Linearly Ordered Attribute Grammar Scheduling Using SAT-Solving

Jeroen Bransen[1], L. Thomas van Binsbergen[2,1], Koen Claessen[3], and Atze Dijkstra[1]

[1] Utrecht University, Utrecht, The Netherlands
{J.Bransen,atze}@uu.nl
[2] Royal Holloway, University of London, Egham, UK
ltvanbinsbergen@acm.org
[3] Chalmers University of Technology, Gothenburg, Sweden
koen@chalmers.se

Abstract. Many computations over trees can be specified using attribute grammars. Compilers for attribute grammars need to find an evaluation order (or *schedule*) in order to generate efficient code. For the class of *linearly ordered attribute grammars* such a schedule can be found statically, but this problem is known to be NP-hard.

In this paper, we show how to encode linearly ordered attribute grammar scheduling as a SAT-problem. For such grammars it is necessary to ensure that the dependency graph is cycle free, which we approach in a novel way by transforming the dependency graph to a chordal graph allowing the cycle freeness to be efficiently expressed and computed using SAT solvers.

There are two main advantages to using a SAT-solver for scheduling: (1) the scheduling algorithm runs faster than existing scheduling algorithms on real-world examples, and (2) by adding extra constraints we obtain fine-grained control over the resulting schedule, thereby enabling new scheduling optimisations.

Keywords: Attribute Grammars, static analysis, SAT-solving.

1 Introduction

Attribute Grammars [Knuth, 1968] are a formalism for describing the semantics of context-free languages, thereby making them suitable for the construction of compilers. Examples of compilers written using attribute grammars are the Java compiler JastAddJ [Ekman and Hedin, 2007], and the Haskell compiler UHC [Dijkstra et al., 2009]. The use of attribute grammars in the UHC motivates this paper and as such UHC is a real world test case for evaluating the effectiveness of the given approach.

An attribute grammar essentially describes computations over trees, also known as folds or catamorphisms. Although the attribute grammar defines exactly *what* should be computed, it does not define *when* values should be computed. It is therefore up to the attribute grammar compiler (the compiler that

© Springer-Verlag Berlin Heidelberg 2015
C. Baier and C. Tinelli (Eds.): TACAS 2015, LNCS 9035, pp. 289–303, 2015.
DOI: 10.1007/978-3-662-46681-0_24

translates the attribute grammar definition into a runtime evaluator) to find an evaluation order. Such an evaluation order or schedule should satisfy the dependencies induced by the specification of attribute computations in the attribute grammar definition and must be found statically, which means that for the given grammar an evaluation order should be found such that for every finite abstract syntax tree of the grammar the evaluation order should compute all values in a finite number of steps. Specifically, no cyclic data dependencies may occur at runtime for any given parse tree.

The class of *ordered attribute grammars* [Kastens, 1980] is a well-known subclass of attribute grammars for which a polynomial time scheduling algorithm exists. However, despite what the name suggests there exist attribute grammars for which a static evaluation order can be found that do not belong to the class of ordered attribute grammars. In our work on compiler construction we often encountered such grammars, giving rise to the need of other scheduling algorithms [Bransen et al., 2012, Van Binsbergen et al., 2015]. We therefore look at the class of *linearly ordered attribute grammars* in this paper, which is the largest class of attribute grammars for which a static evaluation order can be found. The problem of statically finding such evaluation order is known to be NP-complete [Engelfriet and Filè, 1982].

1.1 Summary

We solve the linearly ordered attribute grammar scheduling problem by translating it into the Boolean satisfiability problem (SAT), the standard NP-complete problem [Cook, 1971]. Even though the worst case runtime of all known SAT-solving algorithms is exponential in the input size, many SAT-solvers work very well in practice [Claessen et al., 2009]. By translating into the SAT problem we can therefore use an efficient existing SAT-solver to solve our problem, and even benefit from future improvements in the SAT-community. In our implementation we use MiniSat[1] [Eén and Sörensson, 2004].

The core of the scheduling problem consists of finding a total order on the nodes of a set of dependency graphs (directed acyclic graphs) such that all direct dependencies, which are coming from the input grammar, are satisfied. However, there is interplay between the different dependency graphs caused by the order that is found in one dependency graph resulting in indirect dependencies that need to be satisfied in another dependency graph. This interplay of dependencies, which is explained later in detail, is what makes the problem hard.

To encode this problem in SAT we represent each edge in the dependency graphs as a Boolean variable, with its value indicating the direction of the edge. For the direct dependencies the value is already set, but for the rest of the variables the SAT-solver may choose the direction of the edge. Ensuring cycle-freeness requires us to encode transitivity with SAT-constraints, which in the straight-forward solution leads to a number of extra constraints cubic in the number of variables. To avoid that problem we make our graphs *chordal*

[1] `http://minisat.se`

[Dirac, 1961]. In a chordal graph every cycle of size > 3 contains an edge between two non-adjacent nodes in the cycle. In other words, if there exists a cycle in the graph, there must also exist a cycle of at most three nodes. This allows us to encode cycle freeness more efficiently by only disallowing cycles of length three. Chordality has been used previously to encode equality logic in SAT using undirected graphs [Bryant and Velev, 2002]; to our knowledge this is the first application of chordality to express cycle-freeness of directed graphs.

Apart from the fact that this translation into the SAT problem helps in efficiently (in practice) solving the scheduling problem, there also is another benefit: it is now possible to encode extra constraints on the resulting schedule. We show two scheduling optimisations that are interesting from an attribute grammar point of view, for which the optimal schedule can be found efficiently by expressing the optimisation in the SAT problem.

1.2 Overview

In this paper we make the following main contributions:

- We show how to encode the problem of scheduling linearly ordered attribute grammars as a SAT problem
- We show that chordal graphs can be used to encode cycle-freeness in SAT problems
- We show how certain attribute grammar optimisations can be encoded as part of the formulation of the SAT problem

Furthermore, we have implemented the described techniques in the UUAGC[2] [Swierstra et al., 1999] and show that the technique works well in practice for the UHC.

The outline of the paper is as follows. We start in Section 2 and Section 3 by explaining linearly ordered attribute grammars and the difficulties in scheduling in more detail. We then describe the translation into the SAT problem in Section 4 and show the effectiveness of our approach in Section 5. In Section 6 we describe the two attribute grammar optimisations and finally in Section 7 we discuss some problems and conclude.

2 Attribute Grammars

An attribute grammar consists of three parts: a context-free grammar, a set of attribute definitions and a set of semantic functions. Instead of a context-free grammar describing the concrete syntax of the language, we feel that it is more intuitive to visualise the grammar as describing the abstract syntax tree of the language. We therefore say that an attribute grammar consists of a set of algebraic data types describing the abstract syntax, a set of attribute definitions and a set of semantic functions. The attribute definitions and semantic functions describe how values should be computed over the abstract syntax tree.

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

In the rest of this paper we stick to the usual attribute grammar terminology, so with a *nonterminal* we mean a type, and a *production* is a constructor of that type. Furthermore, the name *lhs* (short for left-hand side) refers to the constructor itself, while the children of a production have separate names. As this paper is about scheduling and data dependencies, we do not explain the syntax of the semantic functions here, but it is important to remark that these functions define how the value of a certain attribute can be computed from other attributes.

There are two types of attributes: *inherited* attributes with values that are passed from a parent to its children, and *synthesized* attributes with values that are passed from the children to their parents. In the pictures we draw inherited attributes on the left side of a node, and the synthesized attributes on its right side.

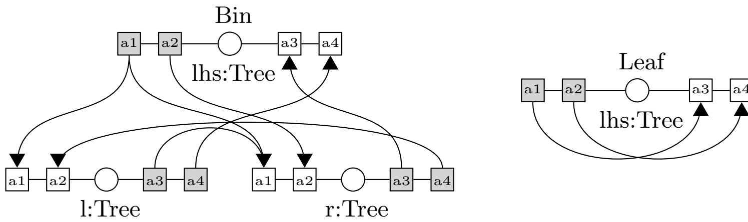

Fig. 1. Example attribute grammar for binary trees with two inherited and two synthesized attributes

In Figure 1 we show an example attribute grammar with one nonterminal *Tree* which has two productions: *Bin* and *Leaf*. The *Bin* production has two children named *l* and *r*, both of the type *Tree*, and *Leaf* has no children. The *Tree* nonterminal has four attributes: two inherited attributes a1 and a2 and two synthesized attributes a3 and a4.

The figure shows *production dependency graphs*, one for every production of the grammar. A set of semantic function definitions are given by the programmer for every production. The definitions specify how the synthesized attributes of a nonterminal are calculated in each of the different productions of that nonterminal. In semantic function definitions the inherited attributes of the nonterminal can be used as well as the synthesized attributes of its children. The attributes that are available to use, coloured gray in the picture, we call the *input* attributes. Besides the semantic functions of the synthesized attributes of the parent node, the semantic functions of the inherited attributes of the children need to be defined. The attributes that require a definition, coloured white in the picture, we call the *output* attributes. Note that with a semantic function definition for all output attributes we know how to compute all attributes as every input attribute is an output attribute of another node[3].

[3] Except for the inherited attributes of the root node. Its values need to be given as an argument to the semantic evaluator.

Although we talk about dependency graphs in the context of scheduling, we actually draw the edges in the other direction. The edges in the picture represent data flow, which from an attribute grammar perspective is much more intuitive. For example, in the *Bin* production the attribute a1 of the child r is computed from a1 from the parent and a3 from the child l. The edges are thus always directed from an input attribute to an output attribute, and the actual dependency graph can be obtained by reversing all edges. The edges that follow directly from the source code are the *direct dependencies*.

3 Linearly Ordered Attribute Grammars

Given an attribute grammar definition the attribute grammar compiler generates a runtime evaluator that takes an abstract syntax tree as input and computes all attribute values. There are two approaches for doing the scheduling: at runtime (dynamic) or at compile time (static).

For dynamic scheduling one could rely on existing machinery for lazy evaluation, for example in Haskell, which is the approach taken by [Saraiva, 1999]. There the attribute grammar definitions are translated into lazy Haskell functions in a straightforward way by producing functions that take the inherited attributes as argument and return the synthesized attributes. Whenever an inherited attribute (indirectly) depends on the value of a synthesized attribute, this means the function has a cyclic definition, which is no problem for languages with lazy semantics, and can actually result in efficient code [Bird, 1984].

There are two problems with dynamic scheduling. First, whenever the attribute grammar contains a true cyclic definition, the evaluator enters an infinite loop at runtime. However, these cycles could have been detected at compile time! Furthermore, the code needs lazy functions while with the static scheduling strict evaluators can be generated, leading to more efficient code in practice.

For static scheduling efficient evaluators can be generated, but the problem is that static scheduling is hard. In [Kastens, 1980] a polynomial time algorithm is given for static scheduling of the class of ordered attribute grammars, but unfortunately we often encounter attribute grammars outside of that class. All our attribute grammars do however fall in the class of linearly ordered attribute grammars, which is the largest class of attribute grammars for which a static schedule can be found. Although the scheduling is NP-complete, we have implemented a backtracking algorithm in earlier work [Van Binsbergen et al., 2015] that is feasible in practice. However, in this paper we show that we can do better by creating an algorithm that is faster and allows for different optimisations on the resulting schedule.

Conceptually, static scheduling of linearly ordered attribute grammars is not very complex. For each nonterminal a total order should be found on its attributes, such that all production dependency graphs all cycle free. However, because a nonterminal can have multiple productions and a production can have children of different types, choices made on the order of attributes of one nonterminal can influence the order of attributes in another nonterminal.

In order to encode this we also construct a *nonterminal dependency graph*. This graph contains all attributes that are defined for the corresponding nonterminal and the edges in this graph must define a total order on the attributes. Furthermore, the edges in the nonterminal dependency graph should agree with the (indirect) dependencies from the production dependency graphs such that no cycles exist.

The nonterminal dependency graphs and production dependency graphs are consistent if 1) whenever there is a path from attribute a to b in a production dependency graph then there needs to be an edge from a to b in the nonterminal dependency graph in which a and b occur[4] and 2) if there is an edge from a to b in a nonterminal dependency graph then there needs to be an edge from a to b for all occurrences of a and b in the production dependency graphs.

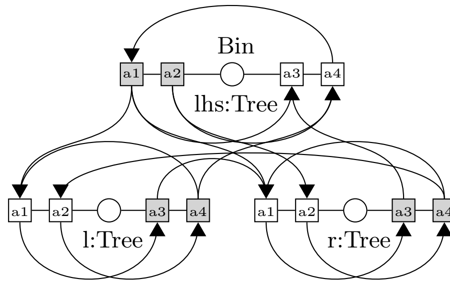

Fig. 2. Production dependency graph for Bin for the order a2 → a4 → a1 → a3

Figure 2 shows the production dependency graph of the production *Bin* with a complete order on the attributes. In this case there is only a single nonterminal so both the parent and the child nodes have the same attribute order, made explicit by extra edges from the nonterminal dependency graph. Because this dependency graph is still cycle free after adding the edges for the complete order and the same holds for the *Leaf* production, the order a2 → a4 → a1 → a3 is a valid schedule for the nonterminal *Tree*.

To find the total order we add edges to the nonterminal dependency graph and rely on the SAT-solver to determine their direction. Instead of adding an edge for all possible pairs of attributes it is sufficient to add an edge for all pairs of inherited and synthesized attributes. The order described by an assignment to these variables is not (always) total as there might be pairs of attributes, which attributes are either both inherited or both synthesized, without a relative ordering. This is not a problem as the evaluation order for such pairs can be chosen arbitrarily.

[4] There is no such nonterminal dependency graph if the two attributes belong to different nodes.

4 Translation into SAT

To represent the scheduling problem as a Boolean formula we introduce a variable for each edge, indicating the direction of the edge. The direct dependencies coming from the source code are constants, but for the rest of the edges the SAT-solver can decide on the direction. However, the encoding has been chosen in such way that a valid assignment of the variables corresponds to a valid schedule.

Our algorithm has the following steps:

1. Construct a nonterminal dependency graph for each nonterminal and add an edge between all pairs of inherited and synthesized attributes
2. Construct a production dependency graph for each production and add an edge for every direct dependency
3. Make all graphs chordal
4. Introduce a SAT variable for each edge in any of the graphs including the chords added in step 3. Variables of edges between attributes of the same nonterminal must be shared between nonterminal dependency graphs and production dependency graphs
5. Set the value of all variables corresponding to direct dependencies
6. Exclude all cycles of length three by adding constraints
7. Optionally add extra constraints for optimisations

The first two steps have been explained in the previous sections. Step 3 is explained below, step 4 is trivial, and step 5 and 6 follow from the explanation below. Finally step 7 is explained in Section 6.

4.1 Chordal Graphs

A chordal graph is an undirected graph in which each cycle of length > 3 contains a *chord*. A chord is an edge between two nodes in the cycle that are not adjacent. As a consequence each cycle of length > 3 can be split up into two smaller cycles, so if a chordal graph contains a cycle it must also contain a cycle of size three. Chordal graphs are therefore sometimes also referred to as *triangulated graphs*.

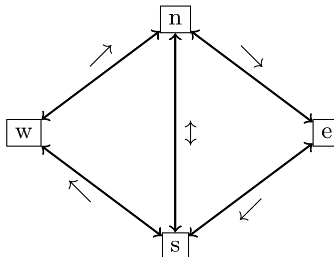

Fig. 3. There can not exists a cycle w-n-e-s without either a cycle n-s-w or n-e-s

In our case the graphs are directed, but we can still apply the same trick! In Figure 3 we illustrate how to use chordal graphs to exclude cycles. If there exists a cycle of length four, then it is not possible to choose a direction for the edge between n and s without introducing a cycle of length three. Hence, if we make our graph chordal by adding edges which may have an arbitrary direction and explicitly exclude all cycles of length three, we ensure that no cycles can exist at all.

4.2 Chordal Graph Construction

There are several algorithms for making a graph chordal. We use an algorithm based on the following alternative definition of a chordal graph:

Definition 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 $G[v_1, \ldots, v_i]$, $\forall(i)$ $(1 \leq i \leq n)$, 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 neighbourhood of v are all vertices w such that there exists an edge between v and w. 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.*

From this definition we can construct the following algorithm for making a graph chordal:

1. While the graph still contains vertices:
   (a) Select a vertex v from the graph
   (b) For every pair (a, b) of unconnected vertices in the neighbourhood of v:
       i. Add the edge $(a \leftrightarrow b)$ to the graph
   (c) Remove v, and all edges connected to v, from the graph.

One important open question in this algorithm is the order in which the vertices should be chosen. In Section 5.1 we show the results for several heuristics that we have implemented and tried on the UHC. We would like to remark that regardless of the heuristic used, this approach always leads to much smaller SAT problems than encoding transitivity in the SAT problem for ruling out cycles.

4.3 Finding the Schedule

When the constructed Boolean formula is given to the SAT-solver, the result is either that the formula is not satisfiable, meaning that no schedule can been found, or satisfiable, meaning that there is a schedule. It is not hard to see that the formula is satisfiable if and only if there exists a valid schedule for the given attribute grammar definition, and in this paper we give no formal proof of this claim.

In the case where the formula is satisfiable, we obviously want to find the result. From the SAT solver we can ask for the truth value of each variable in

the solution, so when our algorithm keeps the connection between edges and variables we can complete our directed graph and trivially find the complete order for all attributes from that. The constraints guarantee that this graph contains no cycles.

4.4 Shared Edges

One important implementation detail is that of shared edges. As explained, the nonterminal dependency graphs and the production dependency graphs share the edges that define the order of the attributes. Because each edge is represented by a variable in the SAT problem we can simply encode this by assigning the same variable to the shared edges.

However, as we also make both graphs chordal, the edges added to make the graphs chordal can also be shared. This is exactly what our implementation does, such that the SAT problem is kept as small as possible. The implementation is therefore slightly more complicated than explained in the previous sections.

5 Empirical Results

We use the Utrecht Haskell Compiler (UHC) [Dijkstra et al., 2009] as the main test case for our work. The source code of the UHC consists of several attribute grammar definitions together with Haskell code. The biggest attribute grammar in the UHC, called *MainAG*, consists of 30 nonterminals, 134 productions, 1332 attributes (44.4 per nonterminal) and 9766 dependencies and is the biggest attribute grammar we know of.

We have compiled several attribute grammars using the Utrecht University Attribute Grammar Compiler (UUAGC) [Swierstra et al., 1999], in which we have implemented our approach. Apart from our SAT approach there are three other scheduling algorithms that have been implemented to which we can compare our approach:

- *Kastens*: the algorithm from [Kastens, 1980] which only works for ordered attribute grammars, of which most examples are not a member.
- *Kennedy-Warren*: The algorithm from [Kennedy and Warren, 1976] that we have implemented in the UUAGC before [Bransen et al., 2012] is an algorithm that schedules all absolutely noncircular attribute grammars, an even larger class than the linearly ordered attribute grammars. However, the scheduling is not completely static so the generated evaluator also does part of the scheduling at runtime.
- *LOAG-backtrack*: we have also implemented a backtracking algorithm for linearly ordered attribute grammars [Van Binsbergen et al., 2015], based on Kastens' algorithm. This algorithm solves exactly the same scheduling problem as the SAT approach described in this paper and uses exponential time in worst case. In case of the *MainAG* no backtracking is required to find a schedule.

Table 1. Comparison of the four scheduling algorithms

Algorithm	Kastens'	Kennedy-Warren	LOAG-backtrack	LOAG-SAT
UHC MainAG	-	33s	13s	9s
Asil Test	-	1.8s	4.4s	3.4s
Asil ByteCode	-	0.6s	29.4s	2.8s
Asil PrettyTree	-	390ms	536ms	585ms
Asil InsertLabels	-	314ms	440ms	452ms
UUAGC CodeGeneration	-	348ms	580ms	382ms
Pigeonhole principle	-	107ms	1970ms	191ms
Helium TS_Analyse	190ms	226ms	235ms	278ms

In Table 1 we show the compilation times for the examples for the four different algorithms. All times include parsing of the attribute grammar description, code generation and adding chords, which is what takes most time in the SAT approach. The SAT-solver takes less than a second to find a solution in all cases.

The other test cases we have used for testing are the UUAGC itself, *Asil* [Middelkoop et al., 2012] which is a byte code instrumenter, the *Helium* compiler [Heeren et al., 2003] which is a Haskell compiler specifically intended for students learning Haskell, and an encoding of the Pigeonhole principle. We removed one clause in such way there exists exactly one valid schedule, resulting in an artificial attribute grammar that is hard to schedule.

5.1 Chordal Graph Heuristics

As explained in Section 4.2 we need to find an order in which to handle the vertices such that the resulting SAT problem is as small as possible. In Table 2 we show the results of different heuristics for the MainAG of the UHC. In this table we use three different sets: $\mathcal{D}$ is the set of direct dependencies (step 2, Section 4), $\mathcal{C}$ is the set of edges that are added to make the graph chordal (step 3, Section 4) and $\mathcal{S}$ is the set of edge between all inherited and synthesized pairs (step 1, Section 4). For each of the sets we take only the edges in the neighbourhood of v for comparison.

6 Optimisations

We have shown that expressing the scheduling problem as a SAT problem and using an existing SAT-solver can improve the running time of the scheduling, but that is not the only advantage. In the SAT problem one can easily add extra constraints to further influence the resulting schedule. In Section 6.2 and Section 6.3 we show two of such optimisations that are useful from an attribute grammar perspective. These optimisations have not been implemented in the release version of the UUAGC, but we have run preliminary experiments to verify that they work as expected.

Table 2. Table showing the number of clauses and variables required for solving the MainAG of the UHC, selecting the next vertex in the elimination order based on different ways to compare neighbourhoods

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

6.1 Interacting with the Solver

Instead of directly expressing all constraints in the initial SAT problem, we use a different trick for implementing the two optimisations: interacting with the solver. After the initial scheduling problem has been solved, we can ask for the truth value of all variables to construct the schedule. MiniSat also keeps some state in memory that allows us to add extra constraints to the problem, and ask for a new solution. In this way we can start with an initial solution and interact with the solver until some optimum has been reached.

6.2 Minimising Visits

The result of the static scheduling is a runtime evaluator that computes the values of the attributes for a given abstract syntax tree. The total order for each nonterminal defines in what order attributes should be computed, but in the implementation of the evaluator we make use of a slightly bigger unit of computation: a *visit*.

A visit is an evaluation step in the runtime evaluator that takes the values of a (possibly empty) set of inherited attributes and produces a (non-empty) set of synthesized attributes. In order to compute these values, visits to the child nodes may happen, and at the top of the tree the wrapping code invokes all visits of the top node one by one.

Because invoking a visit at runtime may have a certain overhead, we would like the number of visits to be as small as possible. In other words, in the total order on the attributes we would like to minimise the number of places where a

synthesized attribute is followed by an inherited attribute, because that is the location where a new visit needs to be performed.

It is theoretically impossible to minimise the total number of visits performed for the full abstract syntax tree, because at compile-time we do not have a concrete abstract syntax tree at hand and only know about the grammar. We therefore try to minimise the maximum number of visits for any nonterminal, which is the number of alternating pairs of inherited and synthesized attributes in the total order.

In our algorithm, we use efficient counting constraints, expressed in the SAT solver using sorting networks [Swierstra et al., 1999]. This enables us to count the number of true literals in a given set of literals, and express constraints about this number. A standard procedure for finding a solution for which the minimal number of literals in such a set is true can be implemented on top of a SAT-solver using a simple loop.

We use the following algorithm for minimising the maximal number of visits:

1. Construct the initial SAT problem and solve
2. Construct the set of all production rules P
3. Construct counting networks that count the number of visits $V(p)$ for all production rules p in P
4. Count the number of visits for each production rule in the current solution; let M be the maximum value
5. Repeat while $M > 0$:
   (a) Add constraints that express that for all productions p in P: $V(p) \leq M$
   (b) Construct a counting network that counts how many production rules p have $V(p) = M$
   (c) Compute a solution for which this number is minimised using the loop described above
   (d) Remove all p in P for which now $V(p) = M$ from the set P
   (e) Compute the new maximum value M of $V(p)$ for all p left in P

The above algorithm features a complicated combination of counting networks; one network for each production rule, and one network for each corresponding output of these networks. Still, the procedure finds optimal solutions very quickly in practice, in times that are negligible and not practically measurable compared to the time of generating the initial SAT-problem. The number of iterations for the minimisation loops has never been more than 5 in any of our problems.

The algorithm is guaranteed to find the global optimum. For our largest example, the solution found had a total of 130 visits, which was 29 visits less in total than the previously known optimum, found using backtracking heuristics.

One could criticise the usefulness of this particular optimisation for attribute grammars. Indeed, details in how one should optimise the number of visits depend very much on the kind of trees we are going to run the compiled code on. Our point is that we can easily express variants of optimisations. For example, we can also minimise the sum of all visits using a similar (but simpler) procedure to the one above. Again, the running time of that procedure is very short.

6.3 Eager Attributes

Another optimisation is the ability to define *eager attributes*. Eager attributes are attributes that should be computed as soon as possible, and must be annotated by the attribute grammar programmer as such. We would like our scheduling algorithm then to schedule them as early as possible in the total order.

As an example, in a typical compiler there is an attribute containing the errors that occur in compilation. When running the compiler one is typically first interested in knowing if there are any errors; if so they must be printed to the screen and the compiler can stop its compilation. If there are no errors, then all other work that is not strictly necessary for the generation of errors can be done to complete the compilation.

In order to schedule a given attribute as early as possible, we are going to partition all attributes contained in the grammar into two sets E (for early) and L (for late). The idea is that E contains all attributes that may be needed to be computed before the eager attribute (i.e. there exist production rules which require this), and L contains all attributes that we can definitely compute after knowing the eager attribute (i.e. no production rule requires any attribute in L for computing the eager attribute). We want to find a schedule for which the size of E is minimal.

To compute this, we introduce a SAT variable $E(a)$ for every attribute a, that expresses whether or not a is in E or not. We set $E(a)$ to be true for the initial eager attribute a. We go over the graphs for the nonterminals and production rules, and generates constraints that express that whenever a points to b and we have $E(b)$, then we also need $E(a)$.

Finally, we use a counting network for all literals $E(a)$, and ask for a solution that minimises the number of literals in E.

We have run this algorithm on every output attribute of the top-level nonterminal of all our examples. For our largest grammar, the hardest output to compute took 1 second. So, while a harder optimisation than the previous one, it is very doable in practice.

One can imagine other variants of this optimisation, where we have a set of eager attributes, or a combination of eager and late attributes. At the time of the writing of this paper, we have not experimented with such variants yet.

7 Discussion and Conclusion

We have explained the difficulties in attribute grammar scheduling and shown how to solve this problem using a SAT-solver. The given approach has been implemented and tested on a full-scale compiler built using attribute grammars. Results show that the algorithm works faster than other approaches.

In the translation into the SAT-problem we have used a novel technique for ruling out cycles in the SAT problem using chordal graphs. Existing work on chordality for expressing equality logic [Bryant and Velev, 2002] was the inspiration for this technique. Using chordal graphs makes the problems much smaller

than directly ruling out cycles, while encoding the same restrictions. We believe that this technique is applicable in other problems using SAT-solvers as well.

Furthermore, we have shown that expressing the problem as a SAT problem has the advantage that extra constraints can be added. We illustrated this with two possible properties of the resulting schedule that an attribute grammar programmer may want to influence. Even though this makes the scheduling problem potentially harder, as the algorithm is left fewer choices, the solution is found very fast in all practical cases we have tried.

Another benefit of this approach that the attribute grammar scheduling can benefit from breakthroughs in the SAT community. The more efficient SAT solvers become, the better the attribute grammar scheduling becomes leading to larger and larger attribute grammars that are feasible to schedule.

One problem with the current approach in contrast to most other scheduling algorithms is the unpredictability. SAT-solvers use certain heuristics to quickly find solutions in many applications, but it can theoretically happen that for a certain attribute grammar the SAT problem that is generated is not suitable for these heuristics. A seemingly innocent change in the attribute grammar definition could therefore theoretically lead to a large increase in compile time. However, we have not encountered this problem and we believe that this situation is unlikely to happen because of the maturity of the SAT-solvers.

All in all, we believe that this approach fully solves the basic scheduling problem in an elegant way. There are ample possibilities for improving the resulting schedules based on attribute grammar knowledge like the two discussed in Section 6, so we have also made room for future improvements in the scheduling and compilation of attribute grammars.

References

[van Binsbergen et al., 2015] van Binsbergen, L.T., Bransen, J., Dijkstra, A.: Linearly ordered attribute grammars: With automatic augmenting dependency selection. In: Proceedings of the 2015 Workshop on Partial Evaluation and Program Manipulation, PEPM 2015, pp. 49–60. ACM, New York (2015)

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

[Bransen et al., 2012] Bransen, J., Middelkoop, A., Dijkstra, A., Swierstra, S.D.: The Kennedy-Warren algorithm revisited: Ordering attribute grammars. In: Russo, C., Zhou, N.-F. (eds.) PADL 2012. LNCS, vol. 7149, pp. 183–197. Springer, Heidelberg (2012)

[Bryant and Velev, 2002] Bryant, R.E., Velev, M.N.: Boolean satisfiability with transitivity constraints. ACM Trans. Comput. Logic 3(4), 604–627 (2002)

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

[Swierstra et al., 1999] Codish, M., Zazon-Ivry, M.: Pairwise cardinality networks. In: Clarke, E.M., Voronkov, A. (eds.) LPAR-16 2010. LNCS, vol. 6355, pp. 154–172. Springer, Heidelberg (2010)

[Cook, 1971] Cook, S.A.: The complexity of theorem-proving procedures. In: Proceedings of the Third Annual ACM Symposium on Theory of Computing, STOC 1971, pp. 151–158. ACM, New York (1971)

[Dijkstra et al., 2009] Dijkstra, A., Fokker, J., Swierstra, S.D.: The architecture of the Utrecht Haskell Compiler. In: Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell, Haskell 2009, pp. 93–104. ACM, New York (2009)

[Dirac, 1961] Dirac, G.A.: On rigid circuit graphs. Abh. Math. Sem. Univ. Hamburg 25, 71–76 (1961)

[Eén and Sörensson, 2004] Eén, N., Sörensson, N.: An extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, pp. 502–518. Springer, Heidelberg (2004)

[Ekman and Hedin, 2007] Ekman, T., Hedin, G.: The JastAdd extensible Java compiler. In: Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications, OOPSLA 2007, pp. 1–18. ACM, New York (2007)

[Engelfriet and Filè, 1982] Engelfriet, J., Filè, G.: Simple multi-visit attribute grammars. Journal of Computer and System Sciences 24(3), 283–314 (1982)

[Heeren et al., 2003] Heeren, B., Leijen, D., van IJzendoorn, A.: Helium, for learning haskell. In: Proceedings of the 2003 ACM SIGPLAN Workshop on Haskell, Haskell 2003, pp. 62–71. ACM, New York (2003)

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

[Kennedy and Warren, 1976] Kennedy, K., Warren, S.K.: Automatic generation of efficient evaluators for attribute grammars. In: Proceedings of the 3rd ACM SIGACT-SIGPLAN Symposium on Principles on Programming Languages, POPL 1976, pp. 32–49. ACM, New York (1976)

[Knuth, 1968] Knuth, D.E.: Semantics of context-free languages. Mathematical Systems Theory 2(2), 127–145 (1968)

[Middelkoop et al., 2012] Middelkoop, A., Elyasov, A.B., Prasetya, W.: Functional instrumentation of actionscript programs with asil. In: Gill, A., Hage, J. (eds.) IFL 2011. LNCS, vol. 7257, pp. 1–16. Springer, Heidelberg (2012)

[Saraiva, 1999] Saraiva, J.: Purely Functional Implementation of Attribute Grammars: Zuiver Functionele Implementatie Van Attributengrammatica's. IPA dissertation series. IPA (1999)

[Swierstra et al., 1999] Swierstra, S.D., Alcocer, P.R.A.: Designing and implementing combinator languages. In: Swierstra, S.D., Oliveira, J.N. (eds.) AFP 1998. LNCS, vol. 1608, pp. 150–206. Springer, Heidelberg (1999)