Dependently Typed Attribute Grammars

Arie Middelkoop, Atze Dijkstra, and S. Doaitse Swierstra

Universiteit Utrecht,
The Netherlands

Abstract. Attribute Grammars (AGs) are a domain-specific language for functional and composable descriptions of tree traversals. Given such a description, it is not immediately clear how to state and prove properties of AGs formally. To meet this challenge, we apply dependent types to AGs. In a dependently typed AG, the type of an attribute may refer to values of attributes. The type of an attribute is an invariant, the value of an attribute a proof for that invariant. Additionally, when an AG is cycle-free, the composition of the attributes is logically consistent. We present a lightweight approach using a preprocessor in combination with the dependently typed language Agda.

1 Introduction

Functional programming languages are known to be convenient languages for implementing a compiler. As part of the compilation process, a compiler computes properties of Abstract Syntax Trees (ASTs), such as environments, types, error messages, and code. In functional programming, these syntax-directed computations are typically written as *catamorphisms*[1]. An *algebra* defines an inductive property in terms of each constructor of the AST, and a catamorphism applies the algebra to the AST. Catamorphisms thus play an important role in a functional implementation of a compiler.

Attribute Grammars (AGs) [3] are a domain-specific language for *composable* descriptions of catamorphisms. AGs facilitate the description of complex catamorphisms that typically occur in complex compiler implementations.

An AG extends a context-free grammar by associating *attributes* with nonterminals. Functional *rules* are associated with productions, and define values for the attributes that occur in the nonterminals of associated productions. As AGs are typically embedded in a host language, the rules are terms in the host language, which may additionally refer to attributes. Attributes can easily be composed to form more complex properties. An AG can be compiled to an efficient functional algorithm that computes the synthesized attributes of the root of the AST, given the root's inherited attributes.

It is not immediately clear how to formally specify and write proofs about programs implemented with AGs. *Dependent types* [1] provide a means to use *types* to encode

[1] Catamorphisms are a generalization of folds to tree-like data structures. We consider catamorphisms from the perspective of algebraic data types in functional programming instead of the equivalent notion in terms of functors in category theory. A catamorphism $cata_\tau\ (f_1, ..., f_n)$ replaces each occurrence of a constructor c_i of τ in a data structure with f_i. The product $(f_1, ..., f_n)$ is called an *algebra*. An element f_i of the algebra is called a *semantic function*.

properties with the expressiveness of (higher-order) intuitionistic propositional logic, and *terms* to encode proofs. Such programs are called correct by construction, because the program itself is a proof of its invariants. The goal of this paper is therefore to apply dependent types to AGs, in order to formally reason with AGs.

Vice versa, AGs also offer benefits to dependently typed programming. Because of the Curry-Howard correspondence, dependently typed AGs are a domain-specific language to write structurally inductive proofs in a *composable*, *aspect-oriented* fashion; each attribute represents a separate aspect of the proof. Additionally, AGs alleviate the programmer from the tedious orchestration of multi-pass traversals over data structures, and ensure that the traversals are *total*: totality is required for dependently typed programs for reasons of logical consistency and termination of type checking. Hence, the combination of dependent types and AGs is mutually beneficial.

We make the following contributions in this paper:

- We present the language $\mathrm{AG_{DA}}$ (Section 3), a light-weight approach to facilitate dependent types in AGs, and vice versa, AGs in the dependently typed language Agda. $\mathrm{AG_{DA}}$ is an embedding in Agda via a preprocessor.
  In contrast to conventional AGs, we can encode invariants in terms of dependently typed attributes, and proofs as values for attributes. This expressiveness comes at a price: to be able to compile to a total Agda program, we restrict ourselves to the class of ordered AGs, and demand the definitions of attributes to be total.
- We define a desugared version of $\mathrm{AG_{DA}}$ programs (Section 4) and show how to translate them to plain Agda programs (Section 5).
- Our approach supports a conditional attribution of nonterminals, so that we can give total definitions of what would otherwise be partially defined attributes (Section 6).

In Section 2, we introduce the notation used in this paper. However, we assume that the reader is both familiar with AGs (see [10]) and dependently typed programming in Agda (see [7]).

2 Preliminaries

In this warm-up section, we briefly touch the Agda and AG notation used throughout this paper. As an example, we implement the sum of a list of numbers with a catamorphism. We give two implementations: first one that uses plain Agda, then another using $\mathrm{AG_{DA}}$. This example does not yet use dependently typed attributes. These are introduced in the next section.

In the following code snippet, the data type *List* represents a cons-list of natural numbers. The type $T'List$ is the type of the value we compute (a number), and $A'List$ is the type of an algebra for *List*. Such an algebra contains a *semantic function* for each constructor of *List*, which transforms a value of that constructor into the desired value (of type $T'List$), assuming that the transformation has been recursively applied to the fields of the constructor. The catamorphism $cata_{List}$ performs the recursive application.

```
data List : Set where        -- represents a cons-list of natural numbers
  nil  : List                -- constructor has no fields
```

```
    cons : ℕ → List → List    -- constructor has a number and tail list as fields
  T′List = ℕ                              -- defines a type alias T′List : Set,
  A′List = (T′List, ℕ → T′List → T′List)  -- and A′List : Set
  cata_List  : A′List → List → T′List     -- applies algebra to list
  cata_List (n, _) nil    = n             -- in case of nil, replaces nil with n
  cata_List alg l   with alg  |  l        -- otherwise, matches on alg and l
  cata_List alg l   | (_, c) | cons x xs   with cata_List alg xs   -- recurses on xs
  cata_List alg l   | (_, c) | cons x xs   | r    = c x r          -- replaces cons with c
```

In Agda, a function is defined by one or more equations. A with-construct facilitates pattern matching against intermediate values. An equation that ends with <u>with</u> e_1 | ... | e_n parameterizes the equations that follow with the values of $e_1, ..., e_n$ as additional arguments. Vertical bars separate the patterns intended for the additional parameters.

The actual algebra itself simply takes 0 for the *nil* constructor, and $\_+\_$ for the *cons* constructor. The function sum_{List} shows how the algebra and catamorphism can be used.

```
  sem_nil   : T′List                       -- semantic function for nil constructor
  sem_nil   = 0                            -- T′List = ℕ (defined above)
  sem_cons : ℕ → T′List → T′List           -- semantic function for cons constructor
  sem_cons = _+_                           -- _+_ : ℕ → ℕ → ℕ (defined in library)
  sum_List : List → T′List                 -- transforms the List into the desired sum
  sum_List = cata_List (sem_nil, sem_cons) -- algebra is semantic functions in a tuple
```

In the example, the sum is defined in a bottom-up fashion. By taking a function type for *T′List*, values can also be passed top-down. Multiple types can be combined by using products. Such algebras quickly become tedious to write. Fortunately, we can use AGs as a domain-specific language for algebras. In the code below, we give an AG implementation: we specify a grammar that describes the structure of the AST, declare attributes on productions, and give rules that define attributes.

We now give an implementation of the same example using $\mathrm{AG_{DA}}$. The code consists of blocks of plain Agda code, and blocks of AG code. To ease the distinction, Agda's keywords are underlined, and keywords of $\mathrm{AG_{DA}}$ are typeset in bold.

A grammar specification is a restricted form of a data declaration (for an AST): data constructors are called *productions* and their fields are explicitly marked as *terminal* or *nonterminal*. A nonterminal field represents a *child* in the AST and has attributes, whereas a terminal field only has a value. A plain Agda data-type declaration can be derived from a grammar specification. In such a specification, nonterminal types must have a fully saturated, outermost type constructor that is explicitly introduced by a grammar declaration. Terminal types may be arbitrary Agda types[2].

```
  grammar List : Set    -- declares nonterminal List of type Set
    prod nil   : List   -- production nil of type List (no fields)
    prod cons : List    -- production cons of type List (two fields)
```

[2] In general, although not needed in this example, nonterminal types may be parametrized, production types may refer to its field names, and field types may refer to preceding field names.

```
term      hd : ℕ     -- terminal field hd of type ℕ
nonterm   tl  : List  -- nonterminal field tl of type List
```

With an interface specification, we declare attributes for nonterminals. Attributes come in two fashions: *inherited* attributes (used in a later example) must be defined by rules of the parent, and *synthesized* attributes may be used by the parent. Names of inherited attributes are distinct from names of synthesized attributes; an attribute of the same name and fashion may only be declared once per nonterminal. We also partition the attributes in one or more *visits*. These visits impose a partial order on attributes. Inherited attributes may not be defined in terms of a synthesized attributes of the same visit or later. We use this order in Section 4 to derive semantic functions that are total.

```
itf List              -- interface for nonterminal List,
  visit compute       -- with a single visit that is named compute,
    syn sum : ℕ       -- and a synthesized attribute named sum of type ℕ
```

Finally, we define each of the production's attributes. We may refer to an attribute using *child.attr* notation. For each production, we give rules that define the inherited attributes of the children and synthesized attributes of the production itself (with *lhs* as special name), using inherited attributes of the production and synthesized attributes of the children. The special name *loc* refers to the terminals, and to local attributes that we may associate with a production.

```
datasem List    -- defines attributes of List for constructors of List
  prod nil    lhs.sum = 0                  -- rule for sum of production nil
  prod cons   lhs.sum = loc.hd + tl.sum    -- refers to terminal hd and attr tl.sum
```

The left-hand side of a rule is a plain Agda pattern, and the right-hand side is either a plain Agda expression or with-construct (not shown in this example). Additionally, both the left and right-hand sides may contain attribute references.

During attribute evaluation, visits are performed on children to obtain their associated synthesized attributes. We do not have to explicitly specify when to visit these children, neither is the order of appearance of rules relevant. However, an inherited attribute *c.x* may not depend on a synthesized attribute *c.y* of the same visit or later (in the interface). This guarantees that the attribute dependencies are acyclic, so that we can derive when children need to be visited and in what order.

AGs are a domain-specific language to write algebras in terms of attributes. From the grammar, we generate the data type and catamorphism. From the interface, we generate the $T'List$ type. From the rules, we generate the semantic functions sem_{nil} and sem_{cons}. AGs pay off when an algebra has many inherited and synthesized attributes. Also, there are many AG extensions that offer abstractions over common usage patterns (not covered in this paper). In the next section we present AGs with dependent types, so that we can formulate properties of attributes (and their proofs).

3 Dependently Typed Example

In this section, we use AG_{DA} to implement a mini-compiler that performs name checking of a simple language *Source*, and translates it to target language *Target* if all used identifiers are declared, or produces errors otherwise. A term in *Source* is a sequence of identifier definitions and identifier uses, for example: $def\ a \diamond use\ b \diamond use\ a$. In this case, b is not defined, thus the mini-compiler reports an error. Otherwise, it generates a *Target* term, which is a clone of the *Source* term that additionally carries evidence that the term is free of naming errors. Section 3.2 shows the definition of both *Source* and *Target*.

We show how to prove that the mini-compiler produces only correctly named *Target* terms and errors messages that only mention undeclared identifiers. The proofs are part of the implementation's code. Name checking is only a minor task in a compiler. However, the example shows many aspects of a more realistic compiler.

3.1 Support Code Dealing With Environments

We need some Agda support code to deal with environments. We show the relevant data structures and type signatures for operations on them, but omit the actual implementation. We represent the environment as a cons-list of identifiers.

```
Ident = String       -- Ident : Set
Env   = List Ident   -- Env : Set
```

In intuitionistic type theory, a data type represents a relation, its data constructors deduction rules for such a relation, and values built using these constructors are proofs for instances of the relation. We use some data types to reason with environments.

A value of type $\iota \in \Gamma$ is a proof that an identifier ι is member of an environment Γ. A value *here* indicates that identifier is at the front of the environment. A value *next* means that the identifier can be found in the tail of the environment, as described by the remainder of the proof.

```
data _∈_ : Ident → Env → Set where
  here : {ι : Ident} {Γ : Env} → ι ∈ (ι :: Γ)
  next : {ι₁ : Ident} {ι₂ : Ident} {Γ : Env} → ι₁ ∈ Γ → ι₁ ∈ (ι₂ :: Γ)
```

The type $\Gamma_1 \sqsubseteq \Gamma_2$ represents a proof that an environment Γ_1 is contained as a subsequence of an environment Γ_2. A value *subLeft* means that the environment Γ_1 is a prefix of Γ_2, and *subRight* means that Γ_1 is a suffix. With *trans*, we transitively compose two proofs.

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

The following functions operate on proofs. When an identifier occurs in an environment, function *inSubset* produces a proof that the identifier is also in the superset of the environment. Given an identifier and an environment, $\iota \in_? \Gamma$ returns either a proof $\iota \in \Gamma$ that the element is in the environment, or a proof that it is not.

$$inSubset : \{\iota : Ident\}\ \{\Gamma_1 : Env\}\ \{\Gamma_2 : Env\} \to \Gamma_1 \sqsubseteq \Gamma_2 \to \iota \in \Gamma_1 \to \iota \in \Gamma_2$$
$$\_ \in_? \_ \quad : (\iota : Ident) \to (\Gamma : Env) \to \neg(\iota \in \Gamma) \uplus (\iota \in \Gamma)$$

A value of the sum-type $\alpha \uplus \beta$ either consists of an α wrapped in a constructor inj_1 or of a β wrapped in inj_2.

3.2 Grammar of the Source and Target Language

Below, we give a grammar for both the *Source* and *Target* language, such that we can analyze their ASTs with AGs[3]. The *Target* language is a clone of the *Source* language, except that terms that have identifiers carry a field *proof* that is evidence that the identifiers are properly introduced.

```
grammar Root      : Set          -- start symbol of grammar and root of AST
  prod root : Root    nonterm top : Source   -- top of the Source tree
grammar Source    : Set          -- grammar for nonterminal Source
  prod use        : Source       -- 'result type' of production
    term   ι      : Ident        -- terminals may have arbitrary Agda types
  prod def        : Source       -- 'result type' may be parametrized
    term   ι      : Ident
  prod _ ⋄ _      : Source       -- represents sequencing of two Source terms
    nonterm left  : Source       -- nonterminal fields must have a nonterm as
    nonterm right : Source       -- outermost type constructor.
grammar Target    : Env → Set    -- grammar for nonterminal Target
  prod def        : Target Γ     -- production type may refer to any field,
    term?  Γ      : Env          -- e.g. Γ. Agda feature: implicit terminal
    term   ι      : Ident        --   (inferred when building a def)
    term   φ      : ι ∈ Γ        -- field type may refer to preceding fields
  prod use        : Target Γ
    term?  Γ      : Env          -- a Target term carries evidence: a
    term   ι      : Ident        -- proof that the identifier is in the
    term   φ      : ι ∈ Γ        -- environment
  _ ⋄ _           : Target Γ
    term?  Γ      : Env
    nonterm left  : Target Γ     -- nonterm fields introduce children that
    nonterm right : Target Γ     -- have attributes
data Err : Env → Set where       -- data type for errors in Agda notation
  scope : {Γ : Env} (ι : Ident) → ¬(ι ∈ Γ) → Err Γ
Errs Γ = List (Err Γ)            -- Errs : Env → Set
```

As shown in Section 2, we generate Agda data-type definitions and catamorphisms from this specification.

[3] In our example, we could have defined the type *Target* instead using conventional Agda notation. However, the grammar for *Target* serves as an example of a parameterized nonterminal.

The concrete syntax of the source language *Source* and target language *Target* of the mini-compiler is out of scope for this paper; the grammar defines only the abstract syntax. Similarly, we omit a formal operational semantics for *Source* and *Target*: it evaluates to unit if there is an equally named *def* for every *use*, otherwise evaluation diverges.

3.3 Dependent Attributes

In this section, we define *dependently typed* attributes for *Source*. Such a type may contain references to preceding[4] attributes using *inh.attrNm* or *syn.attrNm* notation, which explicitly distinguishes between inherited and synthesized attributes. The type specifies a property of the attributes it references; an attribute with such a type represents a proof of this property.

In our mini-compiler, we compute bottom-up a synthesized attribute *gathEnv* that contains identifiers defined by the *Source* term. At the root, the *gathEnv* attribute contains all the defined identifiers. We output its value as the synthesized attribute *finEnv* (final environment) at the root. Also, we pass its value top-down as the inherited attribute *finEnv*, such that we can refer to this environment deeper down the AST. We also pass down an attribute *gathInFin* that represents a proof that the final environment is a superset of the gathered environment. When we know that an identifier is in the gathered environment, we can thus also find it in the final environment. We pass up the attribute *outcome*, which consists either of errors, or of a correct *Target* term.

```
itf Root          -- attributes for the root of the AST
  visit compile    syn finEnv    : Env
                   syn outcome   : (Errs syn.finEnv) ⊎ (Target syn.finEnv)
itf Source        -- attributes for Source
  visit analyze    syn gathEnv   : Env   -- attribute of first visit
  visit translate  inh finEnv    : Env   -- attributes of second visit
                   inh gathInFin : syn.gathEnv ⊑ inh.finEnv
                   syn outcome   : (Errs inh.finEnv) ⊎ (Target inh.finEnv)
itf Target Γ      -- interface for Target (parameterized) is not used in the example.
```

As we show later, at the root, we need the value of *gathEnv* to define *finEnv*. This requires *gathEnv* to be placed in a strict earlier visit. Hence we define two visits, ordered by appearance.

Attribute *gathInFin* has a dependent type: it specifies that *gathEnv* is a subsequence of *finEnv*. A value of this attribute is a proof that essentially states that we did not forget any identifiers. Similarly, in order to construct *Target* terms, we need to prove that *finEnv* defines the identifiers that occur in the term. In the next section, we construct such proofs by applying data constructors. We may use inherited attributes as *assumptions* and pattern matches against values of attributes as *case distinctions*. Thus, with a

[4] We may refer to an attribute that is declared earlier (in order of appearance) in the same interface. There is one exception due to the translation to Agda (Section 5): in the type of an inherited attribute, we may not refer to synthesized attributes of the same visit.

dependently typed AG we can formalize and prove correctness properties of our implementation. Agda's type checker validates such proofs using symbolic evaluation driven by unification.

3.4 Semantics of Attributes

For each production, we give definitions for the declared attributes via rules. At the root, we pass the gathered environment back down as final environment. Thus, these two attributes are equal, and we can trivially prove that the final environment is a subsequence using either *subRight* or *subLeft*.

```
datasem Root prod root              -- rules for production root of nonterm Root
  top.finEnv     = top.gathEnv      -- pass gathered environment down
  top.gathInFin  = subRight {[ ]}   -- subsequence proof, using: [ ] ++ Γ4 ≡ Γ4
  lhs.finEnv     = top.gathEnv      -- pass gathEnv up
  lhs.outcome    = top.outcome      -- pass outcome up
```

For the *use*-production of *Source*, we check if the identifier (terminal *loc.ι*) is in the environment. If it is, we produce a *Target* term as value for the outcome attribute, otherwise we produce a *scope* error. For *def*, we introduce an identifier in the gathered environment. No errors can arise, hence we always produce a *Target* term. We prove ($loc.\phi_1$) that the identifier *loc.ι* is actually in the gathered environment, and prove ($loc.\phi_2$) using *inSubset* and attribute *lhs.gathInFin* that it must also be in the final environment. For _ ⋄ _, we pass *finEnv* down to both children, concatenate their *gathEnv*s, and combine their *outcome*s.

```
datasem Source                           -- rules for productions of Source
  prod use
    lhs.gathEnv   = [ ]                                  -- no names introduced
    lhs.outcome   with loc.ι ∈? lhs.finEnv               -- tests presence of ι
                  | inj1 notIn = inj1 [scope loc.ι notIn]   -- when not in env
                  | inj2 isIn  = inj2 (use loc.ι isIn)      -- when in env
  prod def
    lhs.gathEnv   = [loc.ι]                              -- one name introduced
    loc.φ1        = here {loc.ι} {syn.lhs.gathEnv}       -- proof of ι in gathEnv
    loc.φ2        = inSubset lhs.gathInFin loc.φ1        -- proof of ι in finEnv
    lhs.outcome   = inj2 (def loc.ι loc.φ2)              -- never any errors
  prod _ ⋄ _
    lhs.gathEnv   = left.gathEnv ++ right.gathEnv        -- pass names up
    left.finEnv   = lhs.finEnv                           -- pass finEnv down
    right.finEnv  = lhs.finEnv                           -- pass finEnv down
    left.gathInFin  = trans subLeft lhs.gathInFin        -- proof for left
    right.gathInFin = trans (subRight {syn.lhs.gathEnv} {lhs.finEnv})
                            lhs.gathInFin                -- proof for right
    lhs.outcome   with left.outcome                         -- four alts.
                  | inj1 es with right.outcome
```

```
| inj₁ es₁ | inj₁ es₂ = inj₁ (es₁ ++ es₂)     -- 1: both in error
| inj₁ es₁ | inj₂ _   = inj₁ es₁             -- 2: only left
| inj₂ t₁  with left.outcome
| inj₂ t₁  | inj₁ es₂ = inj₁ es₂             -- 3: only right
| inj₂ t₁  | inj₂ t₂  = inj₂ (t₁ ⋄ t₂)       -- 4: none in error
```

Out of the above code, we generate each production's semantic function (and some wrapper code), such that these together with a catamorphism form a function that translates *Source* terms. The advantage of using AGs here is that we can easily add more attributes (and thus more properties and proofs) and refer to them.

4 AG Descriptions and their Core Representation

In the previous sections, we presented $\mathrm{AG_{DA}}$ (by example). To describe the dependently-typed extension to AGs, we do so in terms of the core language $\mathrm{AG^{X}_{DA}}$ (a subset of $\mathrm{AG_{DA}}$). Implicit information in AG descriptions (notational conveniences, the order of rules, visits to children) is made explicit in $\mathrm{AG^{X}_{DA}}$. We sketch the translation from $\mathrm{AG_{DA}}$ to $\mathrm{AG^{X}_{DA}}$. In previous work [4, 5], we described the process in more detail (albeit in a non-dependently typed setting).

$\mathrm{AG^{X}_{DA}}$ contains interface declarations, but grammar declarations are absent and semantic blocks encoded differently. Each production in $\mathrm{AG_{DA}}$ is mapped to a *semantic function* in $\mathrm{AG^{X}_{DA}}$: it is a domain-specific language for the contents of semantic functions. A terminal $x : \tau$ of the production is mapped to a parameter $loc_l x : \tau$. Implicit terminals are mapped to implicit parameters. A nonterminal $x : N\ \overline{\tau}$ is mapped to a parameter $loc_c x : T'N\ \overline{\tau}$. The body of the production consists of the rules for the production given in the original $\mathrm{AG^{X}_{DA}}$ description, plus a number of additional rules that declare children and their visits explicitly.

```
sem⋄ : T′Source → T′Source → T′Source   -- derived from (non)terminal types
sem⋄ loc_c left loc_c right =           -- semantic function for ⋄
  sem : Source                          -- AG^X_DA semantics block
    child left : Source  = loc_c left   -- defines a child left
    child right : Source = loc_c right  -- defines a child right
    invoke analyze   of left            -- rule requires visiting analyze on left
    invoke analyze   of right           -- rule requires visiting analyze on right
    invoke translate of left
    invoke translate of right
    lhs.gathEnv = left.gathEnv ++ right.gathEnv   -- the AG_DA rules
    ...                                           -- etc.
```

A child rule introduces a child with explicitly semantics (a value of the type $T'Source$). Other rules may declare visits and refer to the attributes of the child. An invoke rule declares a visit to a child, and brings the attributes of that visit in scope. Conventional rules define attributes, and may refer to attributes. The dependencies between attributes induces a def-use (partial) order.

Actually, there is one more step to go to end up with a $\mathrm{AG}^{\mathrm{X}}_{\mathrm{DA}}$ description. A semantics block consists of one of more visit-blocks (in the order specified by the interface), and the rules are partitioned over the blocks. In a block, the *lhs* attributes of that and earlier visits are in scope, as well as those brought in scope by preceding rules. Also, the synthesized attributes of the visit must be defined in the block or in an earlier block. We assign rules to the earliest block that satisfies the def-use order. We convert this partial order into a total order by giving conventional rules precedence over child/invoke rules, and using the order of appearance otherwise:

```
sem◇ : T'Source → T'Source → T'Source      -- signature derived from itf
sem◇ loc_c left loc_c right =              -- semantic function for ◇
  sem : Source                             -- AG^X_DA block
    visit analyze                          -- first visit
      child left : Source  = loc_c left    -- defines a child left
      invoke analyze  of left              -- requires child to be defined
      child right : Source = loc_c right   -- defines a child right
      invoke analyze  of right             -- requires child to be defined
      syn.lhs.gathEnv = syn.left.gathEnv ++ syn.right.gathEnv
    visit translate                        -- second visit
      inh.left.finEnv      = inh.lhs.finEnv  -- needs lhs.finEnv
      inh.right.finEnv     = inh.lhs.finEnv  -- needs lhs.finEnv
      inh.left.gathInFin   = trans ...       -- also needs lhs.gathEnv
      inh.right.gathInFin  = trans ...       -- also needslhs.gathEnv
      invoke translate of left               -- needs def of inh attrs of left
      invoke translate of right              -- needs def of inh attrs of right
      syn.lhs.outcome   with ...             -- needs translate attrs of children
```

It is a static error when such an order cannot be satisfied. Another interesting example is the semantic function for the root: it has a child with a different interface as itself, and has two invoke rules in the same visit.

```
sem_root : T'Source → T'Root               -- semantic function for the root
sem_root locStop =                         -- Source's semantics as parameter
  sem : Root visit compile                 -- only one visit
    child top : Source = loc_c top         -- defines a child top
    invoke analyze of top                  -- invokes first visit of top
    inh.top.finEnv     = syn.top.gathEnv   -- passes gathered environment back
    invoke translate of top                -- invokes second visit of top
    syn.lhs.output     = syn.top.gathEnv   -- passes up the gathered env
    syn.lhs.output     = syn.top.outcome   -- passes up the result
```

Figure 1 shows the syntax of $\mathrm{AG}^{\mathrm{X}}_{\mathrm{DA}}$. In general, interfaces may be parametrized. The interface has a function type τ (equal to the type of the nonterminal declaration in $\mathrm{AG}_{\mathrm{DA}}$) that specifies the type of each parameter, and the kind of the interface (an upper bound of the kinds of the parameters). For an evaluation rule, we either use a with-expression when the value of the attribute is conditionally defined, or use a simple

$e \ ::= \textsc{Agda}\ [\overline{b}]$	-- embedded blocks b in Agda
$b \ ::= i \mid s \mid o$	-- $\text{AG}^{\text{X}}_{\text{DA}}$ blocks
$o \ ::= inh.c.x \mid syn.c.x \mid loc.x$	-- embedded attribute reference
$i \ ::= \textbf{itf}\ I\ \overline{x} : \tau\ v$	-- with first visit v, params x, and signature τ
$v \ ::= \textbf{visit}\ x\ \textbf{inh}\ \overline{a}\ \textbf{syn}\ \overline{a}\ v$	-- visit declaration
$\quad \mid \ \square$	-- terminator of visit decl. chain
$a \ ::= x : e$	-- attribute decl, with Agda type e
$s \ ::= \textbf{sem} : I\ \overline{e}\ t$	-- semantics expr, uses interface $I\ \overline{e}$
$t \ ::= \textbf{visit}\ x\ \overline{r}\ t$	-- visit definition, with next visit t
$\quad \mid \ \square$	-- terminator of visit def. chain
$r \ ::= p\ e'$	-- evaluation rule
$\quad \mid \ \textbf{invoke}\ x\ \textbf{of}\ c$	-- invoke-rule, invokes x on child c
$\quad \mid \ \textbf{child}\ c : I = e$	-- child-rule, defines a child c, with interface $I\ \overline{e}$
$p \ ::= o$	-- attribute def
$\quad \mid \ .\{e\}$	-- Agda dot pattern
$\quad \mid \ x\ \overline{p}$	-- constructor match
$e' ::= \underline{\mathsf{with}}\ e\ \overline{p'\ e'^{?}}$	-- Agda $\underline{\mathsf{with}}$ expression (e' absent when p' absurd)
$\quad \mid \ = e$	-- Agda = expression
p'	-- Agda LHS
x, I, c	-- identifiers, interface names, children respectively
τ	-- plain Agda type

Fig. 1: Syntax of RULER-CORE

equation as RHS. In the next section, we plug such an expression in a function defined via with-expressions, hence we need knowledge about the with-structure of the RHS.

5 Translation to Agda

To explain the preprocessing of $\text{AG}^{\text{X}}_{\text{DA}}$ to Agda, we give a translation scheme in Figure 2 (explained via examples below). This translation scheme is a denotational semantics for $\text{AG}^{\text{X}}_{\text{DA}}$. Also, if the translation is correct Agda, then the original is correct $\text{AG}^{\text{X}}_{\text{DA}}$.

A semantics block in a $\text{AG}^{\text{X}}_{\text{DA}}$ program is actually an algorithm that makes precise how to compute the attributes as specified by the interface: for each visit, the rules prescribe when to compute an attribute and when to visit a child. The idea is that we map such a block to an Agda function that takes values for its inherited attributes and delivers a dependent product[5] of synthesized attributes. However, such a function would be cyclic: in the presented example, the result *gathEnv* would be needed for as input for *finEnv*. Fortunately, we can bypass this problem: we map to a k-visit *coroutine* instead.

[5] A dependent product $\Sigma\ \tau\ f = (\tau, f\ \tau)$ parameterizes the RHS f with the LHS τ.

A coroutine is a function that can be invoked k times. We associate each invocation with a visit of the interface. Values for the inherited attributes are inputs to the invocation. Values for the synthesized attributes are the result of the invocation. In a pure functional language (like Agda), we can encode coroutines as one-shot continuations (or *visit functions* [8]).

$$
\begin{array}{lll}
\llbracket \mathbf{itf}\ I\ \overline{x} : \overline{\tau_x} \to \tau \rrbracket\ v & \rightsquigarrow \llbracket_{\mathsf{iv}}\ v \rrbracket_{I,\tau}^{\overline{x:\tau_x}} \quad ; \quad \llbracket \mathit{sig}\ I \rrbracket : \llbracket \tau \rrbracket \quad ; \quad \llbracket \mathit{sig}\ I \rrbracket = \llbracket \mathit{sig}\ I\ (\mathit{name}\ v) \rrbracket & \\
\llbracket_{\mathsf{iv}}\ \mathbf{visit}\ x\ \mathbf{inh}\ \overline{a}\ \mathbf{syn}\ \overline{b}\ v \rrbracket_{I,\tau}^{\overline{g}} & \rightsquigarrow \llbracket_{\mathsf{iv}}\ v \rrbracket_{I,\tau}^{\overline{g} + \overline{a} + \overline{b}} & \text{-- interface type for later visits} \\
& \quad \llbracket \mathit{sig}\ I\ x \rrbracket : \llbracket_{\mathsf{at}}\ g_1 \rrbracket \to \ldots \to \llbracket_{\mathsf{at}}\ g_n \rrbracket \to \llbracket \mathit{resultty}\ \tau \rrbracket & \\
& \quad \llbracket \mathit{sig}\ I\ x \rrbracket\ \overline{\llbracket_{\mathsf{an}}\ g \rrbracket} = \llbracket_{\mathsf{a}}\ \mathit{inh}.a_1 \rrbracket \to \ldots \to \llbracket_{\mathsf{a}}\ \mathit{inh}.a_n \rrbracket \to & \\
& \qquad\qquad \llbracket \mathit{typrod}\ (\overline{\mathit{syn}}.b)\ (\mathit{sig}\ I\ (\mathit{name}\ v)) \rrbracket & \\
\llbracket_{\mathsf{iv}}\ \square \rrbracket_{I,\tau}^{\overline{g}} & \rightsquigarrow \llbracket \mathit{sig}\ I\ \square \rrbracket = \square & \text{-- terminator (some unit-value)} \\
\llbracket_{\mathsf{a}}\ x : e \rrbracket & \rightsquigarrow \llbracket \mathit{atname}\ x \rrbracket : \llbracket e \rrbracket & \text{-- extract attribute name and type} \\
\llbracket_{\mathsf{at}}\ x : e \rrbracket & \rightsquigarrow \llbracket e \rrbracket & \text{-- extract attribute type} \\
\llbracket_{\mathsf{an}}\ x : e \rrbracket & \rightsquigarrow \llbracket \mathit{atname}\ x \rrbracket & \text{-- extract attribute name} \\
\llbracket \mathbf{sem}\ x : I\ \overline{e}\ t \rrbracket & \rightsquigarrow \llbracket \mathit{vis}\ \mathit{lhs}\ (\mathit{name}\ t) \rrbracket\ \underline{\mathsf{where}}\ \llbracket_{\mathsf{ev}} t \rrbracket_{I}^{\overline{e},\emptyset} & \text{-- top of semfun} \\
\llbracket_{\mathsf{ev}} \mathbf{visit}\ x\ \overline{r}\ t \rrbracket_{I}^{\overline{e},\overline{g}} & \rightsquigarrow \llbracket \mathit{vis}\ \mathit{lhs}\ x \rrbracket : \llbracket \mathit{sig}\ I\ x \rrbracket\ \llbracket \overline{e} \rrbracket\ \overline{\llbracket_{\mathsf{an}}\ g \rrbracket} & \text{-- type of visit fun} \\
& \quad \llbracket \mathit{vis}\ \mathit{lhs}\ x \rrbracket\ \llbracket \mathit{inhs}\ I\ x \rrbracket = \llbracket_{\mathsf{r}}\ \overline{r} \rrbracket_{\llbracket \varsigma \rrbracket} & \text{-- chain of rules} \\
& \quad \llbracket \varsigma \rrbracket \rightsquigarrow\ = \llbracket \mathit{valprod}\ (\mathit{syns}\ I\ x)\ (\mathit{vis}\ \mathit{lhs}\ (\mathit{name}\ t)) \rrbracket & \\
& \qquad\qquad \underline{\mathsf{where}}\ \llbracket_{\mathsf{ev}} t \rrbracket_{I}^{\overline{g} + \overline{a} + \overline{b}} & \text{-- next visit} \\
\llbracket_{\mathsf{ev}} \square \rrbracket_{I}^{\overline{e},\overline{g}} & \rightsquigarrow \llbracket \mathit{vis}\ \mathit{lhs}\ \square \rrbracket : \llbracket \mathit{sig}\ I\ \square \rrbracket\ \llbracket \overline{e} \rrbracket\ \overline{\llbracket_{\mathsf{an}}\ g \rrbracket} \quad ; \quad \llbracket \mathit{vis}\ \mathit{lhs}\ \square \rrbracket = \square & \\
\llbracket_{\mathsf{r}}\ \mathbf{child}\ c : I = e \rrbracket_k & \rightsquigarrow \underline{\mathsf{with}}\ \llbracket e \rrbracket \quad \ldots \mid \llbracket \mathit{vis}\ I\ (\mathit{firstvisit}\ I) \rrbracket\ \llbracket k \rrbracket & \text{-- } k\text{: remaining rules} \\
\llbracket_{\mathsf{r}}\ \mathbf{invoke}\ x\ \mathbf{of}\ c \rrbracket_k & \rightsquigarrow \underline{\mathsf{with}}\ \llbracket \mathit{vis}\ (\mathit{itf}\ c)\ x \rrbracket\ \llbracket \mathit{inhs}\ (\mathit{itf}\ c)\ x \rrbracket & \text{-- pass inh values} \\
& \quad \ldots \mid (\mathit{valprod}\ (\mathit{syns}\ (\mathit{itf}\ c)\ x))\ \llbracket k \rrbracket & \text{-- match syn values} \\
\llbracket_{\mathsf{r}}\ p\ e' \rrbracket_k & \rightsquigarrow \llbracket_{\mathsf{ep}}\ e' \rrbracket_{p}^{k} & \text{-- translation for attr def rule} \\
\llbracket_{\mathsf{ep}}\ \underline{\mathsf{with}}\ e\ \overline{p\ e'} \rrbracket_{p}^{k} \rightsquigarrow \underline{\mathsf{with}}\ e & \overline{\ldots \mid \llbracket p \rrbracket\ \llbracket_{\mathsf{r}}\ p\ e' \rrbracket_k} & \text{-- rule RHS is with-constr} \\
\llbracket_{\mathsf{ep}}\ = e \rrbracket_{p}^{k} & \rightsquigarrow \underline{\mathsf{with}}\ e \quad \ldots \mid \llbracket p \rrbracket\ k & \text{-- rule RHS is expr} \\
\mathit{atref}\ \mathit{inh}.c.x = c_i x & \mathit{atname}\ \mathit{inh}.x = \mathit{inh}_a x & \text{-- naming conventions} \\
\mathit{atref}\ \mathit{syn}.c.x = c_s x & \mathit{atname}\ \mathit{syn}.x = \mathit{syn}_a x & \text{-- } \mathit{atref}\text{: ref to attr value} \\
\mathit{atref}\ \mathit{loc}.x \ \ = \mathit{loc}_l x & \mathit{atname}\ x \qquad = x & \text{-- } \mathit{atname}\text{: ref to attr in type} \\
\mathit{vis}\ I\ x = \mathit{vis}\ \mathit{lhs}\ x & \mathit{sig}\ I \quad = T'I & \text{-- } \mathit{vis}\text{: name of visit function} \\
\mathit{vis}\ c\ x = c_v x & \mathit{sig}\ I\ x = T'I'x & \text{-- } \mathit{sig}\text{: itf types}
\end{array}
$$

Fig. 2: Translation of $\mathrm{AG}_{\mathrm{DA}}^{\mathrm{X}}$ to Agda.

We generate types for coroutines and for the individual visit functions that make up such a coroutine. These types are derived from the interface. For each visit (e.g. *translate* of *Source*), we generate a type that represents a function type from the attribute types of the inherited attributes for that visit, to a dependent product (Σ) of the types of the synthesized attributes and the type of the next visit function. These types are parameterized with the attributes of earlier visits (e.g. $T'Source'translate\ syn_a gathEnv$). The type of the coroutine itself is the type of the first visit.

$$
\begin{array}{l}
T'Source \quad = \; T'Source'analyze \\
T'Source'analyze \quad = \; \Sigma \quad Env \quad T'Source'translate \\
T'Source'translate \quad syn_a gathEnv \; = \\
\quad (inh_a finEnv \; : \; Env) \quad \rightarrow \quad (inh_a gathInFin \; : \; syn_a gathEnv \sqsubseteq inh_a finEnv) \; \rightarrow \\
\qquad \Sigma \; (Errs\ inh_a finEnv \uplus Target\ inh_a finEnv) \\
\qquad\quad (T'Source'\Box \quad syn_a gathEnv\ inh_a finEnv\ inh_a gathInFin) \\
T'Source'\Box \quad syn_a gathEnv\ inh_a finEnv\ inh_a gathInFin\ syn_a outcome \; = \Box
\end{array}
$$

The restrictions on attribute order in the interface ensure that referenced attributes are in scope. The scheme for $[\![_{\mathsf{iv}}\ v]\!]^{I}_{g,\tau}$ formalizes this translation, where g is the list of preceding attribute declarations, and τ the type for I. The *typrod* function mentioned in the scheme constructs a right-nested dependent product.

The coroutine itself consists of nested continuation functions (one for each visit). Each continuation takes the visit's inherited attributes as parameter, and consists of a tree of with-constructs that represent intermediate computations for computations of attributes and invocations of visits to children. Each leaf ends in a dependent product of the visit's synthesized attributes and the continuation function for the next visit[6].

$sem\diamond \; : T'Source \rightarrow T'Source \rightarrow T'Source$	-- example translation for $\diamond$
$sem\diamond \;\; loc_c left\ loc_c right = lhs_v analyze$ <u>where</u>	-- delegates to first visit function
$\quad lhs_v analyze : T'Source'analyze$	-- signature of first visit function
$\quad lhs_v analyze$ <u>with</u> ...	-- computations for *analyze* here
$\quad ... = (lhs_s gathEnv, lhs_v translate)\ ahwere$	-- result of first visit function
$\qquad lhs_v translate : T'Source'translate\ lhs_s gathEnv$	-- last visit function
$\qquad lhs_v translate\ lhs_i finEnv\ lhs_i gathInFin$ <u>with</u> ...	-- computations for *translate* here
$\qquad ... = (lhs_s outcome, lhs_v \Box)$ <u>where</u>	-- result of second visit function
$\qquad\quad lhs_v \Box : T'Source'\Box\ lhs_s gathEnv\ lhs_i finEnv\ lhs_i gathInFin\ lhs_s outcome$	
$\qquad\quad lhs_v \Box = \Box$	-- explicit terminator value

The scheme $[\![_{\mathsf{ev}} v]\!]^{\overline{e},\overline{g}}_{I}$ formalizes this translation for a visit v of interface I, where $\overline{e}$ are type arguments to the interface (empty in the example), and $\overline{g}$ are the attributes of previous visits.

The with-tree for a visit-function consists of the translation of child-rules, invoke-rules and evaluation rules. Each rule plugs into this tree. For example, the translation for $[\![$**child** *left* : *Source* = $loc_s left]\!]$ is:

... <u>with</u> $loc_s left$	-- evaluate RHS to get first visit fun
... | $left_v analyze$ <u>with</u> ...	-- give it a name + proceed with remainder

For $[\![$**invoke** *translate* **of** *left*$]\!]$ the translation is:

... <u>with</u> $left_v translate\ left_i finEnv\ left_i gathInFin$	-- visit fun takes inh attrs
... | $(left_s outcome, left_v sentinel)$ <u>with</u> ...	-- returns product of syn attrs

[6] As a technical detail, a leaf of the with-tree may also be an *absurd pattern*. These are used in Agda to indicate an alternative that is never satisfyable. A body for such an alternative cannot be given.

For $[\![lhs.gathEnv = left.gathEnv + right.gathEnv]\!]$:

$$
\begin{array}{ll}
\ldots\ \underline{\mathsf{with}}\ left_s gathEnv + right_s gathEnv & \text{-- translation for RHS} \\
\ldots\ |\ lhs_s gathEnv\ \underline{\mathsf{with}}\ \ldots & \text{-- LHS + remainder}
\end{array}
$$

For $[\![lhs.outcome\ \underline{\mathsf{with}} \ldots]\!]$ (where the RHS is a with-construct), we duplicate the remaining with-tree for each alternative of the RHS:

$$
\begin{array}{ll}
\ldots\ \underline{\mathsf{with}}\ left_s outcome & \text{-- translation for RHS} \\
\ldots\ |\ inj_1\ es\ \underline{\mathsf{with}}\ right_s outcome & \\
\ldots\ |\ inj_1\ es_1\ |\ inj_1\ es_2\ \underline{\mathsf{with}}\ inj_1\ (es_1 + es_2) & \text{-- alternative one of four} \\
\ldots\ |\ inj_1\ es_1\ |\ inj_1\ es\ \ \ |\ lhs_s outcome\ \underline{\mathsf{with}}\ \ldots & \text{-- LHS + remainder} \\
\ldots\ |\ inj_1\ es_1\ |\ inj_2\ \_\ \ \ \underline{\mathsf{with}}\ inj_1\ es_1 & \text{-- alternative two of four} \\
\ldots\ |\ inj_1\ es_1\ |\ inj_2\ \_\ \ \ |\ lhs_s outcome\ \underline{\mathsf{with}}\ \ldots & \text{-- LHS + remainder} \\
\ldots\ |\ inj_2\ \ldots & \text{-- remaining two alternatives}
\end{array}
$$

The scheme $[\![_{\mathsf{r}}\ r]\!]_k$ formalizes this translation, where r is a rule and k the translation of the rules that follow r.

The size of the translated code may be exponential in the number of rules with with-constructs as RHS. It is not obvious how to treat such rules otherwise. Agda does not allow a with-construct as a subexpression. Neither can we easily factor out the RHS of a rule to a separate function, because the conclusions drawn from the evaluation of preceding rules are not in scope of this function. Fortunately, for rules that would otherwise cause a lot of needless duplication, the programmer can perform this process manually.

When dependent pattern matching brings assumptions in scope that are needed *across* rules, the code duplication is a necessity. To facilitate that pattern matching effects are visible across rules, we need to ensure that the rule that performs the match is ordered before a rule that needs the assumption. We showed in previous work how such non-attribute dependencies can be captured [4].

The translated code has attractive operational properties. Each attribute is only computed once, and each node is at most traversed k times.

6 Partially Defined Attributes

A fine granularity of attributes is important to use an AG effectively. In the mini-compiler of Section 3, we could replace the attribute *outcome* with an attribute *code* and a separate attribute *errors*. This would be more convenient, since it would not require a pattern match against the *output* attribute to collect errors. However, we cannot produce a target term in the presence of errors, thus *code* would not have a total definition. Therefore, we were forced to combine these two aspects into a single attribute *outcome*. It is common to use partially defined attributes in an AG. This holds especially when the attribute's value (e.g. *errors*) determines if another attribute is defined (e.g. *code*). We present a solution that uses the partitioning of attributes over visits.

The idea is to make the availability of visits dependent on the value of a preceding attribute. We split up the *translate* visit in a visit *report* and a visit *generate*. The visit *report* has *errors* as synthesized attribute, and *generate* has *code*. Furthermore, we enforce that *generate* may only be invoked (by the parent in the AST) when the list of errors reported in the previous visit is empty. We accomplish this with an additional attribute *noErrors* on *generate* that gives evidence that the list of errors is empty. With this evidence, we can give a total definition for *code*.

```
itf Source     -- Root's visit needs to be split up in a similar way
  visit report     syn errors   : Errs inh.finEnv     -- parent can inspect errors
  visit generate   inh noErrors : syn.errors ≡ [ ]     -- enforces invariant
                   syn code     : Target inh.finEnv   -- only when errors is empty
datasem Source prod use    -- example for production use
  loc.testInEnv = loc.ι ∈? lhs.finEnv         -- scheduled in visit report
  lhs.code with loc.testIn | lhs.noErrors     -- scheduled in visit generate
    | inj₁ _    | ()                          -- cannot happen, hence an absurd pattern
    | inj₂ isIn | refl = use loc.ι isIn       -- extract the evidence needed for the code term
datasem Source prod ⋄    -- leftNil : (α : Env) → (β : Env) → (α ++ β ≡ [ ]) → (α ≡ [ ])
  left.noErrors = leftNil left.errors right.errors lhs.noErrors    -- right.noErrors similar
  lhs.code      = left.code ⋄ right.code    -- scheduled in visit generate
```

For this approach work, it is essential that visits are scheduled as late as possible, and only those that are needed.

We can generalize the presented approach by defining a fixed number of alternative sets of attributes for a visit, and use the value of a preceding attribute to select one of these sets [6].

7 Related Work

Dependent types originate in Martin-Löf's Type Theory. A variety of dependently typed programming languages are gaining popularity, including Agda [7], Epigram, and Coq. We present the ideas in this paper with Agda as host language, because it has a concept of a dependent pattern match, to which we straightforwardly map the left-hand sides of AG rules. 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 [3] are considered to be a promising implementation for compiler construction. Recently, many Attribute Grammar systems arose for mainstream languages, such as the systems JastAdd and Silver for Java, and UUAG [10] for Haskell. These approaches may benefit from the stronger type discipline as presented in this paper; however, it would require an encoding of dependent types in the host language.

AGs have a straightforward translation to cyclic functions in a lazy functional programming language [9]. To prove that cyclic functions are total and terminating is a non-trivial exercise. Kastens [2] presented Ordered Attribute Grammars (OAGs). In OAGs, the evaluation order of attribute computations as well as attribute lifetime can be determined statically. Saraiva [8] described how to generate (noncyclic) functional coroutines from OAGs. The coroutines we generate are based on these ideas.

8 Conclusion

We presented AG_{DA}, a language for ordered AGs 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 in the type of attributes, and pass proofs of these invariants around as attributes. With a dependently typed AG, we write algebras for catamorphisms in a dependently typed language in a composable way. Each attribute describes a separate aspect of the catamorphism.

The approach we presented is lightweight, which means that we encode AGs as an embedded language (via a preprocessor), such that type checking is deferred to the host language. To facilitate termination checking, we translate the AG to a coroutine (Section 5) that encodes a terminating, multi-visit traversal, under the restriction that the AG is ordered and definitions for attributes are total.

The preprocessor approach fits nicely with the interactive Emacs mode of Agda. Type errors in the generated program are traceable back to the source: in a statically checked AG_{DA} program these can only occur in Agda blocks. These Agda blocks are literally preserved; due to unicode, even attribute references can stay the same. Also, the Emacs mode implements interactive features via markers, which are also preserved by the translation. The AG preprocessor is merely an additional preprocessing step.

With some generalizations, the work we have presented is a proposal for a more flexible termination checker for Agda that accepts k-orderable cyclic functions, if the function can be written as a non-cyclic k-visit coroutine.

Acknowledgments. This work was supported by Microsoft Research through its European PhD Scholarship Programme. We thank the anonymous reviewers of IFL'10 for their extensive comments and suggestions during the post-refereeing process.

References

1. Bove, A., Dybjer, P.: Dependent Types at Work. In: Language Engineering and Rigorous Software Development. LNCS, vol. 5520, pp. 57–99 (2009)
2. Kastens, U.: Ordered Attributed Grammars. Acta Informatica 13, 229–256 (1980)
3. Knuth, D.E.: Semantics of Context-Free Languages. Mathematical Systems Theory 2(2), 127–145 (1968)
4. Middelkoop, A., Dijkstra, A., Swierstra, S.D.: Attribute Grammars with Side Effect. In: HOSC (2010), http://people.cs.uu.nl/ariem/wgt10-journal.pdf
5. Middelkoop, A., Dijkstra, A., Swierstra, S.D.: Iterative Type Inference with Attribute Grammars. In: GPCE '10. pp. 43–52 (2010)
6. Middelkoop, A., Dijkstra, A., Swierstra, S.D.: Visit Functions for the Semantics of Programming Languages. In: WGT '10 (2010), http://people.cs.uu.nl/ariem/wgt10-visit.pdf
7. Norell, U.: Dependently-typed Programming in Agda. In: TLDI '09. pp. 1–2 (2009)
8. Saraiva, J., Swierstra, S.D.: Purely Functional Implementation of Attribute Grammars. Tech. rep., Universiteit Utrecht (1999)
9. Swierstra, S.D., Alcocer, P.R.A.: Attribute Grammars in the Functional Style. In: Systems Implementation 2000. pp. 180–193 (1998)
10. Universiteit Utrecht: Homepage of the Universiteit Utrecht Attribute Grammar System. http://www.cs.uu.nl/wiki/HUT/AttributeGrammarSystem (1998)