Dependently-Typed Atribute Grammars

Arie Middelkoop, Atze Dijkstra, and S. Doaitse Swierstra

Universiteit Utrecht,
The Netherlands

Abstract. Attribute Grammars (AGs) are a convenient formalism to program complex computations over Abstract Syntax Trees (ASTs), which is particularly useful for the construction of compilers. Given such an AG, it is not obvious if it is correct or not. Dependently-typed programming provides a means to address this challenge. We show in this paper the extension of AGs with attributes that may have a dependent type. This allows us to define and prove by-construction the correctness of such computations.

1 Introduction

The static semantics of context-free languages can be described with Attribute Grammars (AGs) [6]. Attribute grammars have the distinct advantage that the descriptions are composable and easy to extend, which turns out to be convenient when working with complex semantics.

AGs extend a context-free grammar with attributes associated with a nonterminal, and equations (rules) between these attributes associated with a production. An Abstract Syntax Tree (AST) represents a proof of the recognition of a sentence in the context-free language. Attributes are convenient: these represent individual aspects or properties of the AST, can be combined at each node of the AST to define more complex attributes, and it is easy to add new attributes to the grammar. From a practical point of view, via a preprocessor, an AG can be compiled to an efficient algorithm in some host language that, given an AST, computes values for these attributes. The translated code in the host language then serves as an implementation for a compiler.

In the last years, we applied AGs in several large projects, including the Utrecht Haskell Compiler [2], the Helium [4] compiler for learning Haskell, and the editor Proxima [14]. The use of AGs in these projects is invaluable, for two reasons: rules for attributes can be written in isolation and combined automatically, and trivial rules can be inferred, saving the programmer from writing boilerplate code.

Given a compiler such as the UHC, the question arises to what extent the compiler generates correct code (for some definition of correctness). In UHC, similar to any project that many people contribute to, we regularly encounter subtle bugs related to broken hidden invariants that are not discovered by regression testing alone.

The holy grail in compiler implementation is to formally verify that a compiler exhibits certain properties of interest. For example, that a type inferencer is a sound and complete implementation of a type system, and that the meaning of a well typed source program is preserved. Despite some successes [8], formal verification is still infeasible

for compilers that deal with complex language features. It is already a struggle to implement and maintain a compiler that gives output that seems correct, let alone to attempt to prove this without having a good intuition about the structure of such a proof.

When we implement our compilers in languages with advanced type systems, we ensure that compilers have certain properties by construction. In mainstream strongly typed languages, it is possible to guarantee that (even for non-trivial inputs) the compiler produces output that conforms to a context-free grammar. It is enforceable that a compiler produces, for example, syntactically-correct HTML or C code. These are properties we get for free when we define our data structures properly in, for example, the strongly-typed programming language Haskell.

Unfortunately, syntactic guarantees do not save us from hidden invariants. For that, we desire semantic guarantees. This is possible by using more complex types, but at a severe cost. It actually requires us to encode proofs by using the type system as a logic. In a dependently-typed language such as Agda, we impose arbitrary complex constraints on values via types, but as a consequence, proving that the program is well typed becomes considerably harder.

To move from a compiler implementation in Haskell to Agda, for example, the least extra effort required—syntax and library issues aside—is to prove termination of the program (ensures that the program is a proof in a consistent logic). Compilers mostly consist of primitive recursive functions. Those parts in a compiler that rely on iteration either compute fixpoints, or have iteration limits. The extra effort is substantial, but not a theoretic challenge.

From there on, the programmer can make the types more specific, and prove stronger invariants. Since we implement our compilers with AGs, we present in this paper how to combine these with dependently-typed programming, using Agda in particular.

We compile AGs to (a sequence of) functions that take a record of inherited attributes, have cases for each type of AST node, recursively visit the children of such a node, and produce a record of synthesized attributes. Such functions can be written in Agda. There are thus no theoretic limitations that prevent AGs to be embedded in a dependently-typed programming language.

However, in the extended version of this paper [10], we show that dependently-typed programming enforces structure to a an AG that gives rise to coarser attributes and right-hand sides of rules that case-analyze values of attributes. The coarser the granularity of attributes, the less benefits an AG has over a conventional fold. In short, to profit from programming with AGs, a fine granularity of attributes is beneficial, but may require partially defined attributes. In a dependently-typed setting, all attributes must have a total definition, and to accomplish that, a coarser granularity of attributes may be required. In the extended version, we show that it is possible to retain the fine granulairty when we allow AST nodes to reside in inspectable states with different attribuations (context-dependent attribuation).

In previous work, we implemented an AG preprocessor RULER-FRONT for Haskell, and some other languages [11, 12]. In this paper, we introduce a variation called $\text{RULER-FRONT}_{\text{AGDA}}$, which has additional syntax to support the more complex types for attributes, and to deal with dependently-typed pattern matching in the left-hand sides of rules. In particular, dependent pattern matching against attributes may impose additional constraints on

other attributes. For example, in the scope of a match against an equality proof, we get the additional information that two attributes are the same and Agda requires them to have a common name. The example in Section 2 again demonstrates this.

As contributions, we extensively discuss an example in Section 2, then describe the AG-language RULER-FRONT in Section 3 and its translation to Agda.

2 Example

In this section, we demonstrate a mini-compiler implemented as dependently-typed attribute grammar. We take as example scope checking of a simple language. The actual concrete syntax of the source language that the compiler takes and the target language that the compiler produces is irrelevant for this paper. The abstract syntax we represent with Agda data types given below. Scope checking of a simple language is only a minor task in a compiler. However, it still shows many aspects that a more realistic compiler has.

The code of this compiler consists of blocks of Agda code, and blocks of RULER-FRONT. To easy the distinction between the two, we underline keywords of Agda, and format keywords of RULER-FRONT bold.

2.1 Dependently-typed program in Agda

The main distinctions between Agda and a conventional programming language such as Haskell are dependent type signatures and dependent pattern matching. We briefly explain these distinctions in this section.

As an example, below is a block of Agda code. We introduce types to represent identifiers, environments, and scoping errors. We simply take identifiers as strings, and an environment is a cons-list of them.

```
Ident : Set
Ident = String

Env : Set
Env = List Ident

data Err : Set where
  scope : (nm : Ident) → Err

Errs : Set
Errs = List Err
```

We use one type of error, a *scope*-error, consisting of an identifier *nm*. The constructor *scope* takes a value of type *Ident* as parameter. In a dependent type, we may give a name to such a value in the type signature. In this example, we give it the name *nm*. This name may be used in the remainder of the signature to specify predicates over *nm* (types that depend on *nm*). We do not require this functionality yet; later, we add an additional field to the *scope* constructor to represent a proof that *nm* is not in some environment.

A type $nm \in \Gamma$ (in *Set*) is an example of a predicate as mentioned above, where *nm* and Γ are value-level identifiers. A value of the above predicate is a proof that the predicate holds over the value-level identifiers.

In the example, we manipulate environments. We reason about identifiers in the environment. Two relations, $\_ \sqsubseteq \_$ and $\_ \in \_$, play an important role. A predicate $\Gamma_1 \sqsubseteq \Gamma_2$ specifies that Γ_1 is a subsequence of Γ_2, and $nm \in \Gamma$ that specifies that nm is an element of Γ. Such relations are representable in Agda as data types, where the clauses of the relation are represented as constructors. A value of such a type is a proof: a derivation tree with constructor-values as nodes.

We use the subsequence relation between environments in the example to reason about environments that we extend by prepending or appending identifiers of other environments. A subset relation would be another possibility, but former suffices and is slightly easier in use. The curly braces in the code below introduce implicit parameters that may be omitted if their contents can be fully determined by Agda through unification.

```
data _⊑_ : (Γ₁ : Env) → (Γ₂ : Env) → Set where
  trans    : {Γ₁ : Env} {Γ₂ : Env} {Γ₃ : Env} → Γ₁ ⊑ Γ₂ → Γ₂ ⊑ Γ₃ → Γ₁ ⊑ Γ₃
  subLeft  : {Γ₁ : Env} {Γ₂ : Env} → Γ₁ ⊑ (Γ₁ ++ Γ₂)
  subRight : {Γ₁ : Env} {Γ₂ : Env} → Γ₂ ⊑ (Γ₁ ++ Γ₂)
```

With *subLeft* and *subRight*, we can add arbitrary environments in front or after the environment on the right-hand side. With *trans* we repeat this process.

For membership in the environment, either the element has to be on the front of the list, or there is a proof that the element is in the tail.

```
data _∈_ : (nm : Ident) → (Γ : Env) → Set where
  here : {nm : Ident} {Γ' : Env} → nm ∈ (nm :: Γ')
  next : {nm : Ident} {nm' : Ident} {Γ : Env} → nm ∈ Γ → nm ∈ (nm' :: Γ)
```

With the above definitions, we prove lemma *inSubset* that, when an environment Γ' is a subsequence of Γ, and nm is an element in Γ', states that nm is also in Γ.

```
append : {nm : Ident} {Γ : Env} → (nm ∈ Γ) → (Γ' : Env) → (nm ∈ (Γ ++ Γ'))
append {nm} {.nm :: Γ} (here)      Γ' = here {nm} {Γ ++ Γ'}
append {nm} {nm' :: Γ} (next inΓ) Γ' = next (append {nm} {Γ} inΓ Γ')
append {_}  {[]}       ()         _

prefix : {nm : Ident} {Γ' : Env} → (nm ∈ Γ') → (Γ : Env) → (nm ∈ (Γ ++ Γ'))
prefix inΓ' []       = inΓ'
prefix inΓ' (x :: Γ) = next (prefix inΓ' Γ)

inSubset : {nm : Ident} {Γ : Env} {Γ' : Env} → (Γ' ⊑ Γ) → nm ∈ Γ' → nm ∈ Γ
inSubset (subLeft {_} {Γ'}) inΓ' = append inΓ' Γ'
inSubset (subRight {Γ})     inΓ' = prefix inΓ' Γ
inSubset (trans subL subR) inΓ' = inSubset subR (inSubset subL inΓ')
```

The proof of *inSubset* consists of straightforward structural induction. The lemmas *append* and *prefix* take care of the base-cases. With *append*, the proof for Γ and $\Gamma ++ \Gamma'$ is the same until we find the element. We then produce a *here* with the appropriate environment. When we have a proof that an identifier is in the environment, we never run into an empty environment without finding the identifier first. Agda's totally checker

requires us to add a clause for when the environment is empty. The match against the proof can then not succeed, hence the match is called absurd (indicated with the empty parentheses), and the right-hand side must be omitted.

Similarly, we prove that given an identifier *nm* and an environment Γ, we either determine that *nm* is in Γ or that it is not in Γ. The sum-type $\_ \uplus \_$ (named *Either* in Haskell) provides constructors inj_1 (*Left* in Haskell) and inj_2 (*Right* in Haskell) for this purpose.

In this proof, we use *notFirst* to prove by contradiction that an identifier is not in the environment if it is not equal to the head of the environment, and neither in the tail of the environment. If it would be in the head, it would conflict with the former assumption. If it would be in the tail, it would conflict with latter assumption.

$$
\begin{aligned}
&\mathit{notFirst} : \{\mathit{nm} : \mathit{Ident}\}\ \{\mathit{nm}' : \mathit{Ident}\}\ \{\Gamma : \mathit{Env}\} \to \neg(\mathit{nm} \equiv \mathit{nm}') \to \neg(\mathit{nm}' \in \Gamma) \\
&\qquad\qquad \to (\mathit{nm}' \in (\mathit{nm} :: \Gamma)) \to \bot \\
&\mathit{notFirst}\ \mathit{prv}_1\ \_\quad \mathit{here}\qquad\ = \mathit{prv}_2\ \mathit{refl} \\
&\mathit{notFirst}\ \_\quad \mathit{prv}_1\ (\mathit{next}\ \mathit{prv}_2) = \mathit{prv}_1\ \mathit{prv}_2
\end{aligned}
$$

To find an element nm' in environment, we walk over the cons-list. If the environment is empty, the proof that the identifier is not in the list is trivial: if it would be, neither *here* nor *next* can match, because they expect a non-empty environment, hence we can only match with an absurd pattern, which provides the contradiction.

If the environment is not empty, then it has an identifier *nm* as the head. We use the string equality test '*mbEqual*' to give us via constructor *yes* a proof of equality between *nm* and nm' or a proof of its negation via *no*. We wish to express the pattern $nm' \in_? (nm' :: \Gamma)$, with the intention of matching the case that nm' matches the head of the environment. In Agda, however, a pattern match must be strictly linear: an identifier may only be matched against once. To express that a value is the same as an already introduced value, we use a dot-pattern. This requires a proof that these are indeed the same, which is provided by a match against *refl* (the only constructor of an equality proof).

$$
\begin{aligned}
&\_ \in_? \_ : (\mathit{nm} : \mathit{Ident}) \to (\Gamma : \mathit{Env}) \to \neg(\mathit{nm} \in \Gamma) \uplus (\mathit{nm} \in \Gamma) \\
&\mathit{nm}' \in_? [\,] = \mathit{inj}_1\ (\lambda()) \\
&\mathit{nm}' \in_? (\mathit{nm} :: \Gamma)\ \underline{\mathsf{with}}\ \mathit{nm} \equiv_? \mathit{nm}' \\
&\mathit{nm}' \in_? (.\mathit{nm}' :: \Gamma)\ |\ \mathit{yes}\ \mathit{refl} = \mathit{inj}_2\ \mathit{here} \\
&\mathit{nm}' \in_? (\mathit{nm} :: \Gamma)\quad |\ \mathit{no}\ \mathit{prv}'\ \underline{\mathsf{with}}\ \mathit{nm}' \in_? \Gamma \\
&\mathit{nm}' \in_? (\mathit{nm} :: \Gamma)\quad |\ \mathit{no}\ \mathit{prv}'\ \ |\ \mathit{inj}_2\ \mathit{prv} = \mathit{inj}_2\ (\mathit{next}\ \{\mathit{nm}'\}\ \{\mathit{nm}\}\ \mathit{prv}) \\
&\mathit{nm}' \in_? (\mathit{nm} :: \Gamma)\quad |\ \mathit{no}\ \mathit{prv}'\ \ |\ \mathit{inj}_1\ \mathit{prv} = \mathit{inj}_1\ (\mathit{notFirst}\ \mathit{prv}'\ \mathit{prv})
\end{aligned}
$$

A with-expression is a variation on case-expressions in Haskell. It allows case distinction on a computed value, but also a refinement of previous pattern matches. Semantically, it computes the value of the scrutinee, then calls the function again with this value as additional parameter. Horizontal bars control which clauses belong to what with-expression.

2.2 Attribute Grammar in Agda

In AG blocks, we define the grammar, attributes, and functions between attributes. The preprocessor takes these blocks and translates them to Agda functions. The compiler is an Agda function implemented as AG that takes a source-language term of the type *Source* and translates it to a target-language term of the type *Target*, provided that there are no scoping errors. Initially, we take *Source* and *Target* as the same language, without using any dependent types. Later, in Section 2.3, we specify that a term of type *Target* is free of scoping errors, and have to prove that our compiler indeed produces such terms.

To use AGs to analyze *Source*-terms, we define a context-free grammar for it below. Later, we declare attributes on the nonterminals in the grammar, and give functions between these attributes. The *Source* language consists of a sequence ($\_ \diamond \_$) of *use* and *def* terms. Its operational semantics is simply: it evaluates to () if for every *use*, there is an equally named *def* in the sequence, otherwise evaluation diverges.

```
grammar Root : Set   -- start symbol of grammar and root of the AST
  prod root nonterm root : Source
grammar Source : Set
  prod use   term    nm    : Ident
  prod def   term    nm    : Ident
  prod _ ◇ _ nonterm left  : Source
             nonterm right : Source
```

The grammar actually represents an Agda data type, where the productions are constructors, and terminals and nonterminals are the fields. We optionally generate this data-type from such a grammar declaration. The following type for the *Target* language (structurally equal to *Source* for now) gives an example of this translation.

```
data Target : Set where
  use   : (nm : Ident) → Target
  def   : (nm : Ident) → Target
  _◇_ : (left : Target) → (right : Target) → Target
```

With an AG, we translate an AST to its semantics, which is conceptually an AST node decorated with attributes. We represent the semantics of an AST node as an Agda function that takes values for inherited attributes as inputs, and produces values for synthesized attributes as outputs. The conceptual decorated tree is the closure of this function.

For the example, we compute bottom-up a synthesized attribute *gathEnv* that contains the identifiers defined by the term. Top-down we pass an inherited attribute *finEnv* that contains all the gathered identifiers of the term combined with the initial environment. Furthermore, we compute bottom up an attribute *errs*, containing errors for each undefined identifier, and an attribute *target* containing the compiled version of the source term.

From the above informal description of the attributes, we can already indicate a problem. The inherited attribute *finEnv* depends on the synthesized attribute *gathEnv*. This requires us to provide a result of the function as part of the argument of the same

function. Such a circular programs are not allowed by Agda's termination checker. However, there is no circular dependency when we compute the attributes in two passes or more passes over the AST. In the first pass, we compute attribute *gathEnv*. In the second pass, we take *gathEnv* as resulting from the first pass and use it to compute the value for the inherited attribute *finEnv* for the second pass. For the class of Ordered Attribute Grammars [5], a multi-pass algorithm is derivable. Earlier work showed that this class is sufficiently strong for type checking Haskell. Several extensions may be needed when fixpoint iteration is required [11].

In Agda, we express a multi-pass algorithm as a coroutine: a coroutine is a function that can be invoked (visited) multiple times, each time it may take additional parameters and yield results. As semantics of a AST node, we take a coroutine that which each invocation takes some of the remaining inherited attributes as a record, and computes a record with some of the remaining synthesized attributes. With an interface declaration, we express statically what attributes are computed in what visit.

```
itf Root
  visit compile  inh initEnv  : Env
                 syn errs     : Errs
                 syn target   : Target
itf Source
  visit analyze  syn gathEnv : Env
  visit translate inh finEnv  : Env
                 syn errs     : Errs
                 syn target   : Target
```

An attribute may only be declared once. However, an inherited attribute with the name x is considered to be different from a synthesized attribute x. The order of appearance determines the visit-order. For an AST node of nonterminal Root, there is only one visit, named *compile*, which takes the initial environment and produces the outcome. It does so by invoking the two visits on its *Source* subtree. First the visit *analyze* to gather the identifier, and then the visit *translate* to produce the outcome.

The implementation of the coroutine is derived from functions that define attributes. These functions are called rules, and we specify them per production with a datasem-block. The left-hand side consists of a pattern that refers to one or more attributes, and the right-hand side is an Agda expression that may refer to attributes.

```
datasem Root
  prod root
    root.finEnv = root.gathEnv ⫩ lhs.initEnv
    lhs.errs    = root.errs
    lhs.target  = root.target
```

An attribute reference is denoted *childname.attrname*. The children names *loc* and *lhs* are special. With *lhs* we refer to the inherited and synthesized attributes of the current AST node. With *loc* we refer to local attributes of the node. A terminal field x of a production is in scope as attribute *loc.x*. A nonterminal field c of a production is in scope as child named c, with attributes as defined by the interface of the corresponding

nonterminal. An attribute $c.x$ on the left-hand side refers to a synthesized attribute if $c = lhs$, and an inherited attribute otherwise. On the right-hand side, it is exactly the other way around. To refer on the right-hand side to synthesized attributes of *lhs*, or to inherited attributes of a child[1], the attribute reference has to be additionally prefixed with *inh*. or *syn*..

For the *use*-production of *Source*, we check if the element is in the environment. If not, we produce a singleton error-list. The *Target* term is a copy of the *Source* term. For *def*, no errors arise, and again the *Target* term is a copy of the *Source* term. Finally, for $\_ \diamond \_$, we pass the *finEnv* down to both children, synthesize the *gathEnv* and *errs* of the children, and synthesize the target term.

```
datasem Source
  prod use
    lhs.errs     with loc.nm ∈? lhs.finEnv
                 | inj1 _ = [scope nm]
                 | inj2 _ = [ ]
    lhs.target   = use loc.nm
  prod def
    lhs.errs     = [ ]
    lhs.target   = def loc.nm
  prod _ ◇ _
    left.finEnv  = lhs.finEnv
    right.finEnv = lhs.finEnv
    lhs.gathEnv  = left.gathEnv ++ right.gathEnv
    lhs.errs     = left.errs ++ right.errs
    lhs.target   = left.target ◇ right.target
```

Some advantages of Attribute Grammars show up in this example. Firstly, the order of appearance of rules is irrelevant. This allows us to write the rules that belong to each other in separate datasem-block of the same nonterminal, and use the preprocessor to merge these separate blocks into a single block. For large grammars with many attributes, such as UHC, this is an essential asset to keep the code maintainable.

Furthermore, the rules for many attributes exhibit a standard pattern, especially when the grammar has many productions. For example, values for many attributes are simply passed down (*finEnv*, for example), or are a combination of some of the attributes of the children (*errs* and *gathEnv*, for example). We provide default rules [11] to abstract over these patterns. Children are ordered per production (for example, alphabetic order). When for a child c an attribute x is not explicitly defined, but there is default-rule for x, then we take as value for x the right-hand side of the default rule applied to a list of all the x attributes of the children smaller than c. In particular, first element of this list is the nearest smaller child. The last element in this list is *lhs* (if it has an attribute x).

[1] This is rarely useful with a conventional AG and can be simulated with local attributes. However, in a dependently-typed AG, we may wish to refer to such values in a predicate, as we see in a later example. Then it is convenient to be able to directly refer to such a value.

```
datasem Source
  default errs = concat    -- i.e. left.errs ++ right.errs
  default finEnv = last    -- i.e. lhs.finEnv
```

In the AG code for UHC about a third of the attributes have an explicit definition. All others are provided by such default-rules.

To effectively be able to use Attribute Grammars, the above two features are a necessity. In the next section, we construct a dependently-typed variation on the above Attribute Grammar, and need extensions to the AG formalism to retain these two features.

2.3 Dependently-Typed Attribute Grammar in Agda

We change the definition of the *Target* language. When constructing a term in this language, we demand a proof that all identifiers exist in the environment. For that, we give *Target* a type depending on a value: the final environment. Furthermore, we demand for each scoping error a proof that the identifier that caused the error is not in the environment. In this section, we show how a compiler that has these properties by construction. These are just a small number of properties. We can define many more, such as demanding that identifiers are not defined duplicately, or that error messages correspond to a used identifier in the source language. As we see in this section, an Attribute Grammar is suitable for exactly this purpose: the additional predicates and proofs just become additional attributes and rules.

```
data Target : (Γ : Env) → Set where
  def   : ∀ {Γ} (nm : Ident) → (nm ∈ Γ) → Target Γ
  use   : ∀ {Γ} (nm : Ident) → (nm ∈ Γ) → Target Γ
  _◇_ : ∀ {Γ} → (left : Target Γ) → (right : Target Γ) → Target Γ
data Err : Env → Set where
  scope : {Γ : Env} (nm : Ident) → ¬(nm ∈ Γ) → Err Γ
Errs : Env → Set
Errs env = List (Err env)
```

In the previous section, we returned both a list of errors and a target term as result, with the hidden invariant that the target term is invalid when there are errors. The advantage is that we could refer to both values as separate attributes. A fine granularity of attributes is important in order to use default rules, and to be able to refer to values without having to unpack it first. This also has a disadvantage: the hidden invariant is not enforced. Patterns like the above occur often in the code of UHC.

With the dependently-typed *Target* type, we are unable to construct such a term in the presence of errors. The hidden invariant is made explicit. However, we cannot construct both error messages and a target type anymore. Instead of two attributes, we now return a single attribute of type (*Errs inh.finEnv*)⊎(*Target inh.finEnv*), that contains either a list of error messages or a valid target term.

Furthermore, both the error messages and the target term take both the final environment as parameter. To keep knowledge about this environment hidden inside the

compiler, we wrap the result in a data type *Outcome*, which hides the environment via an existential. We produce this type at the root of the AST.

```
data Outcome : Set where
  outcome : ∀ {Γ} → (Errs Γ) ⊎ (Target Γ) → Outcome
```

The interfaces for the nonterminals now contain dependent-types that may depend on values of attributes. For example, we require a proof that the gathered environment is a subsequence of the final environment. The type of this proof refers to the two respective attributes.

```
itf Root
  visit compile  inh initEnv    : Env
                 syn outcome    : Outcome
itf Source
  visit analyze  syn gathEnv    : Env
  visit translate inh finEnv    : Env
                 inh gathInFin  : syn.gathEnv ⊆ inh.finEnv
                 syn outcome    : (Errs inh.finEnv) ⊎ (Target inh.finEnv)
```

The dependencies between attributes in the interface must be acyclic. Furthermore, there may not be references to attributes in later visits. That ensures that we can generate a type signature for a coroutine with this interface.

As additional work, we need to construct the proofs for the attribute *gathInFin*, and membership proofs for constructors *use* and *def*. At the root, we prefix the initial environment with the gathered environment. We get the required proof via *subRight*.

```
datasem Root
  prod root
    root.finEnv    = lhs.initEnv ++ root.gathEnv
    root.gathInFin = subRight
    lhs.outcome    = outcome root.outcome
```

For production *use*, we get both of the proofs we desire from the membership test of the name in the environment. For *def*, we have to do a bit more work. The gathered environment for this case is a singleton environment, so we get the proof for this environment via *here*. Since we also get a proof as *lhs.gathInFin* that the gathered environment is a subsequence of the final environment, we get the desired result via lemma *inSubset*. Finally, for the '*diam*' production, we only have to consider the possible outcomes of the two children.

```
datasem Source
  default finEnv  = last
  default gathEnv = concat
  prod use
    lhs.outcome  with loc.nm ∈? lhs.finEnv
                 | inj₁ notIn = inj₁ [scope loc.nm notIn]
```

```
                | inj2 isIn   = inj2 (use loc.nm isIn)
prod def
  loc.inFin       = inSubset lhs.gathInFin here
  lhs.outcome     = inj2 (def loc.nm loc.inFin)
prod _ ◇ _
  left.gathInFin  = trans subLeft lhs.gathInFin
  right.gathInFin = trans (subRight {syn.lhs.gathEnv} {lhs.finEnv})
                          lhs.gathInFin
  lhs.outcome     with left.outcome
                  | inj1 es with right.outcome
                  | inj1 es1 | inj1 es2 = inj1 (es1 ++ es2)
                  | inj1 es1 | inj2 _   = inj1 es1
                  | inj2 t1  with left.outcome
                  | inj2 t1  | inj1 es1 = inj1 es1
                  | inj2 t1  | inj2 t2  = inj2 (t1 ◇ t2)
```

The above code shows again an advantage of using Attribute Grammars: we can easily add additional predicates and write separate rules for them. However, this example also shows a problem: we merged what used to be two attributes into a single attribute *outcome*. In the extended version of this paper, [10], we show that the gradularity of attributes is important, and how we can circumvent merging the attributes.

3 RULER-FRONT and RULER-CORE

In previous work [12, 11], we showed that the syntax of RULER-FRONT is syntactic sugar for a more general core language RULER-CORE. In RULER-CORE, the AG is prepared in such a way that it is suitable for code generation: default-rules are applied, rules are ordered explicitly, and invocations of visits of children are explicit. This process remains the same in a dependently-typed setting, so we focus in this section on (a dependently-typed) RULER-CORE.

Productions in RULER-FRONT are translated to an algebra in RULER-CORE for the grammar involved. For a production, the corresponding semantic function takes the semantic value of the children, and returns the semantic value of the production. For example, for **prod**◇ the translation to RULER-CORE is:

```
sem_seq lSem rSem = sem_Source_nt where    -- reference to the sem fun
  sem nt : Source   -- Agda smart constructor for ◇
    child left : Source = lSem      -- turn lSem into an AG child left
    child right : Source = rSem     -- turn rSem into an AG child right
    invoke analyze of left          -- run visit analyze on left
    invoke analyze of right         -- run visit analyze on right
    invoke translate of left
    invoke translate of right
    ...                             -- the actual rules defined in RULER-FRONT
```

Thus, in the translation to RULER-CORE, rules are added that define the children of the production, and declare visits on these. In this example, the rules still have to be ordered such that each rule is lexically situated before the rule that uses its results [12]. Operationally, these rules are executed in precisely that order. For example, an invoke-rule requires the inherited attributes for that child and visit to be defined, and defines the synthesized attributes for that child.

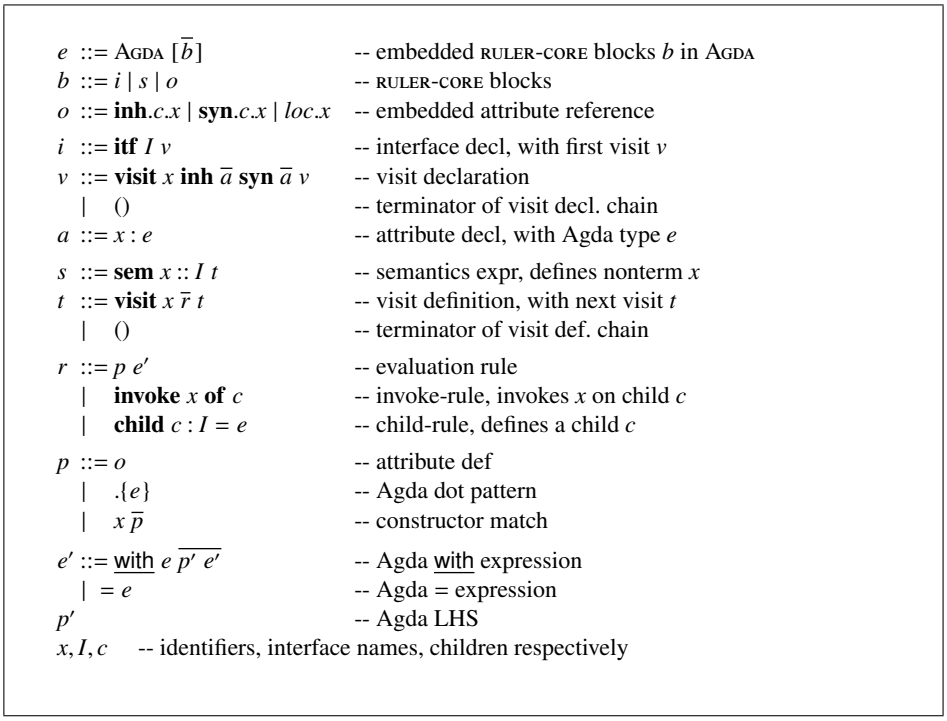

Fig. 1: Syntax of RULER-CORE

Figure 1 shows the syntax of the dependently-typed RULER-CORE (based on Agda). Effectively, a RULER-CORE program is Agda code with embedded RULER-CORE blocks, consisting of interface declarations, sem-blocks, and attribute references. The interface declares the attributes of visits. Note that the types of an attribute may refer to an attribute. The attributes must be ordered such that an attribute referenced in the type of attribute, occurs before the latter attribute. This requires the dependencies to be acyclic.

With a sem-block, we define the semantics of an AG-node given an interface I. The rules are specified per visit, in the order of evaluation. To define an attribute, we either use a with-expression when the value of the attribute is conditionally defined, or use a simple equation as RHS. In the translation to Agda, we plug such an expression in a function defined via with-expression, hence we need knowledge about the with-structure of the RHS.

As (denotational) semantics for RULER-CORE, we describe a translation of RULER-CORE programs to AGDA. Each semantics-block is translated to a coroutine, implemented as one-shot continuations. Each call to the coroutine corresponds with a visit. The parameters of the coroutine are the inherited attributes of the visit. The result of the call is an object containing values for the synthesized attributes, and the continuation to call for the visit.

Such a coroutine for a visit is a function that takes the inherited attributes as parameter, and produces a (nested) dependent product (Σ) containing the synthesized attributes for that visit, and the function to be used as continuation. Attribute references are transcribed to Agda via the naming conventions as listed on the bottom of Figure 2. For example, the type of such a coroutine is:

```
T_Source_analyze = Σ (synAgathEnv : Source) T_Source_translate
T_Source_translate synAgathEnv = (inhAfinEnv : Env) →
  (inhAgathInFin : synAgathEnv ⊆ inhAfinEnv) →
  Σ (Errs inhAfinEnv ⊎ Target inhAfinEnv)
    (T_Source_sentinel synAgathEnv inhAfinEnv inhAgathInFin)
T_Source_sentinel synAgathEnv synAgathEnv inhAfinEnv inhAgathInFin synAoutcome = ()
```

We need dependent products in order to be able to refer to synthesized attributes in the type of an attribute. This is the reason why we require the attributes to be ordered. Also, this representation can be optimized a bit by only passing on attributes that are needed in the types of attributes of later visits.

The coroutine itself has the following structure: it defines nested continuation-functions. Each of these continuation functions takes the inherited attributes as parameter, consists of with-expressions that compute attributes and evaluates children (discussed later), and ends in a product of the synthesized attributes plus the continuation function. For example for 'diam':

```
prod_Source_seq : T_Source → T_Source → T_Source_Analyze
prod_Source_seq lSem rSem = sem_Source_nt where
  sem_Source_nt : T_Source
  sem_Source_nt = vis_Source_analyze where
    vis_Source_analyze : T_Source_Analyze
    vis_Source_analyze with ...
    ... = (lhsSgathEnv, vis_Source_translate) where
          vis_Source_translate : T_Source_Translate lhsSgathEnv
          vis_Source_Translate lhsIfinEnv lhsIgathInFin with ...
```

The with-sequence for a visit-function consists of the translation of child-rules, invoke-rules and evaluation rules. For example, for rule **child** *left* : *Source* = *lSem*, evaluation of the right-hand side gives us the function to invoke for the first visit, hence the translation is:

```
... with lSem
... | leftVanalyze with ...
```

For the rule **invoke** *translate* **of** *left*, we run the visit-function of the child applied to its inherited attributes, and match against the synthesized attributes to bring them in scope:

$$
\begin{array}{lll}
[\![\textbf{itf}\ I\ v]\!] & \leadsto [\![_{\mathrm{iv}}\ v]\!]_I^{\emptyset} \quad ; \quad [\![sig\ I]\!] = [\![sig\ I\ (name\ v)]\!] \\
[\![_{\mathrm{iv}}\ \textbf{visit}\ x\ \textbf{inh}\ \overline{a}\ \textbf{syn}\ \overline{b}\ v]\!]_I^{\overline{g}} & \leadsto [\![_{\mathrm{iv}}\ v]\!]_I^{\overline{g}+\!\!+\overline{a}+\!\!+\overline{b}} & \\
 & \quad [\![sig\ I\ x]\!] : [\![_{\mathrm{at}}\ g_1]\!] \to ... \to [\![_{\mathrm{at}}\ g_n]\!] \to Set & \\
 & \quad [\![sig\ I\ x]\!]\ \overline{[\![_{\mathrm{an}}\ g]\!]} = [\![_{\mathrm{a}}\ \textbf{inh}.a_1]\!] \to ... \to [\![_{\mathrm{a}}\ \textbf{inh}.a_n]\!] \to & \\
 & \qquad\qquad [\![typrod\ \overline{b}\ (sig\ I\ (name\ v))]\!] & \\
[\![_{\mathrm{iv}}\ ()]\!]_I & \leadsto [\![sig\ I\ \texttt{"sentinel"}]\!] = () & \\
[\![_{\mathrm{a}}\ x : e]\!] & \leadsto [\![atname\ x]\!] : [\![e]\!] & \\
[\![_{\mathrm{at}}\ x : e]\!] & \leadsto [\![e]\!] & \\
[\![_{\mathrm{an}}\ x : e]\!] & \leadsto [\![atname\ x]\!] & \\
[\![\textbf{sem}\ x : I\ t]\!] & \leadsto [\![nt\ I\ x]\!] : [\![sig\ I]\!] & \\
 & \quad [\![nt\ I\ x]\!] = [\![vis\ I\ (name\ t)]\!]\textbf{where}\ [\![_{\mathrm{ev}}\ t]\!]_I^{\emptyset} & \\
[\![_{\mathrm{ev}}\ \textbf{visit}\ x\ \overline{r}\ t]\!]_I^{\overline{g}} & \leadsto [\![vis\ I\ X]\!] : [\![sig\ I\ x]\!]\ \overline{[\![_{\mathrm{an}}\ g]\!]} & \\
 & \quad [\![vis\ I\ x]\!]\ [\![inhs\ I\ x]\!] = [\![_{\mathrm{r}}\ \overline{r}]\!]_{[\![cont]\!]} & \\
\quad cont & \leadsto\ = [\![valprod\ (syns\ I\ x)\ (vis\ I\ (name\ t))]\!] & \\
 & \quad \textbf{where}\ [\![_{\mathrm{ev}}\ t]\!]_I^{\overline{g}+\!\!+\overline{a}+\!\!+\overline{b}} & \\
[\![_{\mathrm{r}}\ \textbf{child}\ c : I = e]\!]_k & \leadsto \underline{\mathsf{with}}\ e \quad ... \mid [\![vis\ I\ (firstvisit\ I)]\!]\ [\![k]\!] & \\
[\![_{\mathrm{r}}\ \textbf{invoke}\ x\ \textbf{of}\ c]\!]_k & \leadsto \underline{\mathsf{with}}\ [\![vis\ (\textbf{itf}\ c)\ x]\!]\ [\![inhs\ (\textbf{itf}\ c)\ x]\!] & \\
 & \quad ... \mid (valprod\ (syns\ (\textbf{itf}\ c)\ x))\ [\![k]\!] & \\
[\![_{\mathrm{r}}\ p\ e']\!]_k & \leadsto [\![_{\mathrm{ep}}\ e']\!]_p^k & \\
[\![_{\mathrm{ep}}\ \underline{\mathsf{with}}\ e\ \overline{p'\ e'}]\!]_p^k & \leadsto \underline{\mathsf{with}}\ e \quad \overline{... \mid p'\ [\![_{\mathrm{r}}\ p\ e']\!]_k} & \\
[\![_{\mathrm{ep}} = e]\!]_p^k & \leadsto \underline{\mathsf{with}}\ e \quad ... \mid [\![p]\!]\ k &
\end{array}
$$

$$
\begin{array}{llll}
atref\ \textbf{inh}.c.x & = c\ \texttt{"I"}\ x & nt\ I\ x & = \texttt{"sem\_"}\ I\ \texttt{"\_"}\ x \\
atref\ \textbf{syn}.c.x & = c\ \texttt{"S"}\ x & vis\ I\ c & = \texttt{"vis\_"}\ I\ \texttt{"\_"}\ c \\
atref\ loc.x & = \texttt{"locL"}\ x & vis\ c\ x & = c\ \texttt{"V"}\ x \\
atname\ \textbf{inh}.x & = \texttt{"inhAx"} & sig\ I & = \texttt{"T\_"}\ I \\
atname\ \textbf{syn}.x & = \texttt{"synAx"} & sig\ I\ x & = \texttt{"T\_"}\ I\ \texttt{"\_"}\ x
\end{array}
$$

Fig. 2: Denotational semantics of RULER-CORE

... $\underline{\mathsf{with}}$ *leftVtranslate leftIfinEnv leftIgathInFin*
... | (*leftSoutcome*, *leftVsentinel*) $\underline{\mathsf{with}}$ ...

The name *sentinel* represents a built-in final visit. It is never invoked, but having one simplifies the translation. Finally, for evaluation rules (e.g. *lhs.outcome* $\underline{\mathsf{with}}$...), we take the with-expression of the RHS, but move each RHS of an alternative (e.g. $inj_1\ (es_1 +\!\!+ es_2)$) into a with-expression that matches against the LHS of the evaluation rule:

... $\underline{\mathsf{with}}$ *leftSoutcome*
... | $inj_1\ es$ $\underline{\mathsf{with}}$ *rightSoutcome*
... | $inj_1\ es_1$ | $inj_1\ es_2$ $\underline{\mathsf{with}}$ $inj_1\ (es_1 +\!\!+ es_2)$
... | $inj_1\ es_1$ | $inj_1\ es$ | *lhsSoutcome* $\underline{\mathsf{with}}$ ...

With the above procedure, a visit function is a large nested with-expression that encodes precisely the order of evaluation.

Figure 2 formalizes the above translations. As usual, the semantic brackets indicate an escape from the syntactic to semantics level. The optional subscript on the left bracket disambiguates the intended translation. Both the $[\![_{\mathrm{iv}}\ \cdot]\!]$ and $[\![_{\mathrm{ev}}\ \cdot]\!]$ translations are parameterized over $\overline{g}$, the attributes defined lexically before this visit. The $[\![_{\mathrm{r}}\ \cdot]\!]$ translation of a rule takes a parameter k, representing the with-structure that follows up on that rule (which ends with the dependent product of synthesized attributes). The same technique we use for the translation of a with-rhs using $[\![_{\mathrm{ep}}\ \cdot]\!]$.

4 Related Work

Dependent types originate in Martin-Löf's Type Theory. Various dependently-typed programming languages are gaining popularity, including Agda [13], Epigram [9], and Coq [1]. We present the ideas in this paper with Agda as host language, because it has a concept of a dependent pattern match, which corresponds conveniently with pattern matching in AGs. Also, in Coq and Epigram, a program is written via interactive theorem proving with tactics or commands. The preprocessor-based approach of this paper, however, suits a declarative approach more.

Attribute grammars [6, 7] were considered to be a promising implementation for compiler construction. Recently, many Attribute Grammar systems arose for mainstream languages, such as Silver [17] and JastAdd [3] for Java, and UUAG [15] for Haskell. To our knowledge, Attribute Grammars with more sexy types have not been investigated yet.

In certain languages it is possible to implement AGs via meta-programming facilities, which obliviates the need of a preprocessor. Viera, et al. [16] show how to implement AGs into Haskell through type level programming. A combination of that paper with our work, would fit well in Agda, if Agda had a mechanism similar to Haskell's class system.

Several attribute grammar techniques are important to our work. Kastens [5] introduces ordered attribute grammars. In OAGs, the evaluation order of attribute computations as well as attribute lifetime can be determined statically, which allows us to generate coroutines such that the circularity checker accepts the program as terminating.

5 Conclusion

We investigated Attribute Grammars with dependently-typed attributes: the type of an attribute may refer to the value of another attribute. This feature allows us to conveniently encode invariants of attributes, and pass proofs of these invariants around as attributes, as presented in the use-case in Section 2.

We introduced a language for Ordered AGs, RULER-FRONT, in Section 3 and showed a translation to Agda (preprocessor-based). The requirement on order is important. It allows us to prove termination in Agda, and generate continuation-based functions from

AGs that require more than one visit to compute the attributes. These are tedious functions to write manually. The preprocessor-based approach saves us from writing such functions ourselves.

Acknowledgments. This work was supported by Microsoft Research through its European PhD Scholarship Programme.

References

1. Bertot, Y.: A short presentation of coq. In: Mohamed, O.A., Muñoz, C., Tahar, S. (eds.) TPHOLs. Lecture Notes in Computer Science, vol. 5170, pp. 12–16. Springer (2008)
2. Dijkstra, A., Fokker, J., Swierstra, S.D.: The architecture of the Utrecht Haskell compiler. In: Haskell '09: Proceedings of the 2nd ACM SIGPLAN symposium on Haskell. pp. 93–104. ACM, New York, NY, USA (2009)
3. Ekman, T., Hedin, G.: The JastAdd Extensible Java Compiler. In: Gabriel, R.P., Bacon, D.F., Lopes, C.V., Jr., G.L.S. (eds.) OOPSLA. pp. 1–18. ACM (2007)
4. Heeren, B., Leijen, D., van IJzendoorn, A.: Helium, for Learning Haskell. In: ACM SIGPLAN Haskell Workshop (HW'03). pp. 62 – 71. ACM Press, New York (Sep 2003)
5. Kastens, U.: Ordered Attributed Grammars. Acta Inf. 13, 229–256 (1980)
6. Knuth, D.E.: Semantics of Context-Free Languages. Math. Sys. Theory 2(2), 127–145 (1968)
7. Knuth, D.E.: The Genesis of Attribute Grammars. In: WAGA. pp. 1–12 (1990)
8. Leroy, X.: Formal verification of a realistic compiler. Commun. ACM 52(7), 107–115 (2009)
9. McBride, C.: Epigram: Practical programming with dependent types. In: Vene, V., Uustalu, T. (eds.) Advanced Functional Programming. Lecture Notes in Computer Science, vol. 3622, pp. 130–170. Springer (2004)
10. Middelkoop, A., Dijkstra, A., Swierstra, S.D.: Dependently-typed Attribute Grammars (extended version). http://people.cs.uu.nl/ariem/ifl10-ex.pdf (2010)
11. Middelkoop, A., Dijkstra, A., Swierstra, S.D.: Iterative Type Inference with Attribute Grammars. In: Proceedings of the International Conference on Generative Programming and Component Engineering (2010)
12. Middelkoop, A., Dijkstra, A., Swierstra, S.D.: Side-effectful Attribute Grammars. http://people.cs.uu.nl/ariem/wgt10-sideeffect.pdf (2010)
13. Norell, U.: Dependently typed programming in agda. In: Kennedy, A., Ahmed, A. (eds.) TLDI. pp. 1–2. ACM (2009)
14. Schrage, M.M., Jeuring, J.T.: Proxima - A presentation-oriented editor for structured documents (2004)
15. Universiteit Utrecht: Universiteit Utrecht Attribute Grammar System. http://www.cs.uu.nl/wiki/HUT/AttributeGrammarSystem
16. Viera, M., Swierstra, S.D., Swierstra, W.: Attribute grammars fly first-class: how to do aspect oriented programming in Haskell. In: Hutton, G., Tolmach, A.P. (eds.) ICFP. pp. 245–256. ACM (2009)
17. Wyk, E.V., Bodin, D., Gao, J., Krishnan, L.: Silver: an Extensible Attribute Grammar System. Electr. Notes Theor. Comput. Sci. 203(2), 103–116 (2008)