From Object Algebras to Attribute Grammars

Tillmann Rendel Jonathan Immanuel Brachthäuser Klaus Ostermann
University of Marburg, Germany

Abstract

Oliveira and Cook (2012) and Oliveira et al. (2013) have recently introduced object algebras as a program structuring technique to improve the modularity and extensibility of programs. We analyze the relationship between object algebras and attribute grammars (AGs), a formalism to augment context-free grammars with attributes. We present an extension of the object algebra technique with which the full class of L-attributed grammars – an important class of AGs that corresponds to one-pass compilers – can be encoded in Scala. The encoding is modular (attributes can be defined and type-checked separately), scalable (the size of the encoding is linear in the size of the AG specification) and compositional (each AG artifact is represented as a semantic object of the host language). To evaluate these claims, we have formalized the encoding and re-implemented a one-pass compiler for a subset of C with our technique. We also discuss how advanced features of modern AG systems, such as higher-order and parameterized attributes, reference attributes, and forwarding can be supported.

Categories and Subject Descriptors D.1.5 [*Programming Techniques*]: Object-oriented Programming; D.3.1 [*Programming Languages*]: Formal Definitions and Theory

Keywords Object Algebras; Visitor Pattern; Attribute Grammars; Church-encoding; Embedded Domain-Specific Languages; Modularity; One-Pass Compilers; Scala

1. Introduction

The last years have seen a revival of program structuring techniques based on Church encodings for such topics as data-type generic programming (Hinze 2004, 2006), ad-hoc polymorphic functions (Oliveira and Gibbons 2005), the "finally tagless" embedding of typed languages (Carette et al. 2007, 2009), polymorphic embedding of domain-specific languages (Hofer et al. 2008), and object algebras (Oliveira and Cook 2012; Oliveira et al. 2013). In these approaches, data is not represented as physical data but rather as "recipes" of how to perform computations on the data.

Folds vs Church	Hinze (2006)
Folds vs AGs	Johnsson (1987); Chirica and Martin (1979)
Folds vs Visitors	Gibbons (2006)
Visitors vs Church	Buchlovsky and Thielecke (2006); Oliveira et al. (2008); Oliveira et al. (2013)
Visitors vs AGs	Middelkoop et al. (2011)

Table 1. Excerpt of previous work on relations between approaches from Table 2

Church encodings are similar to folds/catamorphisms on algebraic datatypes in functional programming, to internal visitors in object-oriented programming, and to synthesized attributes in AGs (Table 2). The relationships between almost all of these approaches have been analyzed in detail in previous works, see Table 1; only one pair is missing: Church encodings vs AGs. This paper is supposed to fill that gap. Since object algebras are the variant of Church encodings that is most similar to AGs, we will concentrate on the relation between object algebras and AGs in the remainder of this paper.

We consider the exploration of this relationship a worthwhile endeavour. On one hand, our attempts to encode AGs have significantly extended the expressive power of the object algebra technique. Concepts that are very natural in the AG world, such as grammars with multiple non-terminals, inherited attributes, or various classes of allowed dependencies between attributes have not or only insufficiently been explored in the OA world. Advanced AG features, such as RAGs (Hedin 2000), higher-order attributes (Vogt et al. 1989), or forwarding (Van Wyk et al. 2002) are all potentially useful and interesting to consider from an object algebra perspective as well.

OOPSLA '14, October 19–21, 2014, Portland, OR, USA.

ACM 978-1-4503-2585-1/14/10. . . $15.00.
http://dx.doi.org/10.1145/2660193.2660237

Folds	Visitors	Church Encodings	Attribute Grammars
algebraic datatype	class hierarchy	algebra signature	CFG
catamorphism/fold	visit/accept methods	the value itself	traversal
fold args / algebras	concrete visitor	algebra	attribute
instance of datatype	instance of class	algebra-polymorphic traversal	sentence in language

Table 2. Similarities between program structuring approaches

On the other hand, AGs can benefit from an embedding into a statically typed general purpose language by new possibilities, such as modular type checking, compositional program understanding, and the ability to abstract over entities that are usually not first-class in the AG world. AG systems, such as JastAdd (Hedin 2011), Silver (Van Wyk et al. 2007), and UUAGC (Swierstra et al. 1998) are often implemented as non-compositional source-to-source translations, which hinders compositional program understanding (for example, when static or dynamic errors occur in the generated code). Also, having a proper semantic representation (as semantic objects of the base language) of AG artifacts improves the distinction between syntax and semantics; semantic objects can be composed regardless of the syntax that was used to synthesize them.

Most importantly, object algebras are a quite powerful modularization technique. They provide an elegant solution to the expression problem (Wadler 1998), and they can be used to decompose traversal algorithms in a modular and composable way. We discuss the various dimensions of modularity in Section 2. Broadening object algebras to encode AGs means that object algebras are potentially useful as a novel and highly modular way of building extensible compilers. The core idea of Church-encoding AGs is also what distinguishes this work from many other previous works on compiling or embedding AGs. We use the programming language Scala in this paper, but there is only one feature specific to Scala that is essential for our approach, namely intersection types (A **with** B). Variance annotations are also important for our approach, but similar features (for example, wildcards) also exist for other languages.

In our encoding, we strive for three important and non-trivial properties: modularity, scalability, and compositionality. By modularity we mean that attributes can be defined and type-checked separately, while still guaranteeing statically that the invariants (with respect to dependencies between attributes and termination) of the AG class we encode are preserved. By scalability we refer to the property that the size of the encoding is linear in the size of the grammar and the number and size of the attributes. By compositionality we mean that each AG artifact is represented as a semantic object of the host language.

This paper makes the following contributions:

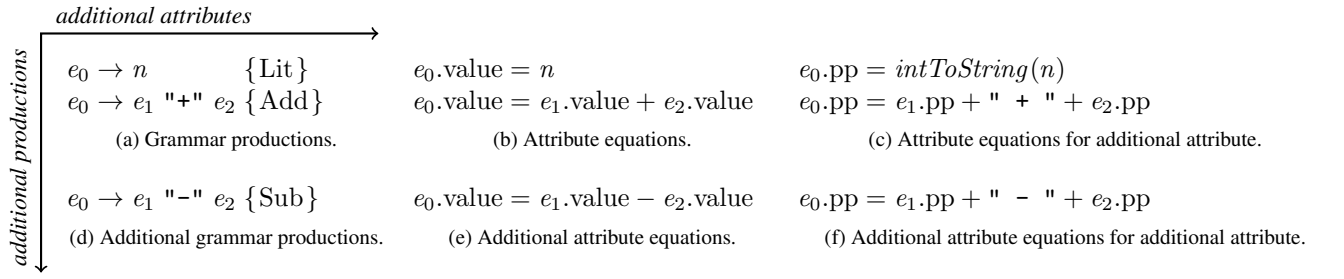
- Section 3 observes that object algebras can be seen as Church-encodings of S-attributed grammars (Lewis et al. 1974).
- Section 4 improves the modularity of this encoding of S-attributed grammars by introducing context-sensitive object algebras.
- Section 5 extends the encoding to L-attributed grammars by encoding context decorators as object algebras, inheriting their modularity properties.
- Section 6 generalizes the example from the previous sections and discusses how any L-attributed grammar can be encoded in terms of object algebras. To this end, we present a program generator that takes the representation of a datatype (grammar) as input and produces the support code required for our encoding. The attributes and their equations are then encoded as ordinary Scala code that uses the support code. We also discuss why our encoding fulfills the three properties mentioned above, modularity, scalability, and compositionality.
- Section 7 evaluates our technique experimentally. We have taken an existing one-pass compiler for a subset of C and reformulated and modularized it with the technique shown in this paper. The case study suggests that our approach scales to programs of realistic size.
- Section 8 discusses how the advanced AG features mentioned above (RAGs, higher-order attributes, forwarding) could be supported by our encoding.

We defer the discussion of related work to Section 9. The implementation of our program generator, the case study, and examples that illustrate the techniques to encode advanced AG features are available online[1].

2. Attribute Grammars and the Expression Problem

Wadler (1998) asks for a representation of tree-shaped data that supports two dimensions of extensibility: Extension with new data variants and extension with new tree traversals. A solution to this *expression problem* should allow extensions to reuse old data variants or traversals without

[1] `http://www.informatik.uni-marburg.de/~rendel/oa2ag/`



(a) Grammar productions. (b) Attribute equations. (c) Attribute equations for additional attribute.

(d) Additional grammar productions. (e) Additional attribute equations. (f) Additional attribute equations for additional attribute.

Figure 1. Two dimensions of extensibility for an attributed grammar.

changing or recompiling their implementation on the one hand, and it should statically ensure that all traversals support all data variants on the other hand. Odersky and Zenger (2005) also require independent extensibility, that is, the combination of independently developed extensions. In this work, we focus on a third dimension of modularity that has been somewhat less studied: Composition of traversal operations from components.

2.1 Two Dimensions of Extensibility

An important example of tree-shaped data is the abstract syntax of a programming language. In this context, the two dimensions of extensibility correspond to an extension of the language with additional syntactic forms and to an extension of an implementation for the language with additional traversals of the abstact syntax tree. For example, Figures 1a and 1b specify a simple language of arithmetic expressions and its evaluation to numbers as an attribute grammar.

Attribute grammars are context-free grammars augmented with attribute definitions. In this example, the non-terminal e is augmented with an attribute value. In AG terminology, value is a *synthesized* attribute: It is an attribute of non-terminals that is computed (or "synthesized") from attributes of its children, as can be seen by the attribute equations in Figure 1b. Here, in the second equation value is expressed in terms of the values for e_1 and e_2, child nodes of e_0.

Our example contains two extensions of this basic system, exemplifying the two dimensions of extensibility: Figure 1c adds the additional operation of pretty printing, and Figures 1d and 1e extend the grammar with an additional syntactic form for subtraction. To use both extensions together as required for independent extensibility, we specify how the pretty printing behaves for subtraction nodes in the abstract syntax tree, as shown in Figure 1f.

2.2 A Third Dimension of Modularity

In this paper, we focus on a third dimension of modularity that is relevant to the traversal of tree-shaped data: The composition of traversal operations from components. Such a decomposition of traversal operations is relevant in practice because operations can be large and complex. With attribute grammars, equations for one attribute can mention other attributes, and an attribute grammar will usually define dozens of such interdependent attributes.

$e_0.\text{pp} =$
if $(e_2.\text{value} \equiv 0)$
  $e_1.\text{pp}$
else
  $e_1.\text{pp} + \text{" + "}\ e_2.\text{pp}$

(a) Accessing the value attribute of a child of the current node.

$e_0.\text{pp} =$
if $(e_0.\text{value} \equiv 0)$
  "0"
else
  $e_1.\text{pp} + \text{" + "}\ e_2.\text{pp}$

(b) Accessing the value attribute of the current node itself.

Figure 2. Two ways of using the value attribute in an equation for the pp attribute.

For example, we present two variants of the simple pretty printer from Figure 1c which use the value attribute in an equation for the pp attribute. The variant in Figure 2a just emits the first operand of an addition if the value of the second one is zero, and the variant in Figure 2b prints "0" for expressions with overall value zero. These variants need to access already computed attributes on a child of the current node or on the current node itself, respectively.

One task of AG implementations is to order the computation of the various attributes in an attribute grammar so that all attribute values are available when they are required. If possible, this should happen without performing unnecessary traversals of the abstract syntax tree. For example, the attribute grammars in Figures 1 and 2 can all be computed in one traversal of the abstract syntax tree, interleaving the computation of the value and pp attributes.

Decomposing a traversal into multiple components should not result in additional traversals at runtime. This suggests that we should distinguish full traversals from reusable components of traversals. In the encoding presented in this paper, the difference between traversals and reusable components of traversals is expressed via the type parameters of a generic interface. These type parameters are also used to encode the dependencies between the reusable components. This allows us to statically check in the Scala type system that all operations expressible with our encoding can be run as a single terminating traversal provided the equations themselves are written in a pure and strongly normalizing subset of Scala. [2]

[2] There are two potential sources of nontermination in attribute grammars: Cyclic attribute dependencies and nonterminating equations, given a suffi-

ExprSig[E]

```
trait ExprSig[E] {
  def Lit: Int ⇒ E
  def Add:(E, E) ⇒ E
}
```

(a) Algebraic signature

```
trait ExprCompose[E1, E2] extends ExprSig[E1 with E2] {
  val alg1: ExprSig[E1]
  val alg2: ExprSig[E2]
}
```

(b) Composition of algebras

Figure 3. Encoding an S1-attributed grammar.

2.3 The Benefits of Encoding

As a source language, it appears as if attribute grammars already solve the expression problem, including the decomposition of traversals into multiple attributes. However, attribute grammar implementations often lack modular reasoning and separate compilation, which are key ingredients to a solution to the expression problem. The encoding we present in this paper combines the source-language modularity properties of attribute grammars with the target-language modularity properties of object algebras in Scala.

Beyond the potential practical use of the encoding as the foundation for an attribute grammar compiler, the encoding serves as a bridge between the two fields of study. The area of attribute grammars is much more developed, so we believe that at first, this work will help to understand object algebras better in the light of what we already know about attribute grammars.

3. Encoding Synthesized Attributes

In this section, as well as in the following two sections 4 and 5, we illustrate our encoding informally by means of a simple example. We will introduce the components of the encoding step-by-step, starting with synthesized single-attribute grammars, and ending with L-attributed grammars. The example grammar considered in this section consists of a single non-terminal only. In Section 6, we will formalize the informal findings from this section and also generalize to the case of multiple non-terminals.

3.1 S1-Attributed Grammars

Oliveira and Cook (2012) present a solution to the expression problem based on the technique of objects algebras. Object algebras address the problem by interpreting variants of an algebraic data type as constructors of an algebraic signature and operations on the data type as implementations of such signatures. It is important to note that a function handling different variants of a type (possibly by pattern matching) is hereby turned into a record containing one function per variant. This transformation then allows usage of object oriented methodology to structure reusable and extensible components.

```
val Expr_value = new ExprSig[Int] {
  def Lit = n ⇒ n
  def Add =(e1, e2) ⇒ e1 + e2
}
```

Figure 4. Equations encoded as Algebra

Following the translation scheme above, we can define the attribute grammar (modulo the concrete syntax) as an object algebra. The structure of the language is defined in terms of the object algebra signature in Figure 3a[3]. We are using Scala's support for function types to closely model the corresponding algebraic signature[4]. Thus Lit is defined to be a unary operation from Int to E. E represents the carrier set and is encoded as type parameter of the trait. We also use upper-case method names for the members of the algebraic signature because these members are used like constructors. This has the additional benefit of avoiding ambiguities with keywords of the host language.

The equations of the attribute grammar translate to implementations of the signature methods. Figure 4 shows the object algebra implementing the value attribute for the language Expr. To this end the above algebraic signature is instantiated, defining the carrier type to be Int. Programs of Expr such as `3 + 5` are represented as functions that are parametric in the object algebra.

```
def threeplusfive[E](alg: ExprSig[E]): E =
  alg.Add(alg.Lit(3), alg.Lit(5))
```

Calling threeplusfive(Expr_value) corresponds to computing the value attribute on the input program using the original attribute grammar defined in Figures 1a and 1b.

cently strong equation language. Our encoding statically avoids the former but is liable to the latter, because it uses the general-purpose language Scala as an equation language. Statically checking whether the Scala equations terminate and are free of side effects is out of scope for this paper.

[3] The figure shows the first step of the development of our encoding. Subsequent figures will add to the definitions and refine existing ones. To illustrate the differences, additions and changes are highlighted with a gray background.

[4] We choose an encoding like **def** Lit: Int ⇒ E over an encoding like **def** Lit (n: Int): E because definition site type inference is better for functions and thus allows more concise implementations.

To also support the same concrete syntax as the attribute grammar, one would have to add a parser that calls the functions of the object algebra signature, but we are not interested in concrete syntax at this point.

The above encoding is straightforward if the AG consists only of a single synthesized attribute. Purely synthesized AGs are equivalent to first-order folds (Duris et al. 1996). As we have seen in the above example, any AG with only a single synthesized attribute (S1-AGs) can immediately be expressed as an object algebra.

3.2 S-Attributed Grammars

One way to encode multiple attributes is to instantiate the type parameter E from above with a tuple or record type, whereby each attribute corresponds to one tuple/record component. However, AG systems usually allow to define each attribute separately. To achieve a similar kind of modularity, we can adapt the technique for feature-oriented decomposition of object algebras by Oliveira et al. (2013), which uses intersection types. In Scala this is realized by trait composition which resembles a form of multiple inheritance with the order of super classes linearized by the compiler at compile time. For example in the declaration $\mathbf{trait}\ C\ \mathbf{extends}\ A\ \mathbf{with}\ B$ the type C is a subtype of both A and B.

In Figure 5a, we adapt the evaluation attribute to this style and also consider the extension of adding a second attribute pp for pretty-printing. This implementation above differs from Oliveira et al.'s proposal because the $\mathrm{HasValue}$ and HasPP helper functions are strict, so the attributes are computed during the traversal, not when pp or value are called. Both attributes are now defined separately, but how can they be combined such that they can be computed in a single, simple traversal? Oliveira et al. propose to use the generic composition trait in Figure 3b, which composes two algebras "pointwise". The instantiation for the two interfaces $\mathrm{HasValue}$ and HasPP can be found in Figure 5b. Using $\mathrm{ComposeValuePP}$ and defining alg_1 to be $\mathrm{Expr\_value}$ and alg_2 to be $\mathrm{Expr\_pp}$, we can compute the values of both attributes in a single traversal.

In this way, S-attributed grammars (Lewis et al. 1974) with independent attributes and a single non-terminal can be encoded.

3.3 Extensibility

Since our encoding is based on object algebras, it also inherits the extensibility properties (Oliveira and Cook 2012). The remainder of this subsection briefly recapitulates how extensibility manifests itself in our setting. For this purpose, let us consider the addition of the grammar production Sub as introduced in Figure 1d. The new algebraic signature extends the existing one, which can be modeled in Scala with inheritance:

```
trait HasValue {def value: Int}
def HasValue(v: Int) = new HasValue {val value = v}
trait HasPP {def pp: String}
def HasPP(p: String) = new HasPP {val pp = p}
val Expr_value = new ExprSig[HasValue] {
  def Lit = n ⇒ HasValue(n)
  def Add =(e1, e2) ⇒ HasValue(e1.value + e2.value)
}
val Expr_pp = new ExprSig[HasPP] {
  def Lit = n ⇒ HasPP(n.toString)
  def Add =(e1, e2) ⇒ HasPP(e1.pp + "+" + e2.pp)
}
```

(a) Two attribute equation-sets

```
trait ComposeValuePP
    extends ExprCompose[HasValue, HasPP] {
  val alg1: ExprSig[HasValue]
  val alg2: ExprSig[HasPP]
  def Lit = n ⇒ new HasValue with HasPP {
    val value = alg1.Lit(n).value
    val pp = alg2.Lit(n).pp
  }
  def Add =(e1, e2) ⇒ new HasValue with HasPP {
    val value = alg1.Add(e1, e2).value
    val pp = alg2.Add(e1, e2).pp
  }
}
```

(b) Composition

Figure 5. Composing two algebras to compute two synthesized attributes at once.

```
trait ExprSubSig[E] extends ExprSig[E] {
  def Sub:(E, E) ⇒ E
}
```

The algebra for *ExprSub_value* can reuse the algebra for $\mathrm{Expr\_value}$ from Figure 5a as follows:[5]

```
trait ExprSub_value
  extends Expr_value with ExprSubSig[HasValue] {
  def Sub =(e1, e2) ⇒ HasValue(e1.value − e2.value)
}
val ExprSub_value = new ExprSub_value { }
```

Independent extensibility is provided by multiple inheritance. The same kind of extensibility holds for all further developments of the encoding as they are presented in the remainder of this paper.

[5] We use $\mathrm{Expr\_value}$ as a trait here, even though it was defined as a value in Figure 5a. In practice, we would follow the standard Scala pattern of defining both a trait and an equally named value instantiation the trait, as we do for $\mathrm{Expr2\_value}$ here. We avoided this extra complexity for presentation purposes in Figure 5a.

PreExprSig[E, Out] *Extends Figure 3*

```
type ExprSig[E] = PreExprSig[E, E]
trait PreExprSig[−E, +Out] {
  def Lit: Int ⇒ Out
  def Add:(E, E) ⇒ Out
}
```

(a) Algebraic signature

```
trait ExprCompose[ I1 , O1, I2 , O2]
    extends PreExprSig[ I1 with I2 , O1 with O2] {
  val alg1: PreExprSig[ I1, O1]
  val alg2: PreExprSig[ I2, O2]

  def Lit = n ⇒ {
    val out1 = alg1.Lit(n)
    val out2 = alg2.Lit(n)
    mix[O1, O2](out1, out2)
  }
  ...

  ...
  def Add =(e1, e2) ⇒ {
    val out1 = alg1.Add(e1, e2)
    val out2 = alg2.Add(e1, e2)
    mix[O1, O2](out1, out2)
  }
}
```

(b) Composition of pre-algebras

Figure 6. Modularizing attribute definitions.

4. Modularity of the Encoding

With the above coding scheme it is possible to define multiple isolated synthesized attributes in parallel. In order to also support modular definition of multiple attributes with dependencies we have to modify the encoding some more.

There are two ways of modularizing the definition of synthesized attributes. An attribute equation for a given nonterminal production can access other attributes that either

1. have already been computed for the children of the nonterminal production or
2. have already been computed for the same nonterminal.

In this section we will develop step-by-step an encoding that supports both modularity aspects.

4.1 Accessing other attributes of children

In the examples of object algebras we have seen so far, both the type of child nodes and the result of the computation have been the same. This is a crucial requirement to be able to use the algebra for folding or in an embedded style as for threeplusfive.

To facilitate reuse of existing components in a type safe way we need a means to specify the required interface for attributes computed on child nodes. By separating the contravariant (input to the current computation) and the covariant (output of the current computation) occurrences of the sort we create a more fine grained signature interface, thus turning the algebra into a *pre-algebra*.

An example of a pre-algebra for the expression language is *PreExprAlg*, defined in Figure 6a. The contravariant type $-E$ specifies requirements on the type of the children[6], independent of the covariant output $+Out$ which is being computed. In Scala $-$ and $+$ are used to denote contravariant and covariant type arguments respectively.

For instance, let us implement the pretty-printer as introduced in Figure 2a which only emits the first operand of an addition if the value of the second is zero.

```
trait Expr_ppOpt extends
  PreExprSig[HasPP with HasValue, HasPP] {
  def Lit = n ⇒ HasPP(n.toString)
  def Add =(e1, e2) ⇒
    if(e2.value ≡ 0) e1 else
      HasPP(e1.pp + "+" + e2.pp)
}
```

Obviously, a pre-algebra is in general not complete: It cannot be used for actual computations unless input and output agree. Hence we reconstruct ExprSig as a special case of PreExprSig as defined in Figure 6a.

Pre-algebras are useful for modularization, though. A generalization of the above mentioned ExprCompose trait can be used to compose pre-algebraic fragments and thereby satisfying (potentially mutually recursive) dependencies.

To define the composition trait (and other future definitions), we need a right-biased object composition function which takes two objects and merges them, whereby the right object "wins" if both objects define the same method:

```
def mix[A, B](a: A, b: B): A with B
```

This function cannot be defined directly in Scala, but it can be simulated using a combination of macros and implicit parameters. For clarity, we use mix instead of the more elaborate simulation in the remainder of this section.

Using mix, we can define a general algebra composition trait for pre-algebras as can be seen in Figure 6b. There is

[6] All child nodes in this example are expressions, hence the name E. Choosing distinct names for the contravariant type parameters is important in a setting with multiple non-terminals as can be seen in Section 6

CtxExprSig[E, Ctx, Out] *Extends Figure 6*

```
type CtxExprSig[−E, −Ctx, +Out] = PreExprSig[E, Ctx ⇒ Out]
type ExprSig[E] ...
trait PreExprSig[−E, +Out] ...
```

(a) Algebraic signature

```
trait ExprCompose[
    E1, C1, O1, E2, C2 >: C1 with O1, O2]
  extends CtxExprSig[E1 with E2, C1, O1 with O2] {
  val alg1: CtxExprSig[E1, C1, O1]
  val alg2: CtxExprSig[E2, C2, O2]
  def Lit = n ⇒ ctx ⇒ {
    val out1 = alg1.Lit(n)(ctx)
    val out2 = alg2.Lit(n)(mix[C1, O1](ctx, out1))
    mix[O1, O2](out1, out2)
  }
  def Add =(e1, e2) ⇒ ctx ⇒ {
    val out1 = alg1.Add(e1, e2)(ctx)
    val out2 = alg2.Add(e1, e2)(mix[C1, O1](ctx, out1))
    mix[O1, O2](out1, out2)
  }
}
```

(b) Composition of pre-algebras

```
trait ExprAssemble[Ctx, Out] extends ExprSig[Ctx ⇒ Out] {
  val alg1: CtxExprSig[Out, Ctx, Out]
  def Lit = n ⇒ ctx ⇒ alg1.Lit(n)(ctx)
  def Add =(e1, e2) ⇒ ctx ⇒ {
    val outL = e1(ctx)
    val outR = e2(ctx)
    alg1.Add(outL, outR)(ctx)
  }
}
```

(c) Assembly of pre-algebras

Figure 7. Depending on information about the same node.

no need to manually define instances for particular sorts as in Figure 5b. It thus allows us to abstract over the implementation details for composing traits and at the same time helps us to concentrate on the dependency structure between object algebras.

Equipped with this composition facility, we can combine Expr_ppOpt and Expr_value to yield a complete:

ExprSig[HasPP **with** HasValue]

4.2 Accessing other attributes on the current node

In the last subsection we have seen how to encode dependencies on attributes which are already computed for the children of the current node. The next step is to also allow modularization with respect to the computation already performed on the current node. An example where this proves useful is the pretty printer in Figure 2b that only prints the full subtree, if the value of the current node is not zero. We already have defined a component to compute the value of expressions that we now should be able to reuse.

To allow referencing already computed attributes such as value in this example we treat them as as input for the current computation. To this end, we instantiate the type parameter Out with a function type $Ctx \Rightarrow Out$ using the additional type parameter Ctx. Figure 7a shows the definition of the new context aware signature CtxExprSig.

When creating an algebra for CtxExprSig this way it is possible to separately specify which attributes should already be computed for the child nonterminals (E) as well as for the current node (Ctx) in order to compute the result (Out). As an example, here is a version of the above mentioned pretty-printer (Figure 2b) that requires to know the value of the current node:

```
trait Expr_ppOpt2 extends
  CtxExprSig[HasPP, HasValue, HasPP] {
  def Lit = n ⇒ ctx ⇒ HasPP(n.toString)
  def Add =(e1, e2) ⇒ ctx ⇒
    if(ctx.value ≡ 0) HasPP("0") else
      HasPP(e1.pp + "+" + e2.pp)
}
```

How can we compose algebras such as Expr_ppOpt_2 with other algebras after having defined them in a modular manner? Figure 7b illustrates the modifications to the composition trait ExprCompose necessary in order to account for the additional computational context. The lower bound $C_2 >: C_1$ **with** O_1 (read as "C_2 is some supertype of C_1 **with** O_1") restricts the required context of the second attribute alg_2 to either also be contained in the required context for the first attribute or to be part of the computed result of alg_1. The type of the composed attribute shows that it still requires the first context C_1 but now computes O_1 as well as O_2.

The implementation of the operations immediately follows from the types. For computing the first attribute the outer context is passed on to alg_1. The second attribute alg_2 then is invoked with a composition of the outer context and the output of the first algebra. The final result then again

InhSig[*Out*] *Extends Figure 7*

```
type CtxExprSig[−E, −Ctx, +Out] ...
type ExprSig[E] ...
trait PreExprSig[−E, +Out] ...
```

(a) Algebraic signature

```
trait ExprCompose[
  E1, C1, O1, E2, C2 >: C1 with O1, O2] ...
```

(b) Composition of pre-algebras

```
trait ExprAssemble[Ctx, Out] extends ExprSig[Ctx ⇒ Ctx with Out] {
  val alg1: CtxExprSig[Ctx with Out , Ctx, Out]
  val alg2: CtxInhSig[Ctx]
  def Lit = n ⇒ ctx ⇒ mix[Ctx, Out](ctx, alg1.Lit(n)(ctx) )
  def Add =(l, r) ⇒ ctx ⇒ {
    val outL =( alg2.Add1 andThen l)(ctx)
    val outR =( alg2.Add2 andThen r)(ctx)
    mix[Ctx, Out](ctx, alg1.Add(outL, outR)(ctx) )
  }
}
```

(c) Assembly of pre-algebras and their context transformed counterpart

```
type CtxInhSig[Ctx] = InhSig[Ctx ⇒ Ctx]
trait InhSig[+Out] {
  def Add1: Out
  def Add2: Out
}
```

(d) Algebraic signature of context decorator

Figure 8. Encoding inherited attributes by transformation of the grammar.

is composed. Our composition operator mix is right biased. Thus definitions in alg_2 can override the ones in alg_1.

ExprCompose has some nice properties. Algebras for the very same attribute can be composed together with the second one overriding and possibly decorating the first one due to the right biased nature of mix.

Dependencies to other attributes are articulated clearly and might improve modular reasoning. On the other hand, the very explicit composition of pre-algebras forces the user to perform a linearisation of the dependencies. Attributes have to be composed in topological order from left to right to allow attributes on the right to depend on already computed attributes to the left. The Scala type system enforces this linearisation, that is, it is impossible to encode attribute grammars with cyclic dependency graphs between attributes. This leaves two ways of implementing non-terminating traversals: Using Scala recursion or side effects inside an equation, or using Scala recursion or side effects to construct a traversal with a cyclic pointer structure on the Scala heap. Neither is considered to be in the image of our encoding, that is, starting from an attribute grammar with a pure and strongly-normalizing equation language, one cannot reach such non-terminating Scala programs by following our encoding.

An instance of CtxExprSig cannot immediately be used for folding since the types of the contravariant E and covariant $Ctx \Rightarrow Out$ carriers do not agree. Eventually, after having composed enough algebras using ExprCompose we will end up with an instance of CtxExprSig where $E = Out$ and the type parameter Ctx only resembles the residual state necessary for computation. Figure 7c illustrates how to close such a signature CtxExprSig $[Out, Ctx, Out]$ to ExprSig$[Ctx \Rightarrow Out]$ by passing down the ctx to all computations without modification, which is reminiscent of how reader monads are implemented. This process is named *closing* of the context algebra, since simple algebras cannot be composed with another context algebras anymore.

This specialized form of pre-algebras described by CtxExprSig is the final shape of our encoding of synthesized attributes in this paper. We also call such algebras *pre-function-algebras*, because they are pre-algebras whose domain has a function type.

5. Encoding Inherited Attributes

5.1 I-Attributed Grammars

Inherited attributes, are attributes that are defined in terms of the parent nodes. Hence, they require evaluating the dependencies on the parent before computing the children. In contrast to synthesized attributes, inherited attributes thus represent top-down computations during a traversal. By choosing the synthesized attribute to be a function we also can try to encode inherited attributes as synthesized attributes (Chirica and Martin 1979), similar to how a Reader monad in functional programming is used to represent context-dependent computations. However, this encoding is not suitable for a modular encoding of inherited attributes, because then the definition of the inherited attributes is tangled with the definition of synthesized attributes that depend on them.

The class of I-Attributed Grammars is rather artificial since inherited attributes on their own are not very useful.

Nevertheless, we consider an attribution of the grammar in Figures 1a and 1b by adding the inherited attribute $indent$ defined in Figure 9b. This attribute might be later used by a synthesized pretty-printer attribute to add a line break after emitting the $+$ and further indent the right-hand-side of the addition. A pretty-printer that would use this inherited attribute might have an equation of the form:

$$e_0.\text{pp} = e_0.\text{pp}\ \texttt{"+"}\ \texttt{"\textbackslash n"}\ e_0.indent\ e_1.\text{pp}\ \{\text{Add}\}$$

Using the alternative encoding described above, the pretty-printer attribute would become a function $\text{String} \Rightarrow \text{String}$, expecting the indentation as an argument. The equations for the indentation attribute $indent$ would need to be entangled with those for the pretty-printer; for instance, in the Add case, this would result in:

$$\begin{array}{l}\textbf{def}\ \text{Add} = (l, r) \Rightarrow indent \Rightarrow \\ \quad l(indent) + \texttt{"+"} + \texttt{"\textbackslash n"} + r(indent + \texttt{"\ \ "})\end{array}$$

If the inherited attribute is used in multiple other attribute definitions, we would even need to replicate the definition.

We have designed a modular and direct encoding of inherited attributes instead. We will first consider the simplest case where the equations for inherited attributes refer to the parent node only (but not to sibling nodes). To specify equations for inherited attributes modularly, we define a derived object algebra signature for inherited attributes. The signature is defined as a transformation of the above signature ExprSig: Inherited attributes are defined in terms of nonterminal occurrences on the right hand side of a production (i.e. e_1 and e_2). Thus it appears necessary to allow specification of equations for those nonterminal occurrences by turning them into operations of our transformed algebra. The result of this transformation is shown in Figure 8d.

Applying the same technique we have used to account for the computational context, CtxInhSig can be defined by instantiating the type parameter E with a simple function type $Ctx \Rightarrow Ctx$. This allows us to encode the class of AGs where an inherited attribute can only refer to the inherited attributes of its parent.

As above, the input type of the function $Ctx \Rightarrow Ctx$ resembles contextual information so we call this type parameter again Ctx. An important difference of the algebras encoding synthesized on the one hand and inherited attributes on the other hand is the usage of their outputs. The first immediately reflects the synthesized output of the overall program. The second serves purely as input for further computation.

Finally, Figure 8d shows the translation of the attribute grammar to this encoding of inherited attributes.

Standing on their own, inherited attributes do not provide much value. However, they are useful when being combined with synthesized attributes. As discussed above, inherited attributes can serve as context decorators for computation performed by synthesized attributes. For this purpose, we can

$e_0 \to n \quad \{\text{Lit}\}$	$e_1.indent = e_0.indent$
$e_0 \to e_1\ \texttt{"+"}\ e_2\ \{\text{Add}\}$	$e_2.indent = e_0.indent + \texttt{"\ \ "}$

(a) Grammar (b) Equations

$$\begin{array}{l}\textbf{val}\ Expr\_indent = \textbf{new}\ \text{CtxInhSig}[HasIndent]\ \{ \\ \quad \textbf{def}\ \text{Add}_1 = ctx \Rightarrow HasIndent(ctx.indent) \\ \quad \textbf{def}\ \text{Add}_2 = ctx \Rightarrow HasIndent(ctx.indent + \texttt{"\ \ "}) \\ \}\end{array}$$

(c) Algebra

Figure 9. Encoding an inherited attribute.

extend the assembly trait from Figure 7c to not only close a pre-function-algebra, but also take an inherited attribute as context decorator into account. The improved version of ExprAssemble in Figure 8c combines an instance of a pre-function-algebra CtxExprSig and an instance of its transformed inh-algebra CtxInhSig to produce a new instance of the closed algebra ExprSig. As can be seen from the type signatures, the synthesized attribute is provided with context information of type Ctx. For already processed children of a node both context, and the synthesized output are available (Ctx **with** Out). The implementation shows that the inherited attribute algebra just serves as decorator for the context provided to the synthesized attribute. At first the corresponding method on the transformed signature (such as $\text{alg}_2.Add1$) is called with the parent context, and then the result is passed as decorated context to the children (here l) in order to compute the synthesized output. [7].

Inherited attributes allow a decomposition of programs which is folklore to the attribute grammar community but represents a novel program structuring technique within general purpose programming languages. Our encoding of attribute grammars as object algebras brings these to software developers. At the same time, by means of embedding in a general purpose language, it offers new abstraction mechanisms to attribute grammar designers, as we will see in Section 7.

Before we talk about other ways to use and compose algebras for inherited attributes, we will first generalize our encoding such that inherited algebras can depend on attributes of siblings to the left. Such AGs are called *L-attributed*.

5.2 L-Attributed Grammars

In this subsection we will see how to encode L-attributed attribute grammars (L-AGs) within the framework of object algebras.

The pos attribute in Figure 11b represents the index of a node as assigned by a pre-order traversal. The second equation for the inherited pos attribute uses a synthesized attribute count of its sibling e_1, which is defined by the equations $e_0.\text{count} = e_1.\text{count} + e_2.\text{count}$ for the addition case and $e_0.\text{count} = 1$ for the literal case.

[7] The method andThen represents composition of functions and is part of the Scala standard library. It assures that $(f\ \text{andThen}\ g)(x) = g(f(x))$

PreInhSig[E, Out] *Extends Figure 8*

```
type CtxExprSig[−E, −Ctx, +Out] ...
type ExprSig[E] ...
trait PreExprSig[−E, +Out] ...
```

(a) Algebraic signature

```
trait ExprCompose[
  E1, C1, O1, E2, C2 >: C1 with O1, O2] ...
```

(b) Composition of pre-algebras

```
trait ExprAssemble[Ctx, Out] extends ExprSig[Ctx ⇒ Ctx with Out] {
  val alg1: CtxExprSig[Ctx with Out, Ctx, Out]
  val alg2: CtxInhSig[ Ctx with Out , Ctx, Ctx]
  def Lit = n ⇒ ctx ⇒ mix[Ctx, Out](ctx, alg1.Lit(n)(ctx))
  def Add =(e1, e2) ⇒ ctx ⇒ {
    val outL =(alg2.Add1 andThen e1)(ctx)
    val outR =(alg2.Add2(outL) andThen e2)(ctx)
    mix[Ctx, Out](ctx, alg1.Add(outL, outR)(ctx))
  }
}
```

(c) Assembly of pre-algebras and their context transformed counterpart

```
type CtxInhSig[ −E , −Ctx, +Out ] =
  PreInhSig[E , Ctx ⇒ Out ]

type InhSig[E] = PreInhSig[E, E]
trait PreInhSig[−E, +Out] {
  def Add1: Out
  def Add2: E ⇒ Out
}
```

(d) Algebraic signature of context decorators

Figure 10. Encoding of an L-attributed grammar.

Since the value of $e_2.\mathrm{pos}$ is defined in terms of the already computed attributes on e_1, this example grammar falls in the class of L-attributed grammars. This class of grammars allows attributes being defined in terms of already processed left neighbours to allow attribute evaluation in one top-down left-to-right pass (Bochmann 1976).

In order to support L-attributed grammars we need to adapt both the transformation of object algebra signatures into inherited attribute signatures as well as the assembly of an object algebra for synthesized attributes and its inherited attribute counterpart.

To address the first, we modify the transformation scheme to allow referencing left neighbours. At the same time, in order to allow modular composition of multiple inherited attributes, where dependencies can be expressed for the current node, we use pre-function-algebras from Section 4.

```
type CtxInhSig[−E, −Ctx, +Out] =
  PreInhSig[E, Ctx ⇒ Out]
```

The important difference between Figures 8d and 10d is the signature of operation Add_2. The argument of type E represents the already calculated attributes for the left neighbour e_1 within the addition. Again, this contravariant occurrence of a sort motivates the usage of pre-algebras.

In Figure 11c it becomes clear that the translation from the attribute grammar to an instance of the pre-function-algebra now is straightforward. The main difference to the attribute grammar specification is the list of type parameters allowing to interface with other attribute implementations to support modular definition of a larger system.

In order to compose the above inherited attribute algebra with a synthesized attribute algebra, the changes in the transformation also have to be reflected in the signature of the ExprAssemble trait, see Figure 10c. Since we know that for already traversed children the context as well as the synthesized attributes have been calculated, the pre-algebraic input sort of the inherited attribute algebra alg_2 has to be changed to Ctx **with** Out. It now becomes visible that at assembly time the type members of synthesized and inherited attribute algebras only differ in their *result type*. Synthesized attribute algebras calculate the overall output while inherited attribute algebras compute necessary context information.

Comparing the implementation of Add with the previous one the reader will notice only one small change. In order to compute the context information for the right hand side of the addition $outL$, the computation result of the left neighbour, is passed to the call of $\mathrm{alg}_2.\mathrm{Add}_2$.

Since we encoded inherited attributes as pre-function-algebras, the same infrastructure for algebra composition can be used for inherited as well as synthesized attribute algebras. The only difference is the concrete shape of the algebra in terms of operations. As we have seen in this

$e_0 \to n$	{Lit}	$e_1.\mathrm{pos} = e_0.\mathrm{pos} + 1$
$e_0 \to e_1$ `"+"` e_2	{Add}	$e_2.\mathrm{pos} = e_0.\mathrm{pos} + e_1.\mathrm{count} + 1$

(a) Grammar (b) Equations

```
val Expr_pos = new CtxInhSig[HasCount, HasPos, HasPos] {
  def Add1 = ctx ⇒ HasPos(ctx.pos + 1)
  def Add2 = e1 ⇒ ctx ⇒ HasPos(ctx.pos + e1.count + 1)
}
```

(c) Algebra

Figure 11. Encoding of an L-attributed grammar.

section, the algebraic signature used to encode inherited attributes is a straightforward transformation of the algebraic signature encoding synthesized attributes.

5.3 Three ways to construct sentences

In the beginning of Section 3, we have seen one way to encode sentences in the language defined by the context free grammar, namely as a function that is parametric in the object algebra:

```
def threeplusfive[E](alg: ExprSig[E]) =
  alg.Add(alg.Lit(3), alg.Lit(5))
```

This is the typical style for Church-encodings known from previous works, such as polymorphic embedding or the "finally tagless" approach. Due to the ExprAssemble trait, we can continue to use this approach even in the presence of inherited attributes. ExprAssemble wires the two object algebras representing synthesized and inherited attributes and closes them, resulting in an instance of ExprSig. Here both covariant and contravariant occurrences of the carrier are the same.

A second possibility is to have a physical tree and then define a function that folds the algebra over the tree:

```
trait Expr {
  def fold[E](alg: ExprSig[E]): E
}
case class Lit(n: Int) extends Expr {
  def fold[E](alg: ExprSig[E]): E = alg.Lit(n)
}
case class Add(l: Expr, r: Expr) extends Expr {
  def fold[E](alg: ExprSig[E]): E =
    alg.Add(l.fold(alg), r.fold(alg))
}
```

This use of our encoding follows the Visitor pattern. The algebra corresponds to the (internal) visitor, and fold corresponds to the *accept* method often used with the Visitor pattern. This style enables easy use of our with physical trees such as Expr.

However, it is important to note that also in the Church encoded style, such as the sentence threeplusfive, trees are created under the cover: a tree of function closures representing the term is constructed in memory. This is undesirable if the goal is to construct an algorithm that works like a one-pass compiler, that is, the output is constructed just in time while the input is read, and the memory consumption is independent of the input size.

It is, however, possible to encode sentences in a third, different way, namely by partially evaluating ExprAssemble with a concrete input. For instance, we can encode the program threeplusfive in this style as follows.

```
def threeplusfive2[Ctx, Out]
  (alg1: CtxInhSig[Ctx with Out, Ctx, Ctx],
   alg2: CtxExprSig[Ctx with Out, Ctx, Out])
     : Ctx ⇒ Ctx with Out =
  ctx ⇒ {
    val in3 = alg1.Add1(ctx)
    val out3 = alg2.Lit(3)(in3)
    val all3 = mix[Ctx, Out](in3, out3)
    val in5 = alg1.Add2(all3)(ctx)
    val out5 = alg2.Lit(5)(in5)
    val all5 = mix[Ctx, Out](in5, out5)
    mix[Ctx, Out](ctx, alg2.Add(all3, all5)(ctx))
  }
```

Once the initial context ctx is passed to the function, all function closures are immediately evaluated. If this new representation of the sentence was used in a parser for the language (and given an initial context Ctx), the attributes would be computed while the parser runs.

Writing programs in this style is to cumbersome to be done for concrete input. However, handwritten parsers can easily be adapted to represent the parsed program as calls to the object algebras. As follow up work, a set of parser combinators could be designed that automatically performs the above partial evaluation of the parsed input.

When calling the algebra functions during parsing, it is important that the grammar class recognized by the parsing technology fits the attribute grammar class (Deransart et al. 1988, Sec. 4.3). For instance, in a recursive-descent parser, one would not know whether to call the $\mathrm{alg}_1.\mathrm{Add}_1\ (ctx)$ function when a numeral is encountered, because it is not yet clear whether it will turn out to be the operand of an addition. When the grammar is refactored to $LL(k)$, the problem does not occur anymore. The problem can also be avoided by using the first variant described in this subsection, but then it would no longer be a true one-pass compiler.

5.4 Summary

In the preceding three sections 3 to 5 we have seen how different classes of attribute grammars correspond to different encodings of object algebras. The simplest class of S1-attributed grammars corresponds to a single sorted signature, known from polymorphic embedding. Multiple, isolated attributes defined in parallel can be composed using a composition trait. In order to support modular definition of synthesized attributes we introduced pre-function-algebras, which allow to separately interface with the requirements on child computation as well as computation on the same nonterminal. Inherited attributes can be encoded by a transformation of the algebraic signature, introducing an operation for every right hand side occurrence of a nonterminal. Using pre-function-algebras for both synthesized as well as inherited attributes and adapting the assembly of the two we can encode L-attributed AGs in the framework of object algebras.

6. Formalization

In this section, we formalize the encoding of L-attributed grammars that we introduced by example in Section 3. This formalization allows us to

- unambiguously communicate the details of our encoding,
- meaningfully discuss modularity, scalability and compositionality of the encoding, and
- support experiments with the encoding by automating the generation of code fragments.

The formalization is based on a generator that we wrote to support our experiments with the encoding. The generator accepts the abstract syntax of a context-free language as input, and generates the Scala source code of the various algebra interfaces and combinators as output. In addition to the features exemplified in the previous section, this formalization as well as the generator also support grammars with multiple nonterminal symbols.

6.1 Grammars and Signatures

We formalize how to encode the L-attribution of a given context-free grammar. For each nonterminal symbol in the grammar, we want to extract the signatures of the corresponding algebras for computing synthesized and inherited attributes.

For this extraction, we have to distinguish nonterminal symbols whose meaning is defined by the grammar, terminal symbols, and built-in symbols that denote primitive types such as integers or strings. With respect to attribution, the difference between these three categories of symbols is what information the corresponding derivations contain: Nonterminal symbols carry attributes, terminal symbols contain no information, and built-in symbols contain a single value of the corresponding primitive type. We also have to label each production in the grammar with a constructor symbol. Since we extract the same information from all non-terminal symbols, we don't have to select a start symbol.

For our purposes, we therefore define that a context-free grammar G is a quintuple (N, T, B, C, P) where N is the set of nonterminal symbols, T is the set of terminal symbols, B is the set of built-in types, C is the set of constructor symbols and

$$P \subset N \times (N \cup T \cup B)^* \times C$$

is the set of productions so that there is exactly one production for every symbol $c \in C$. We use the meta-variable s for symbols in $N \cup T \cup B$ and write productions as

$$n \to s_1 \dots s_k \quad \{c\}$$

We call the nonterminal symbol n the head and the sequence of symbols $s_1 \dots s_k$ the body of the production. We say that a nonterminal symbol n_0 depends on a nonterminal symbol n_1 if n_1 occurs in the body of a production with n_0 as head. For example, an extension of the grammar of the example language from Section 3 is shown in Figure 12a. It uses the nonterminal symbols E and S, the terminal symbols `"+"`, `"="`, `";"`, and `"return"`, and built-in types X for variable names and N for numbers. The nonterminal symbol S depends on S and E, but E only depends on itself.

$E \to N$	$\{\text{Lit}\}$
$E \to E$ `"+"` E	$\{\text{Add}\}$
$E \to X$	$\{\text{Var}\}$
$S \to X$ `"="` E `";"` S	$\{\text{Set}\}$
$S \to$ `"return"` E `";"`	$\{\text{Exp}\}$

(a) Grammar

$\text{Lit} : N$	$\to E$
$\text{Add}: E \times E$	$\to E$
$\text{Var} : X$	$\to E$
$\text{Set} : X \times E \times S$	$\to S$
$\text{Exp}: E$	$\to S$

(b) Operations

Figure 12. Example grammar and algebraic signature.

A context-free grammar for a language directly corresponds to an algebraic signature for the language. For the signature, only the abstract syntax of the language is relevant, so we ignore the terminal symbols from the grammar. The other kinds of symbols are treated as follows:

- Constructor symbols are used as function symbols,
- Nonterminal symbols are used as sorts and
- Built-in symbols are used as predefined types.

With respect to algebraic signatures, the difference between the latter two is that the carrier types for nonterminal symbols are left open for each algebra to decide, whereas the carrier types for predefined types are fixed for all algebras of the same signature.

For our purposes, we also define that an algebraic signature Σ is a quadruple (N, B, C, D) where N, B and C are defined as for grammars, and

$$D \subset C \times (N \cup B)^* \times N$$

is the set of type declarations so that there is exactly one type declaration for every symbol $c \in C$. In the context of signatures, we sometimes call N the set of sorts. We use the meta-variable τ for symbols in $N \cup B$ and write type declarations as:

$$c : \tau_1 \times \dots \times \tau_k \to n$$

We call n the return type and $\tau_1, \tau_2, \dots$ the argument types of c. Every grammar (N, T, B, C, P) induces an algebraic signature (N, B, C, D) with D defined as follows.

$$D = \{c : \tau_1 \times \tau_2 \times \dots \to n \mid (n \to s_1 s_2 \dots \{c\}) \in P, \\ \tau_1 \tau_2 \dots \text{ is } s_1 s_2 \dots \text{ without terminal symbols}\}$$

For example, the signature induced by the grammar from Figure 12a is shown in Figure 12b. This signature is well suited for the implementation of S1-attributed algebras as discussed in Section 3.1, but in order to modularly encode multiple synthesized and inherited attributes, we have to transform this signature into a variant for context-sensitive operations.

	already computed	about to compute
about subtree	n	n'
about context	$\widetilde{n}$	$\widetilde{n}'$

(a) Naming scheme for sorts

$$
\begin{aligned}
\mathrm{Lit} &: N && \times \widetilde{E} \to E' \\
\mathrm{Add}&: E \times E && \times \widetilde{E} \to E' \\
\mathrm{Var} &: X && \times \widetilde{E} \to E' \\
\mathrm{Set} &: X \times E \times S && \times \widetilde{S} \to S' \\
\mathrm{Exp}&: E && \times \widetilde{S} \to S'
\end{aligned}
$$

(b) Context-sensitive operations

$$
\begin{aligned}
\mathrm{Add}_1&: && \widetilde{E} \to \widetilde{E}' \\
\mathrm{Add}_2&: E && \times \widetilde{E} \to \widetilde{E}' \\
\mathrm{Set}_1 &: X && \times \widetilde{S} \to \widetilde{E}' \\
\mathrm{Set}_2 &: X \times E && \times \widetilde{S} \to \widetilde{S}' \\
\mathrm{Exp}_1&: && \widetilde{S} \to \widetilde{E}'
\end{aligned}
$$

(c) Context decorators

Figure 13. Adapting the signature to encode context-sensitive operations and context decorators.

6.2 Context-Sensitivity and Context Decorators

In order to allow operations to compute and access the context of each node, we want to transform the above developed signature for context-free operations into two different signatures: A signature of context-sensitive operations that allows operations to access the context of the current node, and a signature of context decorators that specify the context of each child of a node in terms of the information about the child's left neighbours as well as the context of the current node, that is, the parent of the child in question. This also motivates the name "context decorators": They allow decorating the context of the current node with additional information.

Before we can present the signature transformation, we have to introduce additional sorts to distinguish information available for different parts of a tree, as seen from the "current node" during a tree traversal. We distinguish information that is already computed from information that we are about to compute, and information about a subtree from information about the context of a subtree, for a total of four different sorts per nonterminal symbol. These fine-grained distinctions allow us to specify exact interfaces for the components that together specify a tree traversal.

For every nonterminal n in the grammar, we use the sort n to represent information about already traversed n-children of the current node, $\widetilde{n}$ to represent information about the context of the current n-node, n' to represent the information about the current n-node we are about to compute, and $\widetilde{n}'$ to represent information about the context of n-children of the current node we are about to traverse. For example, the nonterminal symbol E induces the sorts E, E', $\widetilde{E}$, and $\widetilde{E}'$. Figure 13a illustrates how this naming scheme for sorts encodes the two dimensions of "already computed" vs. "about to compute" on the one hand, and information about subtrees vs. information about the context on the other hand.

The signature of context-sensitive operations contains the same function symbols as the signature from Section 6.1, that is, one function symbol per production of the grammar. This relates to the fact that for synthesized attributes, we have to provide one equation for each occurrence of a nonterminal symbol in the head of a production, that is, one equation for each production. If a function symbol f in the signature of context-free operations has the type declaration

$$f\colon \tau_1 \times \ldots \times \tau_k \to n$$

then the corresponding function symbol in the signature of context-sensitive operations has the type declaration

$$f\colon \tau_1 \times \ldots \times \tau_k \times \widetilde{n} \to n',$$

that is, we add an argument to account for the already known information about the current node, and we change the result to mark that we are about to compute it. For example, Figure 13b shows the signature of context-sensitive operations that correspond to the context-free operations in Figure 12b. The alert reader might have noticed that earlier in this paper in Section 4.2 a similar development lead to the introduction of pre-function-algebras.

The signature of context decorators contains different function symbols than the other signatures. We need one context decorator for every context of a child we have to compute. That is, one for every nonterminal argument of any function symbol in the signature of context-free operations. This relates to the fact that for inherited attributes, we have to provide one equation for each occurrence of a nonterminal symbol in the body of a production. If a function symbol f in the signature of context-free operations has the type declaration

$$f\colon \tau_1 \times \ldots \times n_i \times \ldots \times \tau_k \to n$$

then the corresponding function symbols in the signature of context decorators have the type declarations

$$f_i\colon \tau_1 \times \ldots \times \widetilde{n} \to \widetilde{n}'_i$$

for $i = 1 \ldots k$. In other words, we create a symbol f_i if the type of the ith argument of f is a nonterminal. This symbol f_i first takes $i-1$ arguments of types $\tau_1, \ldots, \tau_{i-1}$ and an additional argument to account for the context of the current node. We then set the return type to $\widetilde{n}'_i$ in order to reflect that we are computing the context for the traversal of the ith child. For example, Figure 13c shows the signature of context decorators that correspond to the context-free operations in Figure 12b.

6.3 Object Algebras and Combinators

The context-sensitive operations and the context decorators play different roles during computation of the attributes for

```
trait SSig[E, S] {
  def Set(x: X, e: E, s: S): S
  def Exp(e: E):             S
}
```
(a) Direct translation of the context-free variant.

```
trait CtxSSig[−E, −S, −S̃, +S′] {
  def Set(x: X, e: E, s: S, s̃: S̃): S′
  def Exp(e: E,              s̃: S̃): S′
}
```
(b) Direct translation of the context-sensitive variant.

```
trait CtxSSig[−E, −S, −S̃, +S′] {
  def Set(x: X, e: E, s: S): S̃ ⇒ S′
  def Exp(e: E):             S̃ ⇒ S′
}
```
(c) Currying to separate the common part from the varying part.

```
trait PreSSig[−E, −S, +Out] {
  def Set(x: X, e: E, s: S): Out
  def Exp(e: E):             Out
}
```
(d) Abstraction over the varying part

```
type SSig[E, S] = PreSSig[E, S, S]
type CtxSSig[−E, −S, −S̃, +S′] = PreSSig[E, S, S̃ ⇒ S′]
```
(e) Instantiations for the context-free and the context-sensitive variant.

Figure 14. Different variants of encoding the object algebra that contains Set and Exp.

an abstract syntax tree. The former account for the computation of synthesized attributes, and the latter account for the computation of inherited attributes. However, this difference only matters for the final assembly of a system of attributes, where we have to compose definitions of context-sensitive operations and the corresponding definitions of context decorators to compute both synthesized and inherited attributes in the same traversal over the syntax tree.

Apart from their role in the final assembly, the type declarations of the different kinds of operations are quite similar. They all accept a series of arguments of the form n with information about the children of the current node and a final argument of the form $\widetilde{n}$ with information about the context of the current node. And they all return something of the form n' (or $\widetilde{n}'$) with some information that is about to be computed. These similarities between the type declaration allow us to encode both context-sensitive operations and context decorators as object algebras in the same way, with the same combinators that allow for the same modular decomposition mechanisms.

To exploit the similarities between the operations, we group operations into object algebra signatures based on their return type and on the type of their context argument. For example, the function symbols in Figures 13b and c are grouped into five object algebra signatures: $\{\mathrm{Lit}, \mathrm{Add}, \mathrm{Var}\}$, $\{\mathrm{Set}, \mathrm{Exp}\}$, $\{\mathrm{Add}_1, \mathrm{Add}_2\}$, $\{\mathrm{Set}_1, \mathrm{Exp}_1\}$ and $\{\mathrm{Set}_1\}$. This grouping has the benefit that the methods in an object algebra signature have a more uniform type, which makes it easier to define the object algebra combinators.

Since all methods in an object algebra share the same return type and the type of the context argument, we curry the method declarations in order to abstract over the commonality. This also allows us to express both the signature of context-free operations and the signature of the corresponding context-sensitive operations as instantiations of the same generic interface. Here, the context-sensitive variant corresponds to pre-function-algebras as introduced in Figure 7a.

For example, Figure 14 shows the various stages of this abstraction process for the object algebra that contains Set and Exp. Please note that the type X represents the built-in type for names as introduced above.

Given the translation from grammars to context-sensitive operations and context decorators described in Section 6.2, we know that for every distinct nonterminal h that occurs in the head of a production, we need one object algebra signature because the h productions give rise to context-sensitive operations with type declarations $\ldots \times \widetilde{h} \to h'$ and all of them should be grouped in one signature. We use the name of the nonterminal h in the naming scheme for the Scala definitions related to this object algebra, that is, we generate $\mathrm{Pre}h\mathrm{Sig}$ and define $h\mathrm{Sig}$ and $\mathrm{Ctx}h\mathrm{Sig}$ in terms of $\mathrm{Pre}h\mathrm{Sig}$.

Additionally, for every distinct nonterminal b that occurs in the body of a production of h, we need one object algebra signature for the context decorators between these two nonterminals, because the h productions with b occurrences give rise to context transformers with a type declaration $\ldots \times \widetilde{h} \to \widetilde{b}'$ and all of them should be grouped in one signature. We use the phrase "b in h" in the naming scheme for the Scala definitions related to this object algebra, because these definitions relate to the fact that the nonterminal b occurs in the body of a production of h, that is, we generate $\mathrm{Pre}b\mathrm{in}h\mathrm{Sig}$ and define $b\mathrm{in}h\mathrm{Sig}$ and $\mathrm{Ctx}b\mathrm{in}h\mathrm{Sig}$ in terms of $\mathrm{Pre}b\mathrm{in}h\mathrm{Sig}$.

The definitions generated for context-sensitive operations and the definitions generated for context decorators only differ in the naming scheme. We use the meta variable Obj to abstract over the naming scheme, that is, we talk generally about $\mathrm{Pre}Obj\mathrm{Sig}$, $Obj\mathrm{Sig}$, and $\mathrm{Ctx}Obj\mathrm{Sig}$. Figure 15a shows how the code generated for an $\mathrm{Ctx}Obj\mathrm{Sig}$ looks like in general. This common template allows us to uniformly generate object algebra combinators for these object algebras, independently of whether they represent context-sensitive operations or context decorators.

An object algebra $\mathrm{Ctx}Obj\mathrm{Sig}$ represents only a fragment of a tree traversal. To compose these fragments into larger components, we generate a $Obj\mathrm{Compose}$ trait that can be used to compose two $\mathrm{Ctx}Obj\mathrm{Sig}$ instances so that during the traversal, operations in the second algebra can access the information computed by the first algebra. The $Obj\mathrm{Compose}$ combinator is shown in Figure 15b.

$\mathrm{Ctx}\mathit{Obj}\mathrm{Sig}[-n_1, ..., -\mathit{Ctx}, +\mathit{Out}]$

```
type CtxObjSig[-n1, ..., -Ctx, +Out] =
  PreObjSig[n1, ..., Ctx ⇒ Out]
type ObjSig[n1, ..., ni] =
  PreObjSig[n1, ..., ni, ni]
trait PreObjSig[-n1, ..., +Out] {
  def Ctor1:(τ11, τ12, ...) ⇒ Out
  def Ctor2:(τ21, τ22, ...) ⇒ Out
  ...
}
```

(a) Object algebra signature.

```
trait ObjCompose[A1, B1, ..., C1, Out1, A2, B2, ..., C2 :> C1 with O1, O2]
    extends CtxObjSig[A1 with A2, B1 with B2, ..., C1, O1 with O2] {
  val alg1: CtxObjSig[A1, B1, ..., C1, O1]
  val alg2: CtxObjSig[A2, B2, ..., C2, O2]
  def Ctor1 =(x1, x2, ...) ⇒ ctx1 ⇒ {
    val out1 = alg1.Ctor1(x1, x2, ...)(ctx1)
    val ctx2 = mix[C1, O1](ctx1, out1)
    val out2 = alg2.Ctor1(x1, x2, ...)(ctx2)
    mix[O1, O2](out1, out2)}
  ...
}
```

(b) Composition.

```
trait n0Assemble[n0, ñ0, n1, ñ1, ...] extends n0Sig[ñ0 ⇒ ñ0 with n0, ñ1 ⇒ ñ1 with n1, ...] {
  val algn0: Ctxn0Sig[ñ0 with n0, ñ1 with n1, ..., ñ0, n0]
  val algn0inn0: Ctxn0inn0Sig[ñ0 with n0, ..., ñ0, ñ0]
  val algn1inn0: Ctxn1inn0Sig[ñ0 with n0, ..., ñ0, ñ1]
  ...
  def Ctor1 =(x1, x2, ...) ⇒ ctx ⇒ {
    val y1 = ...; val y2 = ...; ...   // see (d)
    val y = algn0.Ctor1(y1, y2, ...)(ctx)
    mix[ñ0, n0](ctx, y)
  }
}
```

(c) Skeleton of n_0Assemble.

```
// if τi is a built-in type:
val yi = xi
// if τi is a nonterminal symbol:
val yi =(algτiinn0.fi(y1, ..., yi−1) andThen xi)(ctx)
```

(d) The computation of y_i depends on the type τ_i of the ith argument of f.

Figure 15. General form of object algebra signatures, composition of object algebras and their final assembly.

Finally, the assembly of all context-sensitive operations and context decorators for a nonterminal n_0 is shown in Figure 15c. Note that this trait extends n_0Sig which is important for two reasons: On the one hand, we cannot further compose instances of this trait, because object algebra composition is not defined for n_0Sig. But on the other hand, we can use instances of this trait to fold over ASTs or polymorphically embed sentences, as discussed in Section 5.3. Therefore, the intended usage is that in a compiler that is modularized with object algebras, each module exports object algebras with context-sensitive operations or context decorators. Modules are free to compose such object algebras and return the result, but the final closing assembly should be deferred to the main program.

7. Modularizing a Compiler

To evaluate our claims made in the introduction we conducted a case study and translated a one-pass compiler for a subset of C into our encoding. The chosen compiler *C0* is a handwritten, monolithic compiler used for educational purposes at Aarhus University, Denmark[8]. The authors were not involved in the design of the compiler. *C0* is restricted to a subset of the C language consisting of only integer types, a few control structures (while, if, return), function declaration and definition as well as basic I/O. It is implemented in Java as a recursive decent parser with inline semantic actions using function arguments to pass down context information. Due to inlining various aspects of the implementation are highly entangled. The usage of arguments to pass down context information hinders modularity and extensibility.

By translating the *C0* compiler to our encoding we were able to support our claims from Section 1, showing that the encoding is modular, scalable and compositional. In particular it showed that *a*) attributes can be defined and type-checked separately since they are encoded as traits communicating their dependencies over type parameters. The encoding of the dependencies within the type system helps to safely assemble the overall system after modularization. *b*) The encoding scales linearly in the number of nonterminals. For every dependent nonterminal only a constant amount of type parameters has to be introduced. In particular only immediate child nonterminals have to be considered. *c*) The embedding into a high level language allows introducing novel abstractions not anticipated by the origi-

[8] The source is available at `http://cs.au.dk/~mis/d0vs/Czero.java`

#	*AG Artifact*	*Implementation Artifact*	*LoC*		*LoC*
		Generic	140	*Generic or could be generated*	698
9	*Nonterminals*	Signatures and Combinators	534		
		Trees	24		
9	*Nonterminal Dependencies*	Composition and Assembly	101	*Semantic, handwritten code*	**352**
5	*Attributes*	Attribute Interfaces	32		
20	*Equationsets*	Equation implementations (Algebras)	191		
		Utility	28		
		Parser	423	*Syntactic, handwritten code*	**540**
		Scanner	117		
		Bytecode Prelude	25	*Other*	30
		Main	5		
		Total	1620		1620

Table 3. Lines of code for the *C0* case study grouped by category

nal encoding. We have been able to abstract over lists and thus reuse attribute definitions for several instances of lists.

In the remainder of this section, we discuss the evaluation of these claims in further detail.

7.1 Modularity

Leveraging the modularity of our encoding the implementation of the case study spans over around 25 source files. The attribute equation sets, representing the implementation of attributes, are grouped on a nonterminal basis. With our encoding, this choice is up to the developer and other clustering dimensions (for instance, by attribute) are equally possible.

The dependencies between the different components are kept minimal. Two pairs of nonterminals are mutually recursive and hence required some attention. The different nonterminal algebras can be implemented without referencing concrete instances of other nonterminals. References to other nonterminals are only required in order to delegate folding over physical trees to the dependent nonterminals. In order to assemble dependent nonterminals for folding, we used Scala's `lazy val` to model the mutually recursive structure.

7.2 Scalability

In order to experimentally evaluate the scalability of our approach, we measured the lines of code for both the original monolithic program as well as the translation into our modular encoding. An overview of the results can be found in Table 4 for the original source code and in Table 3 for our translation. In Table 3 the center column illustrates how many lines of code each implementation artifact provides. Additionally, the main column is annotated from both sides. The left hand column relates the attribute grammar artifacts to the implementation artifacts and thereby provides insights on how the approach scales in the number of AG artifacts. In order to assist better estimating the implementation effort, the artifacts are grouped in the right hand column.

The four types of attribute grammar artifacts in the table represent possible dimensions of scalability as follows.

Implementation Artifact	*LoC*		*LoC*
Entangled Parser/Compiler	351	*Semantic and*	**580**
Scanner	229	*Syntactic*	
Bytecode	213	*Other*	227
Main	14		
Total	807		807

Table 4. Lines of code for original *C0* Compiler (in Java).

Nonterminals Our encoding of attribute grammars is scalable in terms of the overall number of nonterminals occurring in a grammar. Our encoding of the *C0* compiler is based on a grammar with 9 nonterminals. For every nonterminal only a moderate amount code, that in fact can be generated, is necessary. The case study has been conducted in parallel to the implementation of a code generator for generating this support code. It hereby guided the design of the generator and also assisted the formalization as presented in Section 6. Since the generator had not been completed when implementing the case study, the necessary support code is still handwritten rather than generated. After having finished the code generator, the support code now could be automatically generated given a description similar to that of Figure 12b as input. Thus adding a nonterminal does not require any additional handwritten code.

Nonterminal Dependencies Adding nonterminals to the right hand side of a production has two effects in our encoding. Firstly, in order to use an algebra for folding, all its dependent nonterminal algebras have to be referenced and thus the assembly code might require adaption. Secondly, every dependent nonterminal contributes at most 2 additional type parameters in the corresponding context algebra as can be seen in Section 6. In our case the assembly of all algebras required 101 LoC.

Attributes Adding a new attribute only requires writing an attribute interface like the one for $\mathrm{HasValue}$ introduced in Section 3. More interestingly, adding an attribute as dependency when implementing an algebra amounts to adding it to one of the contravariant type parameters. Only when composing algebras it is then checked whether attribute dependencies are fulfilled.

Equation Sets In our implementation a set of equations mostly coincides with the implementation of an object algebra signature or its context variant. We implemented the 5 attributes for the corresponding nonterminals by providing the necessary definitions in 20 equation sets, hereby contributing 191 LoC. We were able to generically implement some reoccurring attribute definition schemes such as passing on the context

$$\mathbf{def}\ \mathrm{add}(e_1, e_2) \Rightarrow ctx \Rightarrow ctx$$

using the last computed synthesized attribute as result for inherited attributes

$$\mathbf{def}\ \mathrm{add}_2 = e_1 \Rightarrow ctx \Rightarrow e_1$$

and decorating the provided context using a function f.

$$\mathbf{def}\ \mathrm{add}(e_1, e_2) \Rightarrow ctx \Rightarrow f(ctx)$$

Abstracting over these definition schemes enabled us to use them as a simple function call during assembly. Six of the 20 equation sets are hence implemented as one liners.

Parsing The parser has been ported to Scala and has been split into one component per nonterminal. The computation of inherited and synthesized attributes is performed during parsing following the "partial evaluation" usage scheme of $\mathrm{threeplusfive}_2$ in Section 5.3. The scanner is implemented as a lazy stream of tokens and thus a few lines of code could be saved here.

Other This section includes a static bytecode prelude, emitted by both implementations but packed in its representation in Scala and the implementation of the main entry point for the compiler.

Not counting generatable, generic code as well as the auxiliary code (bytecode prelude and main) the handwritten code in our encoding amounts to overall 892 LoC[9]. Compared with 580 LoC of the original entangled parser & compiler implementation we are confident in the scalability of our encoding.

It is not clear how to compare lines-of-code meaningfully across languages. A strong argument for the scalability of our approach, however, is given by the fact that there exists a simple, direct mapping between each artifact of the attribute grammar and the implementing artifacts in Scala.

[9] As illustrated in Section 7.1 our implementation spans over 25 files. This modularity comes with a cost of additional package declarations and import statements ranging from 2 up to 10 LoC per file. This, as well as additional signatures for traits and methods, are the price for reasonably modularized software artifacts.

7.3 Compositionality

Following the translation scheme as formalized in Section 6 for 9 nonterminals we introduced the corresponding algebraic signatures and their context decorator variants as Scala traits. The algebras for statements and functions are parametrized with 3 sorts representing the maximum in this case study. The count of operations varies between 1 and 5 per signature. Sixteen different attribute equation sets implementing 5 different attributes for corresponding algebraic signatures have been defined independent of each other. Every equation is represented as a method implementation in the respective algebra. Since the dependencies are stated explicitly in the type signature of the algebras they can be type checked and compiled individually.

Since our encoding is based on pure embedding into a general purpose language, high level structuring features of the host language can be used to organize the code base. By conducting this case study we discovered a dimension of modularity we did not anticipate: *Parametrized Nonterminals*. The most common example of parametrized nonterminals are lists. While implementing *C0* we encountered four instances of lists for the nonterminals $Declaration$, $Statement$, $Expression$ and $Function$ all structurally very similar. This becomes immediately visible when inspecting the signature for one particular list.

$$\begin{array}{l}\mathbf{trait}\ \mathrm{DeclListSig}[-Decl, -List, +Out]\ \{\\ \quad \mathbf{def}\ \mathrm{Empty}\colon Out\\ \quad \mathbf{def}\ \mathrm{Cons}\colon(Decl, List) \Rightarrow Out\\ \}\end{array}$$

Comparing the algebraic signatures of the four instances of lists we notice that they are alpha equivalent and hence the sort $Decl$ in the example above can just be renamed to $Elem$ to account for its generality. Thus, the algebraic signature of $List$ is not special. It is just a multi sorted abstract algebra signature as introduced in Section 6. Also comparing the necessary support code for the four instances we notice that only the fold function needs modification, since it has to be parametrized by the type of syntax to fold over.

Hence, due to the embedded nature of our encoding, we were able to introduce this novel abstraction without having to modify our translation scheme from attribute grammars to object algebras. Using this newly introduced abstraction one can implement generic attributes and reusable attribute implementations for lists. Examples for such attributes include list size and other operations available on monoids such as sum or concatenation. By means of parametrized nonterminals we were able to abstract over 4 out of 9 nonterminals and thus reduce some complexity of the implementation while facilitating reuse.

In this section we have seen evidence that our encoding is modular, scalable and compositional. Our experience with the development of the C0 case study suggest that it might even be practical to structure programs in this way by hand,

but further experiments have to investigate the notational and conceptual overhead more. We are confident that our embedding opens the door to carry over concepts, methods and techniques from the area of attribute grammars to the area of programming with visitors and Church encodings such as object algebras.

8. Other Forms of Attributes

We sketch briefly how some advanced AG features could carry over to our encoding. Our experiments with these features can be found at the URL given in Section 1.

Higher Order Attributes allow the result of an attribute to be an attributed tree (Vogt et al. 1989). Our approach is based on Church encodings and hence a higher order attribute can be encoded as a Church encoded value with a parameterized return type. To construct attributed trees within an attribute equation a technique similar to example `threeplusfive` in Section 5.3 can be used. Just as in this example, the attribute implementation has to be polymorphic in the type of the algebra used for creating the Church encoded result.

Forwarding enables default definitions for attributes by redirection to another implementation (Van Wyk et al. 2002). Explicit forwarding, that is forwarding for specific attributes, can be implemented as reusable decorator by delegating the attribute implementation to a given higher order attribute.

Reference Attributed Grammars allow attributes to refer to (the attributes of) other nodes in the tree (Hedin 2000). We have reimplemented the TINY RAG from Hedin in our object algebra style. The encoding does not change when reference-valued attributes are present, but the objects containing the computed attributes refer to each other. To allow this, it was sufficient to turn the attribute objects into Scala `lazy val`s and make the involved `mix` functions non-strict.

Parameterized Attributes accept arguments to compute their value (Hedin 2000). By using first class functions as attribute result type, parameterized attributes can be emulated quite easily. Since we embedded the attribute specification into a general purpose programming language it seems straightforward to add support for memoization within the attribute trait in order to avoid recomputation.

Circular Attributes enable iterative fix point construction by recursive definitions (Magnusson and Hedin 2007). Oliveira et al. (2013) present an object algebra for computing first and follow sets of a context-free grammar. We believe that their code can be translated to our encoding and that it can be generalized to support arbitrary circular attributes.

More General Dependencies. We focus on L-attributed grammars, but in the general case, the dependencies between attributes can be arbitrary; as long as the dependency graph is acyclic, the attributes can be evaluated in topological order. A standard way to allow arbitrary dependencies is to make attribute evaluation lazy (Johnsson 1987), and this would work in our approach, too. In future work, we also consider explicit sequencing operators for multiple passes, such that well-definedness is guaranteed and the programmer has better control over the evaluation order.

9. Related Work

There is an enormous amount of related work on Church encodings and in particular on attribute grammars, hence we will discuss only the most closely related works.

Object algebras were proposed by Oliveira and Cook (2012) and Oliveira et al. (2013) as a way to modularize algorithms on structured data. The first paper shows how to solve the expression problem with object algebras, while the second paper concentrates on a decomposition of algebras into "features". What we have called pre-algebras are called *generalized object algebras* by Oliveira et al. (2013). Compared to these works, the main technical novelty of our approach is the incorporation of inherited attributes, the shift of computation to traversal-time and associated possibility of one-pass compilation, and our algebra composition operator which encodes well-definedness into the type system (whereas the one used by Oliveira et al. (2013) can lead to circular dependencies and thus non-termination).

There are many works on compiling or embedding AGs (in)to various programming language paradigms (Paakki 1995, Sec. 3). The aim of first-class attribute grammars (De Moor et al. 2000), Kiama (Sloane et al. 2013), and AspectAG (Viera et al. 2009) is quite similar to ours, namely to achieve a compositional encoding of AGs into a programming language, including composition operators for attributes. Technically, these works are rather different, since they all operate on physical trees in memory. Achieving a similar kind of modularity as in our encoding would require a solution to the expression problem for algorithms on physical trees for the respective languages. In AspectAG, this is achieved by quite sophisticated type-level programming. Dependencies in these works are not as explicitly represented as in our approach; for instance, circular equations are not rejected by the type checker.

In general, to the best of our knowledge this is the first approach to *Church-encode* attribute grammars in a functional language. Due to the careful generalization of the object algebra technique, the well-known modularity and extensibility benefits of object algebras carry over to our AG encoding.

10. Conclusion

Based on previous work on object algebras, we successfully Church-encode L-attributed grammars. We formalize the encoding and gain initial experimental evidence that the encoding is modular, scalable and compositional as well as potentially useful in practical applications. In future work, we want to extend our technique to support more forms of attributes and to apply it to the design of extensible compilers.

Acknowledgments

We would like to thank Paolo G. Giarrusso for comments and discussions about earlier drafts of this paper and the anonymous reviewers for their helpful comments. This work is supported by the European Research Council, grant #203099 "ScalPL".

References

G. V. Bochmann. Semantic evaluation from left to right. *Communications of the ACM*, 19(2):55–62, 1976.

P. Buchlovsky and H. Thielecke. A type-theoretic reconstruction of the visitor pattern. In *Proceedings of the Conference on Mathematical Foundations of Programming Semantics*, pages 309–329. Elsevier ENTCS 155, 2006.

J. Carette, O. Kiselyov, and C. Shan. Finally tagless, partially evaluated. In *Proceedings of the Asian Symposium on Programming Languages and Systems*, pages 222–238. Springer LNCS 4807, 2007.

J. Carette, O. Kiselyov, and C. Shan. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *Journal of Functional Programming*, 19(5):509–543, Sept. 2009.

L. M. Chirica and D. F. Martin. An order-algebraic definition of Knuthian semantics. *Mathematical Systems Theory*, 13:1–27, 1979.

O. De Moor, K. Backhouse, and S. D. Swierstra. First-class attribute grammars. *Informatica*, 24, 2000.

P. Deransart, M. Jourdan, and B. Lorho. *Attribute Grammars: Definitions, Systems, and Bibliography*. Springer LNCS 323, 1988.

É. Duris, D. Parigot, G. Roussel, and M. Jourdan. Attribute grammars and folds: Generic control operators. Rapport de recherche RR-2957, INRIA, 1996.

J. Gibbons. Design patterns as higher-order datatype-generic programs. In *Proceedings of the Workshop on Generic Programming*, pages 1–12. ACM, 2006.

G. Hedin. Reference attributed grammars. *Informatica (Slovenia)*, 24(3):301–317, 2000.

G. Hedin. An introductory tutorial on JastAdd attribute grammars. In *Proceedings of the Summer School Conference on Generative and Transformational Techniques in Software Engineering*, pages 166–200. Springer LNCS 6491, 2011.

R. Hinze. Generics for the masses. In *Proceedings of the International Conference on Functional Programming*, pages 236–243. ACM, 2004.

R. Hinze. Generics for the masses. *Journal of Functional Programming*, 16(4-5):451–483, July 2006.

C. Hofer, K. Ostermann, T. Rendel, and A. Moors. Polymorphic embedding of DSLs. In *Proceedings of the Conference on Generative Programming and Component Engineering*. ACM, 2008.

T. Johnsson. Attribute grammars as a functional programming paradigm. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, pages 154–173, 1987.

P. M. Lewis, D. J. Rosenkrantz, and R. E. Stearns. Attributed translations. *Journal of Computer and System Sciences*, 9(3): 279–307, Dec. 1974.

E. Magnusson and G. Hedin. Circular reference attributed grammars: Their evaluation and applications. *Science of Computer Programming*, 68(1):21–37, Aug. 2007.

A. Middelkoop, A. Dijkstra, and S. D. Swierstra. Visitor-based attribute grammars with side effect. In *Proceedings of the Workshop on Generative Technologies*, pages 47 – 69. Elsevier ENTCS 264(5), 2011.

M. Odersky and M. Zenger. Independently extensible solutions to the expression problem. In *Proceedings of the Workshop on Foundations of Object-Oriented Languages*, 2005.

B. C. Oliveira, M. Wang, and J. Gibbons. The visitor pattern as a reusable, generic, type-safe component. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 439–456. ACM, 2008.

B. C. d. S. Oliveira and W. R. Cook. Extensibility for the masses: Practical extensibility with object algebras. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 2–27. Springer LNCS 7313, 2012.

B. C. d. S. Oliveira and J. Gibbons. Typecase: A design pattern for type-indexed functions. In *Proceedings of the Haskell Workshop*. ACM, 2005. ISBN 1-59593-071-X.

B. C. d. S. Oliveira, T. van der Storm, A. Loh, and W. R. Cook. Feature-oriented programming with object algebras. In *Proceedings of the European Conference on Object-Oriented Programming*. Springer LNCS 7920, 2013.

J. Paakki. Attribute grammar paradigms: A high-level methodology in language implementation. *ACM Computing Surveys*, 27(2): 196–255, June 1995.

A. M. Sloane, L. C. Kats, and E. Visser. A pure embedding of attribute grammars. *Science of Computer Programming*, 78(10): 1752–1769, 2013.

S. D. Swierstra, P. R. Azero Alcocer, and J. Saraiva. Designing and implementing combinator languages. In *Advanced Functional Programming*, pages 150–206, 1998.

E. Van Wyk, O. De Moor, K. Backhouse, and P. Kwiatkowski. Forwarding in attribute grammars for modular language design. In *Proceedings of the Conference on Compiler Construction*, pages 128–142. Springer LNCS 2304, 2002.

E. Van Wyk, L. Krishnan, A. Schwerdfeger, and D. Bodin. Attribute grammar-based language extensions for Java. In *Proceedings of the European Conference on Object-Oriented Programming*. Springer LNCS 4609, 2007.

M. Viera, S. D. Swierstra, and W. Swierstra. Attribute grammars fly first-class: How to do aspect oriented programming in Haskell. In *Proceedings of the International Conference on Functional Programming*, pages 245–256. ACM, 2009.

H. H. Vogt, S. D. Swierstra, and M. F. Kuiper. Higher order attribute grammars. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 131–145. ACM, 1989.

P. Wadler. The expression problem. Note to Java Genericity mailing list, Nov. 1998.