From Attribute Grammars to Constraint Handling Rules

Alejandro Serrano

Jurriaan Hage

Technical Report UU-CS-2016-010
November 2016

Department of Information and Computing Sciences
Utrecht University, Utrecht, The Netherlands
www.cs.uu.nl

ISSN: 0924-3275

Department of Information and Computing Sciences
Utrecht University
P.O. Box 80.089
3508 TB Utrecht
The Netherlands

From Attribute Grammars to Constraint Handling Rules

Alejandro Serrano (✉) and Jurriaan Hage*

Department of Information and Computing Sciences
Utrecht University, The Netherlands
{A.SerranoMena, J.Hage}@uu.nl

Abstract. Attribute grammars provide a framework to define computations over trees, by decorating those trees with attributes. Attribute grammars have been successfully applied in many areas, including compiler construction and natural language processing. In this paper we present a translation of attribute grammars to Constraint Handling Rules, a formalism based on constraint rewriting.
Our translation is able to express in a simple way several extensions to attribute grammars. Higher-order attributes are attributes whose value is again a tree, for which attributes can be computed recursively. Lookahead enables attribute definitions to depend not only on the current node, but also on the shape of its subtrees. Specialization provides a way to override the default computation of an attribute when some conditions are met; a natural way to define exceptions to the default tree processing.

Keywords: Attribute grammars; Constraint Handling Rules; Higher-order attribute grammars

1 Introduction

Attribute Grammars (AGs from now on) provide a formalism to define computations over trees [12]. Each node in a tree is augmented with a set of *attributes*, the AG describes how the value of each attribute is computed from those of the parent or the children of a given node. AGs are very useful for implementing compilers [6], since the operations happen naturally over the Abstract Syntax Tree of the program. Among their advantages, AGs are very easy to combine and relieve the programmer from thinking about the order in which a tree ought to be traversed to compute all the attributes.

As a first example of attribute grammar, let us define one to compute the minimum value in a binary tree. The first step is describing the shape of the trees. Throughout this paper we use the syntax from UUAGC, the Attribute Grammar System of Universiteit Utrecht [19], which is very close to Haskell's.

* This work was supported by the Netherlands Organisation for Scientific Research (NWO) project on "DOMain Specific Type Error Diagnosis (DOMSTED)" (612.001.213).

$$\begin{aligned}&\textbf{data}\ \mathit{Tree} \mid \mathit{Node}\ \mathit{left}, \mathit{right} :: \mathit{Tree}\\ &\qquad\quad \mid \mathit{Leaf}\ \ \mathit{value} :: \mathit{Int}\end{aligned}$$

The second step is defining the attributes to be found in each node. In this case we have only one, holding the minimum value of a tree. The information for this attribute flows bottom-up; we call this kind of attributes *synthesized*. Attributes flowing in the other direction are called *inherited*.

$$\begin{aligned}&\textbf{attr}\ \mathit{Tree}\\ &\quad \textbf{syn}\ \mathit{min} :: \mathit{Int}\end{aligned}$$

Finally, we must provide *semantic functions* which explain how to derive the value of *min* for each kind of node. In UUAGC we use the syntax @*field* to refer to the fields from the **data** declaration, and @*tree*.*attribute* to speak about the attribute in the stated tree. The name **lhs** is used to refer to the current node for which the attribute is being computed.

$$\begin{aligned}&\textbf{sem}\ \mathit{Tree}\\ &\quad \mid \mathit{Node}\ \textbf{lhs}.\mathit{min} = \{\, \mathit{min}\ @\mathit{left}.\mathit{min}\ @\mathit{right}.\mathit{min} \,\}\\ &\quad \mid \mathit{Leaf}\ \ \textbf{lhs}.\mathit{min} = @\mathit{value}\end{aligned}$$

The other character in our play is the formalism known as *Constraint Handling Rules* [8]. The operation of a set of CHRs is based on constraint rewriting. Originally devised as a way to describe constraint solvers, CHRs are now used for a wide variety of scenarios [17]. In this paper we focus on how an attribute grammar can be implemented as a set of CHRs (§ 3) and the benefits that arise.

A key point in our translation is the representation of a tree as a set of constraints. We use a different kind of constraint per constructor in our trees. That constraints holds a unique *identifier* which identifies the node, plus the information in each field. Those fields which contain a subtree reference its identifier. For example, the binary tree *Node* (*Node* (*Leaf* 1) (*Leaf* 2)) (*Leaf* 3) is represented as the following set of constraints:

$$\mathsf{node}\,(\alpha, \beta, \gamma), \mathsf{node}\,(\beta, \delta, \epsilon), \mathsf{leaf}\,(\delta, 1), \mathsf{leaf}\,(\epsilon, 2), \mathsf{leaf}\,(\gamma, 3)$$

where α, β, ... are distinct identifiers.

Attributes in a node are represented in a similar fashion: by a constraint which holds the identifier of the node and the value attached to that node. Semantic functions are turned into rules manipulating those constraints. The functions corresponding to the computation of the *min* attribute are translated into:

$$\begin{aligned}&\mathsf{node}\,(\mathit{Id}, L, R), \mathsf{min}\,(L, N), \mathsf{min}\,(R, M) &&\Longrightarrow\ X = \mathit{min}\ N\ M \mid \mathsf{min}\,(\mathit{Id}, X)\\ &\mathsf{leaf}\,(\mathit{Id}, N) &&\Longrightarrow\ \mathsf{min}\,(\mathit{Id}, N)\end{aligned}$$

This attribute grammar is fairly simple, though: it computes just a single synthesized attribute. However, when we have inherited and synthesized attributes which depend of each other, we need to *schedule* the computations to ensure that at each point of execution all the necessary information is available. Kennedy

and Warren [11] and Kastens [10] provide algorithms to statically compute such a scheduling. In § 3.2 we give a different solution to the problem by encoding the dependencies between attributes also as constraints.

Our representation of AGs as CHRs inherit the main benefits of the former, namely, the modularity and ease of composition. In addition, we can implement some extensions to AGs in a straightforward way:

- In Higher-order Attribute Grammars [21] we can give an attribute a value which is itself a tree, and then ask for computation of attributes over that tree. HOAGs can be used to represent different phases of a compiler, or to perform iterative computations. In § 4 we extend our translation to CHRs to account for higher-order attributes.
- In standard AGs only the constructor of a node is inspected to decide which semantic function to use. This is not enough in some cases: for example, when using specialized type rules [16] we override the default typing algorithm for some specific expressions. In § 5 we describe how special cases where look-ahead is needed can be handled by our translation.

2 A quick recap of Constraint Handling Rules

Before we dive in the central topic of this paper, let us refresh some notions about Constraint Handling Rules. The language of CHRs has three kinds of rules:

$$\begin{array}{rcl} H^r & \iff & G \mid B \quad \text{simplification} \\ H^k & \implies & G \mid B \quad \text{propagation} \\ H^k \setminus H^r & \iff & G \mid B \quad \text{simpagation} \end{array}$$

In each case, H^k, H^r and B are sets of constraints, called the heads and the body respectively. We use $\top$ to represent an empty set of constraints. In order for a rule to be applied, some constraints from the current set must match the heads, and the guard G must be satisfied. Rewriting depends on the kind of rule: with simplification rules the constraints H^r are replaced by B, in propagation rules the constraints B are added to the set but H^k are kept. Simpagation rules are a generalization of both: H^k constraints are kept and H^r are removed. In fact, we can see any CHR as a simpagation rule where the heads might be empty.

CHRs are applied non-deterministically. For a given initial constraint set, many different sequences of applications of rules are usually possible. Confluence, that is, the fact that the outcome of the process does not depend on the other in which rules are applied, must be guaranteed by the author of the CHRs.

The execution trace for the example in the introduction starts by (non-deteministically) generating $\mathsf{min}\,(\delta, 1)$, $\mathsf{min}\,(\epsilon, 2)$ and $\mathsf{min}\,(\gamma, 3)$ from the leaves. Once we have the values for δ and ϵ, the rule for the *Node* case applies and the system generates a new constraint $\mathsf{min}\,(\beta, 1)$. Eventually the same rule applies for the node with β and γ as their children, obtaining a final $\mathsf{min}\,(\alpha, 1)$ constraint.

2.1 Extensions to CHRs

The standard language of CHRs is enough to describe AGs, with or without higher-order attributes, but without special cases. For the latter, we make use in § 5 of two extensions to CHRs.

In their original formulation, when more than one rule matches a set of constraints, the CHR engine is free to choose which one to apply. Once this decision is made, though, everything is settled: it is not possible to explore the implications of using the other rule instead. This mode of operation is called *commited choice*. In some cases we prefer the engine to try the different options, performing backtracking if one fails, in a similar fashion to Prolog. CHR$^\vee$ [1] provides one such *choice* operator. Rules now take the general form:

$$H^k \setminus H^r \iff G \mid B_1 \,;\, \cdots \,;\, B_n$$

In this setting, we must have some way to indicate the engine that a given branch leads to failure, and thus another branch should be tried. In this case, it is done by means of a special constraint $\bot$.

The semantics of the choice operator gives preference to the left-most branch which succeeds. However, the order of exploration of the branches is not fixed. In some systems, this order can be programmed by the user of the CHRs [13].

A (certainly convoluted) way to compute the minimum value of a binary tree involves trying the two possibilities coming from the two subtrees, and then pruning those cases which are not consistent.

$$\begin{array}{lcl}
\mathsf{node}(P,L,R), \mathsf{min}(L,N), \mathsf{min}(R,M) & \Longrightarrow & \mathsf{min}(P,N) \;;\; \mathsf{min}(P,M) \\
\mathsf{node}(P,L,R), \mathsf{min}(P,N), \mathsf{min}(L,M) & \Longrightarrow & N > M \mid \bot \\
\mathsf{node}(P,L,R), \mathsf{min}(P,N), \mathsf{min}(R,M) & \Longrightarrow & N > M \mid \bot \\
\mathsf{leaf}(L,N) & \Longrightarrow & \mathsf{min}(L,N)
\end{array}$$

Even when the set of constraint matches is not equal, more than one rule may be available for application at a certain moment. Once again, the standard semantics of CHRs is non-deterministic. *Rule priorities* [4] add the ability to prefer application of a rule between others; in case of having more than one rule than might apply, the one with highest priority is chosen.

This ability is important to have good complexity and performance in some cases, and even for correctness in others. The archetypical example of the latter case is the following implementation of Dijkstra's shortest-path algorithm [4]:

$$\begin{array}{lcll}
\mathsf{source}(A) & \Longrightarrow & \mathsf{dist}(A,0) & \textbf{priority}\ \ 1 \\
\mathsf{dist}(A,D1) \setminus \mathsf{dist}(A,D2) & \Longleftrightarrow & D1 < D2 \mid \top & \textbf{priority}\ \ 1 \\
\mathsf{dist}(A,D), \mathsf{edge}(A,B,W) & \Longrightarrow & \mathsf{dist}(B, D+W) & \textbf{priority}\ \ D+2
\end{array}$$

For a correct computation of all shortest paths, we must explore vertices with a known smaller path to them before those with a larger distance. The **priority** declaration in the third rule ensures that this is the case. Note that rule priority may depend on information in the matched constraints.

3 First-Order Attribute Grammars

In this section we present the general methodology for turning an attribute grammar into a set of CHRs. Initially we generate rules describing only the relation between attributes; then we concern ourselves with the problem of scheduling the order of computation of the attributes.

Definition 1 *A* (first-order) attribute grammar[1] *is a triple* $\langle G, A, D\rangle$ *where:*

- G *contains a set of* data type definitions*, representing non-terminals in a context-free grammar, and whose* constructors *represent the production rules.*
- A *is the set of* attributes *of the form* $(X \cdot a : \tau, d)$*, meaning that data type* X *holds an attribute* a *of type* τ*. Each attribute is also assigned a* direction d*, which can be either* inherited *or* synthesized*.*
- D *is the set of* semantic function definitions $X_{c,f} \cdot a = \lambda$*, where function* λ *defines the attribute* a *in the field* f *of the constructor* c *of data type* X*.*

Following the convention in UUAGC, we shall refer to the current node in a constructor c *as* $X_{c,\mathbf{lhs}}$*.*

Definition 2 *Given a constructor* c *in a data type* X*, we define its* input occurences $\mathcal{O}_{in}(X_c)$ *and its* output occurences $\mathcal{O}_{out}(X_c)$*.*

$$\begin{aligned}\mathcal{O}_{in}(X_c) &= \{X_{c,\mathbf{lhs}} \cdot a \mid X \cdot a \text{ is inherited}\}\\ &\quad \cup \{X_{p,f} \cdot a \mid X_{p,f} \text{ is a field of type } Y, Y \cdot a \text{ is synthesized}\}\\ \mathcal{O}_{out}(X_c) &= \{X_{c,\mathbf{lhs}} \cdot a \mid X \cdot a \text{ is synthesized}\}\\ &\quad \cup \{X_{p,f} \cdot a \mid X_{p,f} \text{ is a field of type } Y, Y \cdot a \text{ is inherited}\}\end{aligned}$$

Definition 3 (Normal forms)

- *An AG is written in* Bochmann Normal Form *[2] if the right-hand side of every semantic function uses only input occurences.*
- *An AG is* normalized *if is written in BNF and only output occurences have semantic functions.*
- *An AG is* complete *if there is a semantic function for every output occurence.*

From this moment on, we assume that all AGs are normalized, complete, and that semantic functions are well-typed.

Constraint gathering as an AG. Type inference algorithms based on constraints are common in the literature [22,18,15]. The operation of these algorithms is split in two phases: the first one traverses the AST of the program gathering constraints, and the second one simplifies and solves those constraints.

As a running example, we are going to define constraint gathering for a simple λ-caculus using an AG. Its syntax is given by the following declaration. Note that abstractions are annotated with the type of its argument.

[1] Our definition is very similar to the descriptions in UUAGC. For a definition closer to the area of context-free languages, see [12].

```
data Expr | Var  v :: TermVar
          | Abs  v :: (TermVar, Type), e :: Expr
          | App  e1, e2 :: Expr
```

The syntax of types contains variables, type constructors and function types:

```
data Type | TyVar  v :: TyVar
          | Constr c :: ConstrName, args :: [Type]
          | Fun    source, target :: Ty
```

While traversing the tree we keep track of an environment, a mapping of term variables to types, which flows top-down. As a result of gathering, at each node in the tree we synthesize the type and the constraints related to that subexpression.

```
attr Expr
  inh env :: [(TyVar, Type)]
  syn res :: (Type, [Constraint])
```

The last step is defining the semantic functions. The environment is only enlarged in an abstraction node.[2] Constraints are introduced in application nodes to related the shape of the function and its arguments. In the code we use a *freshTyVar* function which returns a fresh type variable each time it is called.

```
sem Expr
  | Var  lhs.res = (fromJust (lookup @v @lhs.env), [])
  | Abs  e.env   = (fst @v, snd @v) : @lhs.env
         lhs.res = (Fun (snd @v) (fst @e.res), snd @e.res)
  | App  e1.env = @lhs.env
         e2.env = @lhs.env
         lhs.res = (σ, (fst @e1.res ≡ Fun (fst @e2.res) σ)
                       : snd @e1.res ++ snd @e2.res) where σ = freshTyVar
```

3.1 Translation to CHRs

As hinted in the introduction, if we want to translate AGs to CHRs, we first need to model the trees themselves, which form the input to the decoration, as constraints which can be inspected by the CHR engine.

Since once a tree is flattened into a set of constraints we lose the contextual information about its position, we need to introduce an *identifier* for each node. The nature of such identifier is irrelevant: we only need to be able to compare different identifiers for equality. In order to decide which semantic function to apply, we need to distinguish which constructor (or production) has been used to build a node of the tree. For each constructor c in an AG, we introduce a corresponding constraint c. The arity of the constraint c is one plus the arity of the constructor c, to make room for the identifier.

[2] We assume that all term variables have been renamed to remain unique.

The procedure for flattening a tree works top-down. It assigns new identifiers to each node, and replaces subtrees in a node with their identifiers.

$$\begin{aligned}\mathsf{flatten}(c\ a_1\ \ldots\ a_n) &= \mathsf{c}(i, \hat{a}_1, \ldots, \hat{a}_n)\\ &\text{where } i = \text{fresh identifier}\\ \hat{a}_j &= \begin{cases}\text{identifier of } \mathsf{flatten}(a_j) & \text{if } a_j \text{ is a subtree}\\ a_j & \text{otherwise}\end{cases}\end{aligned}$$

The identifiers also serve to link each subtree with its attributes. For each attribute a in an AG, we define a corresponding constraint $\mathsf{a}(i, v)$, where the first argument i is the identifier of the tree the attribute is computed for, and the second v is the value assigned to the attribute.

We have described the shape of constraints in our CHR. It is now time for the rules: we get one per semantic function in the AG. Or given that our AGs are always normalized and complete, one per output attribute. Each of these semantic functions λ is given for an attribute a in the context of a constructor c. Furthermore, it may reference attributes b_1, …, b_m from some of the subtrees. In the CHR world, the dependency on other attributes and the way to compute the attribute defines a propagation rule:

$$\mathsf{c}(I, A_1, \ldots, A_n), \mathsf{b}_1(J_1, V_1), \ldots, \mathsf{b}_m(J_m, V_m) \implies X = \hat{\lambda} \mid \mathsf{a}(I, X)$$

where $\hat{\lambda}$ is a version of λ in which the references to information in the node and attributes from other subtrees have been replaced by references to the metavariables $A_1, \ldots, A_n$ and $V_1, \ldots, V_m$.

Constraint gathering as CHRs. Let us go back to our running example of typing for λ-calculus. As explained above, we need to introduce five kinds of constraints: $\mathsf{var}(I, V)$, $\mathsf{abs}(I, V, T, E)$ and $\mathsf{app}(I, E_1, E_2)$ correspond to tree nodes; $\mathsf{env}(I, E)$ and $\mathsf{res}(I, T, C)$ to attributes.[3]

The simplest semantic function is the computation of *res* for the *Var* constructor. We need to match two constraints in the corresponding rule: the var corresponding to the node itself, and its env attribute which is used in the lookup.

$$\mathsf{var}(I, V), \mathsf{env}(I, E) \implies T = \mathit{fromJust}\ (\mathit{lookup}\ V\ E) \mid \mathsf{res}(I, T, [\,])$$

The propagation of environments flows from parents to children:

$$\begin{aligned}\mathsf{abs}(I, V, T_V, B), \mathsf{env}(I, E) &\implies E' = (V, T_V) : E \mid \mathsf{env}(B, E')\\ \mathsf{app}(I, E_1, E_2), \mathsf{env}(I, E) &\implies \mathsf{env}(E_1, E)\\ \mathsf{app}(I, E_1, E_2), \mathsf{env}(I, E) &\implies \mathsf{env}(E_2, E)\end{aligned}$$

whereas the computation of types and constraints follows the converse direction:

$$\begin{aligned}\mathsf{abs}(I, V, T_V, B), \mathsf{res}(B, T_B, C_B) &\implies T = \mathit{Fun}\ T_V\ T_B \mid \mathsf{res}(I, T, C_B)\\ \mathsf{app}(I, E_1, E_2), \mathsf{res}(E_1, T_1, C_1), \mathsf{res}(E_2, T_2, C_2) &\implies \ldots \mid \mathsf{res}(I, T, C)\end{aligned}$$

For conciseness, we have omitted the definition of the T and C metavariables in the last rule. They can be easily recovered from the corresponding semantic function by applying the transformation described above.

[3] To increase readability, we have inlined pairs as two different arguments.

3.2 Attribute Scheduling

The translation we have described works by forward reasoning. At each point of the execution, all the additional constraints which could be derived from the already existing information are generated. One benefit of this working model is that dependencies are tracked in a very fine-grained way: to compute a certain attribute we only ask for the minimal information needed to derive it.

Definition 4 (Dependency graphs)

- *The dependency graph for a* constructor *c has as vertices all attribute occurences $X_{c,f} \cdot a$ and an edge between $X_{c,f_1} \cdot a_1$ and $X_{c,f_2} \cdot a_2$ if the semantic function for $X_{c,f_2} \cdot a_2$ mentions $X_{c,f_1} \cdot a_1$.*
- *The dependency graph for a* tree *$\mathcal{T}$ is obtained by pasting the dependency graphs for each node in $\mathcal{T}$.*

Definition 5 *We say that an AG is* well-defined *[12] if for every possible tree $\mathcal{T}$, the dependency graph for $\mathcal{T}$ contains no cycles.*

The translation we have described works only if the AG is indeed well-defined. Otherwise, at some point we might enter a loop or miss some information needed to move forward. Note however that the order in which attributes are computed might differ from tree to tree.

There are several disadvantages to forward reasoning, though. The first one is that we need a CHR engine with set-based semantics, that is, one which stops a rule from firing if the constraints to generate are already in the current set. Alas, most CHR implementations do not track duplicates. Since our translation only uses propagation rules, nothing stops it for running indefinitely.

The second problem is that we might end up doing more work than needed. If we are only interested in the value of a certain attribute a, we prefer to compute the least amount of attributes on the tree needed to obtain that information.

In order to solve both problems, we present a different translation based on backward reasoning about dependencies. For each attribute a, apart from the constraint a, we introduce two new constraints: $\mathsf{deps\_a}$ signals that we need to produce dependencies for a, and $\mathsf{needs\_a}$ that we are interested in computing a.

For each attribute a in a constructor c which references $b_1, \ldots, b_m$ from subtrees, we have a first rule which defines the dependencies needed for compute a. Once the dependencies are generated, the $\mathsf{deps\_a}$ is turned into $\mathsf{needs\_a}$.

$$\mathsf{c}(I, A_1, \ldots, A_n), \mathsf{deps\_a}(I) \iff \mathsf{needs\_a}(I), \mathsf{deps\_b}_1(J_1), \ldots, \mathsf{deps\_b}_m(J_m)$$

In order to prevent duplicate computation, we include a rule to ensure that at most one $\mathsf{needs\_a}$ constraint is present per node in the tree:

$$\mathsf{needs\_a}(I) \setminus \mathsf{needs\_a}(I) \iff \top$$

If the CHR engine supports rule priorities (§ 2.1), the rules for dependency tracking should be given the highest priority to ensure that duplicates are removed before any other work is started.

We want to compute the attribute a only if it has been signalled as necessary. For that reason, we refine the translation of semantic functions to match also a needs_a constraint. Once the rule fires, we remove that last constraint, to ensure that the computation does not happen again.

$$\mathsf{c}(I, A_1, \ldots, A_n), \mathsf{b}_1(J_1, V_1), \ldots, \mathsf{b}_m(J_m, V_m) \setminus \mathsf{needs\_a}(I) \iff X = \hat{\lambda} \mid \mathsf{a}(I, X)$$

Constraint gathering as CHRs, redux. The main change to introduce backwards reasoning in our example of λ-calculus is the set of CHRs taking care of dependency tracking. We obtain them by inspection of the semantic functions:

$$\begin{array}{rrcll}
\mathsf{var}(I,V), & \mathsf{deps_res}(I) & \iff & \mathsf{needs_res}(I), & \mathsf{deps_env}(I) \\
\mathsf{abs}(I,V,T_V,B), & \mathsf{deps_env}(B) & \iff & \mathsf{needs_env}(B), & \mathsf{deps_env}(I) \\
\mathsf{abs}(I,V,T_V,B), & \mathsf{deps_res}(I) & \iff & \mathsf{needs_res}(I), & \mathsf{deps_res}(B) \\
\mathsf{app}(I,E_1,E_2), & \mathsf{deps_env}(E_1) & \iff & \mathsf{needs_env}(E_1), & \mathsf{deps_env}(I) \\
\mathsf{app}(I,E_1,E_2), & \mathsf{deps_env}(E_2) & \iff & \mathsf{needs_env}(E_2), & \mathsf{deps_env}(I) \\
\mathsf{app}(I,E_1,E_2), & \mathsf{deps_res}(I) & \iff & \mathsf{needs_res}(I), & \mathsf{deps_res}(E_1), \mathsf{deps_res}(E_2)
\end{array}$$

We refrain for writing down the rules preventing duplication of needs_a constraints, since they all follow the same simple pattern.

Termination. Whereas in the first translation we presented the scheduling of attributes could depend on information gathered during its processing, in the new translation the schedule is generated once and for all at the beginning of the process. The set of needs_a constraints is thus an *over-approximation* of the actual set of dependencies. As a consequence, we have narrowed the set of AGs for which this process is guaranteed to terminate.

The AGs for which our dependency tracking terminates are known as *Absolutely Non-Circular Attribute Grammars* (ANCAGs) [11]. This class of AGs is very important in practice; they are considered the largest class for which generating a strict evaluator is tractable.

In the class of ANCAGs the choice of which attributes to evaluate in a subtree may depend on the particular constructor from which the node is built. This is clear from our characterization as CHRs, since the constraint describing the node appears in the head. We might ask when is it possible to find a schedule for the attributes for an entire data type, independently of the constructors used in each node. This problem has received quite some attention in the literature [10,14,3], leading to the definition of different subsets of AGs, such as Linearly Ordered, Partitionable or Ordered AGs.

3.3 Extensibility

Our translation of AGs into CHRs makes them very easy to extend along two orthogonal axes: either by adding new attributes or by adding new constructors.

These axes corresponds to the two facets of the Expression Problem. We exemplify this fact by adding a new attribute λs which returns the number of λ-abstractions in a program, and a new constructor for **let** bindings.

The semantic functions for the new attribute are computed as follows:

sem *Expr*
| *Var* **lhs**.$\lambda s = 0$
| *Abs* **lhs**.$\lambda s = 1 + @e.\lambda s$
| *App* **lhs**.$\lambda s = @e1.\lambda s + @e2.\lambda s$

As explained above, we need to generate two sets of CHRs. The first one build up the $\mathsf{need\_\lambda s}$ constraints by tracking dependencies:

$$\begin{aligned}
\mathsf{var}(I, V),\, \mathsf{deps_\lambda s}(I) &\iff \mathsf{needs_\lambda s}(I) \\
\mathsf{abs}(I, V, T_V, B),\, \mathsf{deps_\lambda s}(I) &\iff \mathsf{needs_\lambda s}(I),\, \mathsf{deps_\lambda s}(B) \\
\mathsf{app}(I, E_1, E_2),\, \mathsf{deps_\lambda s}(I) &\iff \mathsf{needs_\lambda s}(I),\, \mathsf{deps_\lambda s}(E_1),\, \mathsf{deps_\lambda s}(E_2)
\end{aligned}$$

The second step is computing the attribute values themselves:

$$\begin{aligned}
\mathsf{var}(I, V) \setminus \mathsf{needs_\lambda s}(I) &\iff \lambda\mathsf{s}(I, 0) \\
\mathsf{abs}(I, V, T_V, B),\, \lambda\mathsf{s}(B, L) \setminus \mathsf{needs_\lambda s}(I) &\iff \lambda\mathsf{s}(I, L + 1) \\
\mathsf{app}(I, E_1, E_2),\, \lambda\mathsf{s}(E_1, L_1),\, \lambda\mathsf{s}(E_2, L_2) \setminus \mathsf{needs_\lambda s}(I) &\iff \lambda\mathsf{s}(I, L_1 + L_2)
\end{aligned}$$

We can just add these new CHRs to the original set and have our new attribute computed. No change to the original set of CHRs is needed.

The other facet of extensibility is being able to add a new constructor. In this case, we are interested in the description of the **let** $x :: \tau = e_1$ **in** e_2 construct. Using UUAGC notation, our constructor is defined as:

data *Expr* | *Let* $v ::$ *TermVar*, *ty* :: *Type*, *e1*, *e2* :: *Expr*

A new kind of constraint $\mathsf{let}(I, V, T, E_1, E_2)$ represents this new type of node. The computation of attributes on the other type of nodes remains the same, so no change is needed in those CHRs. Of course, we need to specify how to compute the attributes for the new *Let* constructor. Fpr the sake of conciseness, we shall not write down the rules corresponding to dependency tracking.

$$\begin{aligned}
\mathsf{let}(I, V, T, E_1, E_2), \mathsf{env}(I, E) \setminus \mathsf{needs_env}(E_1) &\iff \mathsf{env}(E_1, E) \\
\mathsf{let}(I, V, T, E_1, E_2), \mathsf{env}(I, E) \setminus \mathsf{needs_env}(E_2) &\iff \\
E' = (V, T) : E \mid \mathsf{env}(E_2, E) & \\
\mathsf{let}(I, V, T, E_1, E_2), \mathsf{res}(E_1, T_1, C_1), \mathsf{res}(E_2, T_2, C_2) \setminus \mathsf{needs_res}(I) &\iff \\
C' = T_1 \equiv T : C_1 \;+\!\!+\; C_2 \mid \mathsf{res}(I, T_2, C') &
\end{aligned}$$

In conclusion, we have achieved full extensibility for adding both new attributes and new constructors. It should be noted, however, that some of this power comes from the fact that our CHRs need not be typechecked, in contrast to implementations such as UUAGC which embodies Haskell's type system.

4 Higher-Order Attribute Grammars

At the end of the previous section we extended the language of λ-expressions with a **let** construct. Then, we defined the typing procedure completely independent of the rest of constructors. However, we usually expect **let** $x = e_1$ **in** e_2 to behave similarly to $(\lambda x.e_2)\ e_1$. Thus, another approach to define typing for **let** bindings is to transform them, and then compute the typing on this version.

Let us encode this idea by using a new attribute *trans* and redefining the semantic function for *res* accordingly:

```
attr Expr
  syn trans :: Expr
sem Expr
  | Let lhs.trans      = App (Abs @v @ty @e2) @e1
        lhs.trans.env  = @lhs.env
        lhs.res        = @lhs.trans.res
```

In this case, *trans* is an example of a *higher-order attribute*, an attribute whose value is itself a tree. Thus, we can apply the attribute grammar to that value and decorate it with attributes. In the computation of *res* we take advantage of that fact: we make use of the *res* attribute as computed for the translated version of the **let** expression.

Higher-order attribute grammars (or HOAGs, for short) blurry the distinction between attributes and trees [21]. Semantic functions may not only depend on the value of other attributes and the fields of a constructor, but also on information obtained from decorating an attribute. HOAGs embody the idea of multi-pass compilers, in which the output of a phase over an AST is again another AST, which is fed to the next analysis or transformation.

The translation of semantic functions into CHRs only needs small changes to accommodate higher-order attributes:

- In order to apply the attribute grammar over the computed tree, we need to flatten the tree, in the sense of § 3.1. In most cases this operation makes use of fresh identifiers for the nodes of the tree.
- The constraint representing a higher-order attribute saves the identifier of the root node of the tree. In contrast, the constraint representing a regular attribute saves the entire value.

Applying these ideas we reach the following translation of the new semantic functions to compute the *res* value over *Let* nodes in a λ-expression:

$$\mathsf{let}(I,V,T,E_1,E_2) \setminus \mathsf{needs\_trans}(I) \iff \mathsf{trans}(I,X), \mathsf{app}(X,Y,E_1), \mathsf{abs}(Y,V,T,E_2) \quad \text{where } X \text{ and } Y \text{ are fresh identifiers}$$

$$\mathsf{let}(I,V,T,E_1,E_2), \mathsf{trans}(I,X), \mathsf{env}(I,E) \setminus \mathsf{needs\_env}(X) \iff \mathsf{env}(X,E)$$

$$\mathsf{let}(I,V,T,E_1,E_2), \mathsf{trans}(I,X), \mathsf{res}(X,R) \setminus \mathsf{needs\_res}(I) \iff \mathsf{res}(I,R)$$

The remaining question is: how do we schedule the attributes? If we try to use the same approach as for first-order attribute grammars, we end up in a situation in which we do not know which nodes to refer in the dependency constraints, because those nodes do not exist before the higher-order attribute is computed.

The solution is to introduce new types of constraints to recall the more complex dependency structure of the code. Right now we have $\mathsf{deps\_a}(N)$ to mean that we need to process the dependencies of attribute a on node N. We extend this idea to further levels of depth: $\mathsf{deps\_a\_b}(N)$ means that we need the attribute b computed over the value of attribute a on node N. The second step is to introduce tokens $\mathsf{on\_a\_deps\_b}(N)$ to remember that we need to process the dependencies of attribute b once we know the value of attribute a.

This lengthy description is better understood by looking at our example of the λ-calculus. The attribute *trans* belongs to a node in the tree, so the simpler algorithm suffices. Furthermore, it has no dependencies on other attributes.

$$\mathsf{let}(I, V, T, E_1, E_2) \setminus \mathsf{deps\_trans}(I) \iff \mathsf{needs\_trans}(I)$$

In order to compute *res* for **lhs**, we need to compute *res* over the attribute *trans*:

$$\mathsf{let}(I, V, T, E_1, E_2) \setminus \mathsf{deps\_res}(I) \iff \mathsf{needs\_res}(I), \mathsf{deps\_trans\_res}(N)$$

At this point we need to stage our work in two phases. In order to compute *trans.res*, we first need to compute *trans*. However, we cannot schedule the dependencies of *trans.res* before we know the value of *trans*. We need to defer the computation of dependencies until *trans* is known. For that reason we introduce the token $\mathsf{on\_trans\_deps\_res}$ and remove it once we know the value of *trans*.

$$\begin{aligned} \mathsf{let}(I, V, T, E_1, E_2) \setminus \mathsf{deps\_trans\_res}(I) &\iff \\ \mathsf{deps\_trans}(I), \mathsf{on\_trans\_deps\_res}(I) & \\ \mathsf{let}(I, V, T, E_1, E_2), \mathsf{trans}(I, X) \setminus \mathsf{on\_trans\_deps\_res}(I) &\iff \mathsf{deps\_res}(X) \end{aligned}$$

Notice that we do not need to add any rule for *trans.env*. Rather, if at some point in the computation of the dependencies of *trans.res* the value of that attribute is needed, a constraint $\mathsf{needs\_env}(X)$ is produced. The rule corresponding to the semantic function defining its value in terms of the environment of the pre-translation node then fires and computes the corresponding value.

In conclusion, our approach to AGs as CHRs extends naturally if higher-order attributes are present. The extra rules we define take care of the implicit dependency between a tree and the attributes which decorate that tree. Be aware that if we opt for a forward reasoning translation instead, the initial values of inherited attributes must be provided at the same time as the tree in the higher-order attribute is built.

5 Look-Ahead and Specialization

Attribute grammars operate uniformly on trees and base their decision on which semantic function to apply solely on the kind of node we are looking at. In some

scenarios, though, we need a more flexible way to make this decision. First, we may look at more elements of the node than just the constructor: some values of the fields, or the structure of some subtrees. This ability is termed *look-ahead* [7]. Second, we want to express the fact that a semantic function should only apply in a specific scenario, and let the rest of the cases be handled by the default rule. We call this ability *specialization*. In this section we see how our transformation of AGs into CHRs copes with these requirements.

Our main application of these ideas comes from specialized type rules [9,16]. Specialized type rules allow the writer of an embedded domain-specific language to provide a different typing procedure and error messages for the terms in that language. The goal of this customization is to phrase error diagnosis also in terms of the specific domain at hand, instead of in host language terms. Since specialized type rules take over default type rules, they are an example of specialization. Furthermore, the rules in [16] provide rich mechanisms for matching trees where a rule applies, which can be simulated using look-ahead.

An example of specialized type rules comes from typing literal lists. After desugaring, a list $[e_1, e_2, ..., e_n]$ is converted into $cons\ e_1\ (cons\ e_2\ (...\ (cons\ e_n\ nil)))$. The default typing procedure works bottom-up, which means that if two terms do not have the same type, the right-most one will be shown as the culprit. A procedure which is nearer to the programmer expectation is to gather all the types from the terms, and then check that all of them are equal. As a result, if typing fails, the blame is on all the expressions, without bias. We express this procedure using an extension of our attribute grammar syntax:[4]

```
attr Expr ! list
  syn eltTys : [Type], eltCs : [Constraint]
attr Expr ! list
  | Var "nil"  lhs.eltTys = []
               lhs.eltCs  = []
               lhs.res    = (List freshTyVar, [])
  | App (App (Var "cons") e1) e2
               e1.env     = lhs.env
               e2.env     = lhs.env
               lhs.eltTys = fst @e1.res : @e2.eltTys
               lhs.eltCs  = snd @e1.res ++ @e2.eltCs
               lhs.res    = @lhs.eltCs ++ [σ ≡ τ | τ ← @lhs.eltTys]
                            where σ = freshTyVar
```

In this case, we have a specialization called *list*, which makes use of two extra attributes, *eltTys* and *eltCs*, which hold the types and the constraints of each term in the list, respectively. The new semantic functions for the specialization only apply to the case of the empty list or the consing operator. Since in the latter case the definition of *res* uses the *eltCs* attribute for *e2*, which is defined

[4] This description is not in Bochmann Normal Form, but it can be rewritten in such a shape by replacing **lhs**.*eltCs* and **lhs**.*eltTys* by their definitions in **lhs**.*res*.

only when the *list* specialization applies, that means that this specialization must have also been applied to *e2*. The result is that the specialization only applies if a complete list literal, ending in *nil*, is found verbatim in the source code.

5.1 Translation to CHRs

The look-ahead part is easy to encode in our CHR translation. Instead of just matching on one single node, match on the entire structure you wish to check. For example, the definition of *res* for the *nil* case is:

$$\mathsf{var}(I, \texttt{"nil"}) \setminus \mathsf{needs\_res}(I) \iff \sigma = \mathit{freshTyVar} \mid \mathsf{res}(I, \mathit{List}\ \sigma, [\,])$$

However, in this form the rule overlaps with the default rule for var constraints. The CHR engine is free to choose whatever rule it prefers. But a specialized type rule must be *preferred* over the default one, whenever it applies.

Our solution is to introduce a new sort of constraints, *specialization tokens*, which mark the semantic functions chosen for each node. There is one such token $\mathsf{spec}(N)$ per specialization, plus a $\mathsf{default}(N)$ one to mark the default case. Now we can distinguish which semantic function to apply for the case of variables:

$$\begin{aligned} \mathsf{list}(I), \mathsf{var}(I, \texttt{"nil"}) \setminus \mathsf{needs\_res}(I) &\iff \sigma = \mathit{freshTyVar} \mid \mathsf{res}(I, \mathit{List}\ \sigma, [\,]) \\ \mathsf{default}(I), \quad \mathsf{var}(I, V) \setminus \mathsf{needs\_res}(I) &\iff \ldots \end{aligned}$$

The CHRs taking care of dependencies must also be modified to account for the differences between specializations and default cases. For example, the computation of the *res* attribute for the *cons* case in the *list* specialization needs both *eltCs* and *eltTys*, and such computation in turn needs the value of some attributes in the inner expressions.

$$\begin{aligned} \mathsf{list}(I), \mathsf{app}(I, X, E2), \mathsf{app}(X, V, E1), \mathsf{var}(V, \texttt{"cons"}) \setminus \quad \mathsf{deps\_res}(I) &\iff \\ \mathsf{needs\_res}(I), \mathsf{deps\_eltCs}(I), \mathsf{deps\_eltTys}(I) & \\ \mathsf{list}(I), \mathsf{app}(I, X, E2), \mathsf{app}(X, V, E1), \mathsf{var}(V, \texttt{"cons"}) \setminus \mathsf{deps\_eltTys}(I) &\iff \\ \mathsf{needs\_res}(I), \mathsf{deps\_res}(E1), \mathsf{deps\_eltTys}(E2) & \\ \mathsf{list}(I), \mathsf{app}(I, X, E2), \mathsf{app}(X, V, E1), \mathsf{var}(V, \texttt{"cons"}) \setminus \quad \mathsf{deps\_eltCs}(I) &\iff \\ \mathsf{needs\_res}(I), \mathsf{deps\_res}(E1), \mathsf{deps\_eltCs}(E2) & \end{aligned}$$

The remaining question is how to assign the specialization tokens to each node. Since we do know whether a certain assignment will succeed in computing attributes, we use the disjunction mechanism in CHR$^\vee$ described in § 2.1. The different possibilities are introduced in a single rule, $\mathsf{do}(N)$, which has to be generated with all possible specializations in place. In our case it reads:

$$\mathsf{do}(N) \iff \mathsf{list\_do}(N)\ ;\ \mathsf{default\_do}(N)$$

These do constraints have to be propagated downwards. As in the case of dependencies, we introduce proxy constraints list_do and default_do to ensure that only one such token is produced per node.

$$\begin{aligned} \mathsf{var}(I, V) \setminus \mathsf{list\_do}(I) &\iff \mathsf{list}(I) \\ \mathsf{abs}(I, V, T_V, B) \setminus \mathsf{list\_do}(I) &\iff \mathsf{list}(I), \mathsf{do}(B) \\ \mathsf{app}(I, E_1, E_2) \setminus \mathsf{list\_do}(I) &\iff \mathsf{list}(I), \mathsf{do}(E_1), \mathsf{do}(E_2) \end{aligned}$$

In the case of specialization, the propagation only happens in those cases matched by the semantic functions.

$$\begin{aligned} \mathsf{var}(I, \texttt{"nil"}) \setminus \mathsf{list\_do}(I) &\iff \mathsf{list}(I) \\ \mathsf{app}(I, X, E2), \mathsf{app}(X, V, E1), \mathsf{var}(V, \texttt{"cons"}) \setminus \mathsf{list\_do}(I) &\iff \mathsf{list}(I), \mathsf{do}(E_1), \mathsf{do}(E_2) \end{aligned}$$

Here comes the trick: in the rest of the cases, the specialization cannot be applied. We encode this by a simplification to $\bot$. This in turn starts the backtracking procedure and tries another specialization or the default case. However, in order to ensure that such a failure rule only applies when no other matches, we need to assign the *lowest possible priority*, which we represent here by $-\infty$.

$$\mathsf{list\_do}(I) \iff \bot \ \textbf{priority} \ -\infty$$

Using the same ideas, other specialization procedures could be devised. For example, we could check some property on the fields, not only its shape. Or we could defer failure, and thus backtracking, until some attributes are already computed, and base our decision on those. However, the more complex matching we do, the more convoluted and less modular our set of CHRs becomes.

6 Related and Future Work

One of the most important pieces in our translation from AGs to CHRs is the handling of dependencies between nodes and attributes. Our rules dealing with deps and do constraints can be seen as a constraint solver for scheduling attributes of a specific AG. This contrasts with the classic trend of having just one algorithm dealing with a subset of AGs.

Another proposal to schedule the attributes per specific AG is to use SAT solving instead of constraint solving [3]. This approach is quite useful when the end goal is to produce a strict evaluator for an AG which minimizes the number of tree traversals. In our CHR translation, however, the notion of traversal is not present, since we flatten the tree beforehand.

Our translation deals only with AGs defined by semantic *functions*. This results on CHR guards composed only of equalities. However, we can lift this restriction and deal with semantic *relations* instead. This moves us closer to relational attribute grammars [5]. We want to explore this relation in future work, as a way to unify the approaches of bidirectional and constraint-based type systems.

The CHRs resulting from our translation can be easily combined and extended, as shown in § 3.3. This property holds even when higher-order attributes are present. It is not clear, though, how modular and extensible specialization and look-ahead are. As future work we plan to compare our approach to others such as the AspectAG system [20] or the `fold` Haskell package.

References

1. Abdennadher, S., Schütz, H.: CHR$^\vee$: A Flexible Query Language. In: Proceedings of the 3rd Intl. Conf. on Flexible Query Answering Systems. pp. 1–14. FQAS '98
2. Bochmann, G.V.: Semantic evaluation from left to right. Commun. ACM 19(2), 55–62 (Feb 1976)
3. Bransen, J., van Binsbergen, L.T., Claessen, K., Dijkstra, A.: Linearly Ordered Attribute Grammar Scheduling Using SAT-Solving. In: Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, London, UK, April 11-18, 2015. Proceedings. pp. 289–303 (2015)
4. De Koninck, L., Schrijvers, T., Demoen, B.: User-definable Rule Priorities for CHR. In: Proceedings of the 9th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming. pp. 25–36. PPDP '07, ACM (2007)
5. Deransart, P., Małuszynski, J.: Relating logic programs and attribute grammars. The Journal of Logic Programming 2(2), 119 – 155 (1985)
6. Dijkstra, A.: Stepping through Haskell. Ph.D. thesis, Department of Information and Computing Sciences, Universiteit Utrecht (November 2005)
7. Engelfriet, J.: Top-down tree transducers with regular look-ahead. Mathematical systems theory 10(1), 289–303 (1976)
8. Frühwirth, T.: Constraint Handling Rules. Cambridge University Press (2009)
9. Heeren, B., Hage, J., Swierstra, S.D.: Scripting the Type Inference Process. In: 8th ACM SIGPLAN Intl. Conf. on Functional Programming. ICFP '03, ACM (2003)
10. Kastens, U.: Ordered attributed grammars. Acta Informatica 13(3), 229–256 (1980)
11. 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. pp. 32–49. POPL '76, ACM (1976)
12. Knuth, D.E.: Semantics of Context-Free Languages. Mathematical Systems Theory 2(2), 127–145 (1968)
13. Koninck, L., Schrijvers, T., Demoen, B.: Constraint handling rules. chap. A Flexible Search Framework for CHR, pp. 16–47. Springer-Verlag (2008)
14. Natori, S., Gondow, K., Imaizumi, T., Hagiwara, T., Katayama, T.: On eliminating type 3 circularities of ordered attribute grammars (1999)
15. Pottier, F., Rémy, D.: The essence of ML type inference. In: Pierce, B.C. (ed.) Advanced Topics in Types and Programming Languages. MIT Press (2005)
16. Serrano, A., Hage, J.: Type Error Diagnosis for Embedded DSLs by Two-Stage Specialized Type Rules. In: 25th European Symp. on Programming, ESOP 2016
17. Sneyers, J., Weert, P.V., Schrijvers, T., Koninck, L.D.: As time goes by: Constraint Handling Rules. TPLP 10(1), 1–47 (2010)
18. Sulzmann, M., Wazny, J., Stuckey, P.J.: A Framework for Extended Algebraic Data Types. In: 8th Intl. Conf. on Functional and Logic Programming. pp. 47–64 (2006)
19. Swierstra, S.D., Alcocer, P.R.A., Saraiva, J.: Designing and implementing combinator languages. In: Advanced Functional Programming. pp. 150–206 (1998)
20. Viera, M., Swierstra, S.D., Swierstra, W.: Attribute Grammars Fly First-Class: How to do Aspect Oriented Programming in Haskell. In: ICFP'09: Proceedings of the 2009 SIGPLAN International Conference on Functional Programming (2009)
21. Vogt, H., Swierstra, S.D., Kuiper, M.F.: Higher-order attribute grammars. In: Proceedings of the ACM SIGPLAN'89 Conference on Prog. Lang. Design and Implementation (PLDI), Portland, Oregon, USA, June 21-23. pp. 131–145 (1989)
22. Vytiniotis, D., Peyton Jones, S., Schrijvers, T., Sulzmann, M.: OutsideIn(X): Modular type inference with local assumptions. Journal of Functional Programming 21(4-5), 333–412 (Sep 2011)