal for the runtime of the pipeline (input AG $\rightarrow$ optimised evaluator), we expect that it grants us the freedom to extend it with arbitrary optimisations. It allows us to solve every optimisation problem in a different way.

Combining optimisations

Another task is deciding how to combine optimisations. One can imagine situations in which a linear combination of certain optimisations is desired ("optimising the schedule with respect to optimisation x is twice as important as optimising it with respect to y"). This problem requires an optimisation procedure that uses an arbitrary cost function to evaluate schedules. A genetic algorithm or simulated annealing algorithm might be successful, given some mutation operator or neighbourhood that produce only valid schedules. It is also not clear how a programmer should be able to provide such a cost function to the optimisation procedure.

We have decided not to combine optimisations other than considering them one by one in a certain order. Optimisations can be sorted in order of the expected impact (amount of changes) the optimisations entail. For example, Optimisation 3 and 4 of the proposed optimisations are expected to have smaller impact than Optimisation 1. Deciding in which order to perform the optimisations can also be left to the programmer. A programmer can specify which optimisations are desired (and in which order) using command line flags or syntax in the AG description.

Focus

In our attempts to show how optimisations can be implemented, we have focused on reducing the number of visits (Optimisation 1) as an extension of the LOAG algorithm. We can reuse the Boolean formula used for finding an arbitrary schedule and manipulate it to find a schedule that satisfies additional criteria. As such, the optimisation procedure does not only take the current set of interfaces as input, but also the Boolean formula used to construct it (the procedure is not separated from the LOAG algorithm entirely).

First we investigate the relation between Kastens' algorithm and the AOAG algorithm concerning Optimisation 1. From the investigation we extract a way to improve the arbitrary schedule produced by the LOAG algorithm (the SAT-solver finds an arbitrary satisfying assignment). We wished to add procedures for all of the proposed optimisations of the previous section. Due to a lack of time, this is left as future work.

6.4.1 Kastens' algorithm

Kastens' algorithm finds a schedule, for OAGs, that contains the minimum number of visits in each of the interfaces. The interfaces are constructed from the graph ID_S (see Section 3.4.1). Graph ID_S is constructed such that only direct dependencies and the paths they induce are represented. The number of visits in an interface is determined by the longest paths of intra-visit dependencies implied by the interface. Consider the graphical representation of an interface,

e.g. in Figure 6.5. The intra-visit dependencies are all the arrows we can draw from attributes in any column c to the attributes in the column to the right of c. To obtain an interface with a smaller number of visits, at least one intra-visit dependency of every longest path has to be swapped. However, there is at least one longest path in every interface that consists only of dependencies in $ID_S(X)$ (based on the way Kastens' algorithm constructs interfaces). None of the dependencies in $ID_S(X)$ can be swapped as it would result in a type 1 error. Therefore, there is no way to shorten all of the longest paths and to reduce the number of visits.

6.4.2 AOAG algorithm

The AOAG algorithm applied to an OAG produces the same optimal schedule as Kastens' algorithm. If it is applied to an LOAG that is not OAG, the algorithm performs a small number of correcting changes to the optimal (but invalid) schedule Kastens' algorithm produces for it, by swapping intra-visit dependencies. The results is a set of interfaces with a larger number of visits.

The number of newly introduced visits is not easily predicted as swapping an intra-visit dependency might alter the interfaces for other non-terminals as well (through induced paths). Moreover, the algorithm takes only a limited number of candidates into account. Perhaps fake dependencies exist that are not considered as candidates, according to the lemmas in Section 4.3, that resolve all conflicts *and* produce smaller interfaces. We have not investigated this claim. It is interesting to see whether it is possible to predict the number of newly introduced visits. If so, choosing the right candidate might result in better schedules. It is also interesting to see whether considering more candidates actually leads to more backtracking.

In order to take the other optimisation criteria into account the algorithm needs to be adjusted more drastically. Instead we have chosen to show how the LOAG algorithm can be extended to reduce the size of the interfaces it produces.

6.4.3 LOAG algorithm

To minimise the number of visits in the constructed interfaces we have to minimise the length of the largest path of intra-visit dependencies for every non-terminal. Consider the graphical representation of an interface in Figure 6.5. Every arrow represents an assignment to one of the variables in a non-terminal problem graph. In the image, all edges must point from left to right. As opposed to the interfaces constructed by Kastens' algorithm, the interfaces that follow from an assignment to the problem graph might now contain longest paths with intra-visit dependencies that are not already in ID_S. The first interface (left) in Figure 6.5 contains three visits, as it contains three columns of inherited and three columns of synthesized attributes. If the orders of the assignments to the variables corresponding to edges $i1 \rightarrow s1$, $i1 \rightarrow s2$, $i2 \rightarrow s1$ and $i2 \rightarrow s2$ are swapped simultaneously we obtain the second (right) interface of Figure 6.5 with two visits. Swapping an assignment is not possible if the reverse assignment causes a type 1, 2 or 3 cycle. The assignments must be swapped simultaneously. We maintain a path through of six intra-visit dependencies if we maintain one of the assignments. Moreover, swapping just a subset of the assignments results in an *increase* of of the number of visits. Consider swapping *only* dependency $(i_1 \rightarrow s_1)$. Say that s_1 is reassigned to S_3 in order to adhere the new dependency. Since assignment $(i_2 \rightarrow s_1)$ is still in place, i_2 should be reassigned to I_3. All the attributes in S_3 point to i_2 and should therefore be reassigned to a new column S_4. All attributes in I_3 are assigned to new column I_4. We end up with an interface with *four* visits.

We have implemented a procedure performing the following steps:

- For every non-terminal:

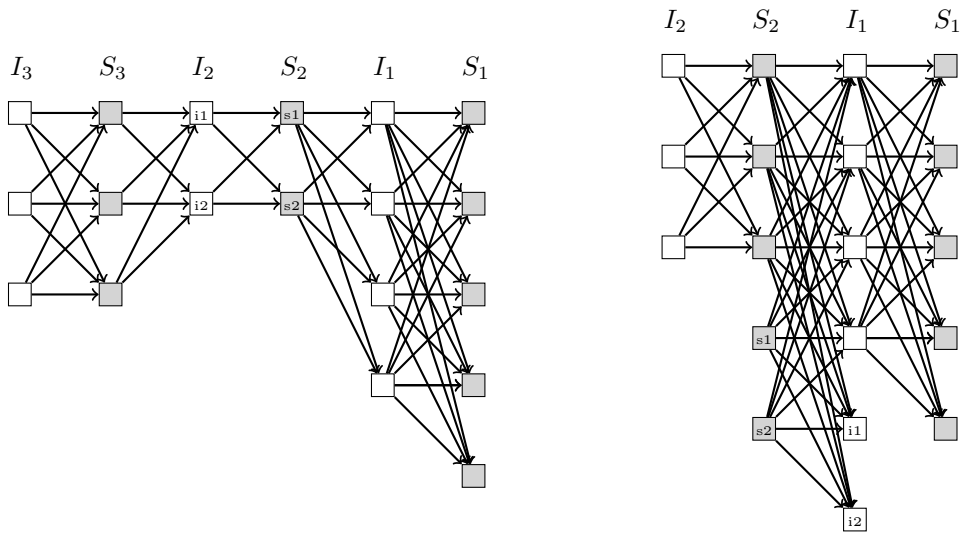

Figure 6.5: If edges $i1 \to s1$, $i1 \to s2$, $i1 \to s1$ and $i1 \to s2$ from a bottleneck are swappable simultaneously, less visits are present in the resulting interface.

- For every pair $I_i \times S_i$ (or $S_i \times I_{i-1}$)):
  - Count all dependencies $(a \to b)$ with $a \in I_i$ (or S_i) and $b \in S_i$ (or I_{i-1}) and sort the pairs $I_i \times S_i$ (or $S_i \times I_{i-1}$)) based on the count ascendingly. In that order, query the SAT-solver to find out if it all the dependencies can be swapped and whether the swap results in an interface with less visits. If so, add clauses to the SAT-solver that forces these dependencies are swapped.

The order in which the non-terminals and the 'bottlenecks' are considered is very important. The procedure is heuristic and does not produce optimal schedules.

Figure 6.6 show the best results obtained using this procedure when applied to to the main AG of the UHC and another AG description from the UHC called 'ToGrin'. It is interesting to see that the LOAG algorithm with optimisation produces a smaller maximum number of visits for ToGrin than the AOAG algorithm (proving that AOAG does not produce optimal schedules). The results for the main AG are not that good, although the maximum number of visits did drop by 15%. Note that the minimising the maximum number of visits might not yield the best result for all derivation tree given to the optimised evaluator. Decorating derivation trees in which certain non-terminals occur frequently might be less costly if the optimisation focuses on the frequent non-terminals.

AG	AOAG algorithm	LOAG algorithm	LOAG with optimisation
MainAG	(13, 139, 4.63)	(19, 204, 6.80)	(19, 173, 5.77)
ToGrin	(3, 40, 2.00)	(7, 67, 3.35)	(3, 32, 1.60)

Figure 6.6: Table showing the maximum, total and average number of visits for two large AGs from the Utrecht Haskell Compiler, as computed by the AOAG and LOAG algorithms.

While writing this final section of the thesis, better results have been achieved by directly encoding the lenghts of dependency paths in interfaces into the SAT-problem. As such, the SAT-solver can be forced to find increasingly smaller interfaces, until the optimum is reached. The

resulting maximum number of visits found for ToGrin is 26 and 130 for the main AG. As opposed to the aforementioned approach, this approach has not yet been fully tested and implemented in the UUAGC.

6.5 Conclusions

In this chapter we have shown how to generate evaluators from the schedules produced for arbitrary LOAGs. The evaluators compute the values of the attribute instances of all possible derivation trees in a statically determined order. As the order is known at compile time, the compiler can optimise the evaluator regarding arbitrary optimisation criteria.

We have proposed a number of criteria. The criteria have been chosen to support the optimisation of three important types of evaluators: evaluators generated in imperative languages, evaluators generated in purely functional languages and incremental evaluators.

Unfortunately we were not able to extend the AOAG or LOAG algorithm to optimise with respect to all of the proposed criteria. Using one optimisation criteria as a test case we have convinced ourselves that the LOAG algorithm is capable of performing arbitrary optimisations. We are therefore confident that the LOAG algorithm is useful in an AG compiler that generates evaluators for all kinds of host languages and execution models. Actually implementing the implementations proposed in this chapter, among others, is left as future work.

Chapter 7

Related Work

7.1 OAG*

According to Deransart et al.[DJL88] there exists a family of subclasses of LOAG, named OAG(i) by Barbar, of which Kastens' OAG is a member. Another family member is the class OAG* introduced by Natori et al.[NGI+99]. Their paper, introducing OAG*, is (also) motivated by the wish to schedule LOAGs without the need for fake dependencies. The described algorithm finds strongly connected components on the direct dependencies between attributes (sharing direct dependencies between occurrences of the same attribute). As a result, more information about the dependencies at production level is available for the procedure that constructs the interfaces (non-terminal level).

Class OAG* recognises a larger subclass of LOAGs than Kastens' algorithm. It is interested to compare the class OAG* with LOAG in greater detail, analysing the gap between the two classes. Furthermore, it is interesting to see whether the main AG of the UHC (and other large LOAGs developed at Utrecht University) are a member of OAG*. If so, OAG* is a good alternative to the algorithms developed in this thesis.

7.2 DAT-graphs

In his PhD thesis[Pen94], Pennings investigates incremental evaluators generated from visit-sequences. Although incremental attribute evaluation is beyond the scope of this thesis, his work has inspired the optimisation concerns discussed in Chapter 6. In order to generate visit-sequences that are better suited in an incremental setting, Pennings defines a new member of the OAG(i) family, using *DAT-Graphs*. Pennings articulates the need for more control over the schedules produced in specific AG implementations. LOAGs are helpful here, as static evaluation orders are required to have direct control over the generated evaluators.

7.3 Purely functional implementation

The purely functional implementation of OAGs described in Saraiva's PhD thesis forms the basis of the UUAGC's implementation of Kastens' algorithm. Lazy code can be generated with the UUAGC in the form of folds and algebras as described by Swierstra et al.[SAS99]. Our implementation of the AOAG algorithm also depends on this work. In his thesis, Saraiva describes

the implementation of all aspects of an AG compiler, including higher-order attributes[VSK89], local attributes and incrementality, in a pure functional language[Sar99].

7.4 Chordal graphs and satisfiability

The notion of triangulated (chordal) graphs is used to solve many different types of problems and much work has been done to find minimal elimination orders efficiently[Ros70, AKR91]. One of many interesting examples of the need to find a specific triangulation is in probabilistic network theory[BKE05]. The idea to relate chordal graphs to Boolean satisfiability comes from Bryant et al.[BV02].

Chapter 8

Results and Future Work

8.1 Theoretical contributions

This thesis provides a detailed analysis of the Linear Ordered Attribute Grammars, including: different characterisations of the class, the intuitions behind the characterisations, the benefits of the evaluators generated for LOAGs, and an analysis of the problem of scheduling LOAGs, resulting in a minimal formulation of this decision problem.

Two characterisations are investigated in further details, namely LOAGs as Arranged Orderly Attribute Grammars (AOAGs) and Partitionable Attribute Grammars (PAGs).

AOAGs use the notion of fake dependencies, a familiar trick to transform LOAGs into Ordered Attribute Grammars (OAGs). This thesis examines how fake dependencies are used and describes a method for finding them automatically. The result is an algorithm for recognising and scheduling all LOAGs.

PAGs use interfaces, one for every the non-terminal, to define a linear evaluation order. This thesis describes how interfaces are constructed and how evaluators are generated from them. By fixing an interface for a non-terminal a number of intra-visit dependencies are introduced. The intra-visit dependencies might contradict each other or the intra-visit dependencies imposed by other interfaces. An algorithm is given to select a set of non-conflicting intra-visit dependencies from which valid interfaces are constructed. The algorithm constructs a Boolean formula for which a satisfying assignment is used to find the intra-visit dependencies. The problem is thus defined as a SAT-problem.

This SAT-based algorithm solves the problem of scheduling LOAGs in an abstract fashion such that it can easily be tweaked for different purposes. The algorithm allows programmers to optimise the schedules it produces. In this thesis a number of optimisation criteria are proposed. The SAT-based algorithm is expected to be extendable to include some of these optimisations.

8.2 Practical contributions

The main motivation for this project is compiling the main AG of the UHC without using fake dependencies. Not only did we achieve this, the main AG can now be compiled faster (shown in Figure 8.1) using the LOAG algorithm. We have implemented both algorithms in the UUAGC. The LOAG algorithm has been implemented using the popular and prize winning SAT-Solver Minisat[1].

[1] Available at: https://github.com/niklasso/minisat

Algorithm	Fake deps?	main AG	Compiletime UHC	Runtime UHC
Kastens'	Y	25s	3m48s - 25s = 3m23s	3m46s
K&W	N	2s	4m27s - 02s = 4m25s	3m58s
AOAG	N	35s	4m43s - 35s = 4m08s	3m55s
LOAG	N	10s	4m23s - 10s = 4m13s	3m56s
LOAG=1	N	14s	4m27s - 14s = 4m13s	3m58s

Figure 8.1: Comparison of the four algorithms discussed in this thesis. Kastens' algorithm (1980), K&W revisited (2012), AOAG for automatic selection of fake dependencies and LOAG using Minisat to construct interfaces. LOAG=1 is the LOAG algorithm extended with visit number reduction (Section 6.4.3).

The table shows for every algorithm mentioned in this thesis how long it takes to schedule the main AG of the UHC, the compiletime required to build the complete UHC and the runtime of the created executable when it is used to build a set of standard Haskell libraries (Data and System among others). The compile time of the complete UHC includes compiling all its AG descriptions and compiling the Haskell code generated from them. The UHC is compiled into an executable by the Glasgow Haskell Compiler (GHC). The compiletime is dominated by the the GHC's work and the work for compiling the main AG. By subtracting the time required for the main AG we have an indication of the time spent by the GHC on the code generated by the different algorithms.

The difference in compile and runtime of the UHC between Kastens' algorithm and the others is likely explained by the fact that it is the only algorithm using a different code generation procedure. Therefore the comparison is not fair. Comparing the other algorithms we see that the AOAG algorithm is the fastest in compiling the UHC and also that the executable produced is slightly faster. The former can be explained by the smaller total number of visits (139, against 204 for LOAG and 173 for LOAG=1). Less visits means less visit functions to generate resulting in a smaller Haskell source file (2.5mb for AOAG on the main AG, against 2.8mb for LOAG and 2.8mb for LOAG=1) and in a smaller executable (28.0mb, against 29.0mb and 28.8 mb). The difference in runtime might also be explained by the smaller number of visits, although the difference is very small. Moreover, LOAG=1 is slight slower than LOAG, contradicting this observation. The differences are so small that they might be caused by external factors.

The Kennedy & Warren algorithm revisited[BMDS12] compiles the main AG very fast. However, it does not find a static evaluation order. The evaluation order is determined at runtime which means that the K&W algorithm is likely to produce duplicated and code. Some visits will produce the same synthesized attributes and not all of them will be used by the evaluator. We expect that code duplication explains the slightly larger compile time between K&W and AOAG, LOAG and LOAG=1. We would also expect to see a slight overhead from deciding which visit to perform reflected in the runtime of the UHC. Although the runtime is slightly larger compared to AOAG and LOAG we are not convinced. Perhaps the runtime of the UHC is dominated by algorithms that are not defined as AGs. For example type inferencing is not implemented as an AG as unification of types is not easily defined with AGs[MDS10].

8.3 Future work

8.3.1 Experimenting with ANCAG's and LOAG's preconditions

Section 3.3.3 describes the induced dependency graph ID_P used in Kastens' algorithm and compares it with the induced dependency graph DG_P used in Kennedy-Warren. Both graphs are used as a precondition test for their respective algorithms. A cycle in the induced dependency

graph implies that AG can not be scheduled with that algorithm. Graph ID_P contains graph DG_P entirely, therefore it defines a class of AGs strictly included in Kennedy and Warren's ANCAG and a strict super class of LOAG (see Figure 8.2).

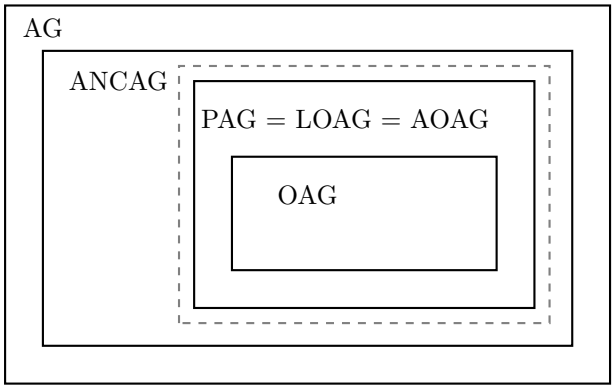

Figure 8.2: AGs with an acyclic induced dependency graph ID_P form a super class of LOAG (gray and dashed).

We have not examined the class of AGs that satisfy the precondition but are not linear ordered. The main reason is that AGs in this gap have not been encountered in practical examples available to us. Natori et al.[NGI+99] have given an example of such an AG, showing that the gap exists (the precondition defines a strict superclass of LOAG).

AOAG approach

The gap is interesting as it contains the AGs that might pose problems for the AOAG algorithm. They are not linear ordered, so the algorithm will not be able to find a schedule. Moreover, the precondition has been met and the backtracking procedure will be called. In order to figure out that the AG is not linear ordered, the algorithm requires a full search through the search space. We have neither theorised about the size of such a search nor ran experiments, due to the lack of practical examples. In the example given by Natori et al. a certain combination of productions is given that 'fool' any LOAG algorithm: every possible fake dependency propagates to another production rule to cause a type 2 (cycle in ID_P). We expect more of these combinations exist. The combinations can be used by a *generator* for AGs. It might be interesting to develop a tool that takes an input LOAG and transforms it, using problematic combinations of productions, into an AG that satisfies the precondition but is no longer a LOAG. With this tool we can generate arbitrary variations on the main AG of the UHC to use as test cases for the AOAG algorithm.

SAT approach

The LOAG algorithm tests no preconditions at all. All cycle detection is delegated to the SAT-solver. It is interesting to see whether AGs that do not satisfy the preconditions of either Kennedy-Warren and Kastens' algorithm are quickly diagnosed by the SAT-solver.

8.3.2 SAT-solvers and cycle detection

The LOAG algorithm proposed in Chapter 5, uses the notions of problem and assignment graphs to separate graphs in which cycles are disallowed (assignment graphs) from the graphs represented

by Boolean formulas (problem graphs, in which variables denote the direction of an edge). The separation allows us to reason about possible graphs a SAT-solver might produce, in terms of the variables of the SAT-problem. The approach is very general and can be used for all problems in which cycles have to be prevented or detected, relying on Theorem 5.1 (Section 5.2). The abstract problem Chapter 5 solves is "Is it possible to transform graph $G = \langle V, E \rangle$ into an acyclic graph, by swapping only edges in some $Q \subseteq E$?".

We expect that this problem has instances in many different fields of computing science. It is interesting to investigate the applicability of the SAT approach to the problem in different domains. Using domain specific knowledge the SAT-problem can be reduced. Section 5.3 shows how this is done for LOAGs, providing an exemplary use case, relying on additional Theorems.

The approach might generalised further to place arbitrary other constraints (other than acyclicity) on assignment graphs, when Theorems alternative to Theorem 5.1 can be proven.

8.3.3 Formalising efficiency concerns of generated code

As discussed in Chapter 6, the runtime efficiency of the evaluators generated by different scheduling algorithms is not easily compared. No indefinite answer to the question "Which currently know AG scheduling algorithm produces the fastest code?". We can ask similar questions about other concerns as well: for example, memory usage, size of generated code, compiletime of generated code, degree of incrementality and the runtime of the scheduling algorithm itself. In general, we would like to compare executions of evaluators in detail to pinpoint any weaknesses that an AG compiler can circumvent. To address these issues, at least the following has to be done:

- Implement an AG compiler that contains multiple pipelines, one for each algorithm to use in the comparison. The pipelines should share all preprocessing steps and as much as possible of the code generation phase. The preprocessing phase influences the produced code by, for example, generating extra semantic function definitions from USE and COPY rules. Clearly the code generation phase influence the result greatly. For example, a bug in this phase might produce a duplication of code. Therefore, any comparison is only fair if the preprocessing and code generation is shared by all algorithms. It also allows a fair comparison of the runtime of the scheduling algorithms themselves.

- The machine executing the generated code should provide a large number of statistics, preferably at every point in time. To inspect the the evaluators in detail, it should be possible to perform an execution step by step.

- Collect a large number of AGs of different classes, possibly by generating them. However, generated AGs do not necessarily represent realistic AGs encountered in practice.

Abstract State Machine

In order to perform the investigations some form of Abstract State Machine (ASM) can be used. The evaluators are generated in the pseudo-code, specialised for tree-based for tree-base computations, designed for the ASM. The notion of ASMs is ideal for comparing and analysing code fragments. A virtual machine executes the code on a physical machine. In this model programming with AGs is entirely standalone (without host language). Whether standalone AGs are desired and easily integrated in practical programming environments, remains to be seen.

Implementing optimisations

The investigation proposed here, might raise a number of weaknesses (looking at runtime, space complexity, etc.) of evaluators generated from visit-sequences. These weaknesses can be taken into a account in an algorithm for scheduling LOAGs. Chapter 6 proposes a number of optimisations that address *expected* weaknesses. Either optimisation criteria found by experimentation or the criteria proposed in Chapter 6 can be used to show that LOAGs are capable of producing efficient evaluators. Therefore we would like to see a number of optimisations implemented in the LOAG algorithm presented in this thesis.

8.3.4 Acknowledgements

I would like to thank Jeroen Bransen and Atze Dijkstra for supervising me throughout what must be one of the best supervised master projects at the UU in recent years.

With Jeroen I was able to discuss both the big picture and the fine details of the problems I was facing. Especially his help in keeping my Haskell implementations efficient is greatly appreciated.

Atze was of great help discussing the issues in a broad perspective, reminding me of what I am doing, why I am doing it and outlining the different ways to proceed. Without his guidance I would have certainly lost myself in little and insignificant details.

Finally I would like to thank Koen Claessen whose visiting professorship at UU could not have been timed better. We discussed different (SAT-)formulations of the LOAG scheduling problem at length and his enthusiasm for the subject kept me motivated to keep looking. Without his help there would have been a lot of ideas and no successful implementation of the LOAG algorithm.

Bibliography

[AKR91] Ajit Agrawal, Philip Klein, and R Ravi. Approximating fill in solving sparse linear systems. 1991.

[BKE05] Hans L. Bodlaender, Arie MCA Koster, and Frank van den Eijkhof. Preprocessing rules for triangulation of probabilistic networks. *Computational Intelligence*, 21(3):286–305, 2005.

[BMDS12] Jeroen Bransen, Arie Middelkoop, Atze Dijkstra, and S. Doaitse Swierstra. The kennedy-warren algorithm revisited: Ordering attribute grammars. In *Proceedings of the 14th International Conference on Practical Aspects of Declarative Languages*, PADL'12, pages 183–197, Berlin, Heidelberg, 2012. Springer-Verlag.

[BV02] Randal E. Bryant and Miroslav N. Velev. Boolean satisfiability with transitivity constraints. *ACM Transactions on Computational Logic (TOCL)*, 3(4):604–627, 2002.

[Dij05] Atze Dijkstra. *Stepping Through Haskell*. 2005.

[DJL88] Pierre Deransart, Martin Jourdan, and Bernard Lorho. *Attribute Grammars: Definitions, Systems, and Bibliography.*, volume 323 of *Lecture Notes in Computer Science*. Springer, 1988.

[EF82] Joost Engelfriet and Gilberto Filè. Simple multi-visit attribute grammars. *Journal of computer and system sciences*, 24(3):283–314, 1982.

[HITT97] Zhenjiang Hu, Hideya Iwasaki, Masato Takeichi, and Akihiko Takano. Tupling calculation eliminates multiple data traversals. *ACM Sigplan Notices*, 32(8):164–175, 1997.

[Kar72] Richard M. Karp. Reducibility among combinatorial problems. In Raymond E. Miller and James W. Thatcher, editors, *Complexity of Computer Computations*, The IBM Research Symposia Series, pages 85–103. Plenum Press, New York, 1972.

[Kas80] Uwe Kastens. Ordered attributed grammars. *Acta Informatica*, 13(3):229–256, 1980.

[Kas91] Uwe Kastens. Implementation of visit-oriented attribute evaluators. In *Attribute Grammars, Applications and Systems*, pages 114–139, 1991.

[Knu68] Donald E. Knuth. Semantics of context-free languages. *Mathematical systems theory*, 2(2):127–145, 1968.

[KW76] Ken Kennedy and Scott 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.

[MDS10] Arie Middelkoop, Atze Dijkstra, and S. Doaitse Swierstra. Iterative type inference with attribute grammars. In *Proceedings of the Ninth International Conference on Generative Programming and Component Engineering*, GPCE '10, pages 43–52, New York, NY, USA, 2010. ACM.

[NGI+99] Shin Natori, Katsuhiko Gondow, Takashi Imaizumi, Takeshi Hagiwara, and Takuya 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.

[Pen94] Maarten C. Pennings. *Generating incremental attribute evaluators.* Ph.D. thesis, Computer Science, Utrecht University, November 1994.

[PJea03] Simon Peyton Jones et al. *Haskell 98, Language and Libraries. The Revised Report.* Cambridge University Press, 2003.

[Ros70] Donald J. Rose. Triangulated graphs and the elimination process. *Journal of Mathematical Analysis and Applications*, 32(3):597–609, 1970.

[RT89] Thomas W. Reps and Tim Teitelbaum. *The synthesizer generator reference manual (3. ed.).* Texts and monographs in computer science. Springer, 1989.

[Sar99] João Saraiva. *Purely Functional Implementation of Attribute Grammars: Zuiver Functionele Implementatie Van Attributengrammatica's.* IPA dissertation series. IPA, 1999.

[SAS99] S. Doaitse Swierstra, Pablo R. Azero Alcocer, and João Saraiva. Designing and implementing combinator languages. In *Advanced Functional Programming*, pages 150–206. Springer Berlin Heidelberg, 1999.

[Swi05] Wouter Swierstra. Why Attribute Grammars Matter. *The Monad.Reader*, 4, July 2005.

[VSK89] Harald Vogt, S. Doaitse Swierstra, and Matthijs. F. Kuiper. Higher order attribute grammars. In *Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation*, PLDI '89, pages 131–145, New York, NY, USA, 1989. ACM.

[VSS09] Marcos Viera, S. Doaitse Swierstra, and Wouter Swierstra. Attribute grammars fly first-class: how to do aspect oriented programming in haskell. *Acm Sigplan Notices*, 44(9):245–256, 2009.

Index