

# Aspect-Oriented Compilers

Oege de Moor<sup>1</sup>, Simon Peyton–Jones<sup>2</sup>, and Eric Van Wyk<sup>1</sup>

<sup>1</sup> Oxford University Computing Laboratory

<sup>2</sup> Microsoft Research, Cambridge

**Abstract.** Aspect-oriented programming provides the programmer with means to *cross-cut* conventional program structures, in particular the class hierarchies of object-oriented programming. This paper studies the use of aspect orientation in structuring syntax directed compilers implemented as attribute grammars. Specifically, it describes a method for specifying definitions of related attributes as ‘aspects’ and treating them as first-class objects, that can be stored, manipulated and combined. It is hoped that this embedding of an aspect-oriented programming style in Haskell provides a stepping stone towards a more general study of the semantics of aspect-oriented programming.

## 1 Introduction

Compilers are often structured by recursion over the abstract syntax of the source language. For each production in the abstract syntax, one defines a function that specifies how a construct is to be translated. The method of structuring compilers in this syntax–directed manner underlies the formalism of *attribute grammars* [2, 14, 18]. These provide a convenient notation for specifying the functions that deal with each of the production rules in the abstract syntax. The compiler writer need not concern himself with partitioning the compiler into a number of passes: the order of computation is derived automatically. One way of achieving that ordering is to compute the attribute values in a demand-driven fashion. Indeed, attribute grammars can be viewed as a particular style of writing lazy functional programs [11, 19].

Unfortunately, however, compilers written as attribute grammars suffer from a lack of modularity [16]. In their pure form, the only way in which attribute grammars are decomposed is by *production*. It is not possible to separate out a single semantic *aspect* (such as the ‘environment’) across all productions, and then add that as a separate entity to the code already written. The compiler writer is thus forced to consider all semantic aspects simultaneously, without the option of separating his concerns. Many specialised attribute grammar systems offer decomposition by aspect, but only at a *syntactic* level, not at a *semantic* one. In particular, aspects cannot be parameterised, and the compiler writer cannot define new ways of combining old aspects into new. For the purpose of this paper, let us define an *aspect* as a set of definitions of one or more related attributes. This paper proposes an implementation of aspects that makes

them independent semantic units, that can be parameterised, manipulated and compiled independently.

Our proposed compiler aspects are implemented in a variant of the programming language Haskell, augmented with extensible records. It is this highly flexible type system which allows us to give a type to each aspect. In particular, it ensures that each attribute is defined precisely once — an important feature when attribute grammars are composed from multiple components. It is assumed that the reader is familiar with programming in Haskell [5]. The concepts are not dependent on the Haskell language and they could be presented in a abstract, language independent form. However, we hope that an concrete implementation (available on the web [6]) will encourage others to explore aspect-oriented compilers. The L<sup>A</sup>T<sub>E</sub>X source of this paper is itself an executable Haskell program. The lines preceded by the > symbol are the Haskell program that is this paper. Note however, that some unenlightening portions of the program code appear in L<sup>A</sup>T<sub>E</sub>X comments and are thus not visible in the printed version.

## 2 A polymorphic type system for extensible records

We shall use the *Trex* extension of Haskell, which provides a rich set of record operations [9]. In this variant of Haskell, a record with three fields called *x*, *y*, and *z* may be written (*x*=0, *y*='a', *z*="abc"). The type of this expression is `Rec(x :: Int, y :: Char, z :: String)`. For each field name, there is a *selection function*, named by prefixing with a #. We thus have, for example, that `#y (x=0, y='a', z="abc")` evaluates to 'a'. Records can be extended with new fields. The function

```
f r = (z = "abc" | r)
```

adds a new field named *z* to its argument. The type of *f* reflects that this function should not be applied to a record that already has a field named *z*:

```
f :: r\z => Rec r -> Rec (z :: String | r)
```

That is, for each *row* *r* of fields that *lacks* *z*, *f* maps a record with fields *r* to a record that has one more field, namely *z*, whose value is a string.

Since a record can be extended, it is natural to consider a starting point for such extensions, namely the empty record, which is written `EmptyRec`, and whose type is `Rec EmptyRow`.

## 3 Motivating example: Algol 60 scope rules

In contrast to a good many of its successors, Algol 60 has very clear and uniform scope rules. A simplification of these scope rules is a favourite example to illustrate the use of attribute grammars [16]. A definition of an identifier *x* is visible in the smallest enclosing block with the exception of inner blocks that also contain a definition of *x*. Here we shall study these scope rules via a toy language that has the following example program and abstract syntax:

```

>example = [Use "x", Use "y",
>          Local [Dec "y", Use "y", Use "x"],
>          Dec "x", Use "x", Dec "y"]

>type Prog = Block
>type Block = [Stat]
>data Stat = Use String | Dec String | Local Block

```

We aim to translate programs to a sequence of instructions for a typical stack machine. The type of instructions is

```

>data Instr = Enter Int Int | Exit Int | Ref (Int,Int)

```

Each block entry and exit is marked with its lexical level. Each entry is also marked by the number of local variables declared in that block. Each applied occurrence of an identifier is mapped to a (level, displacement) pair, consisting of the lexical level where the identifier was declared, and the displacement, which is the number of declarations preceding it at that level. To wit, we wish to program a function `trans :: Prog -> [Instr]` so that, for instance, we have

```

trans example = [Enter 0 2, Ref (0,0), Ref (0,1),
                Enter 1 1, Ref (1,0), Ref (0,0), Exit 1,
                Ref (0,0), Exit 0]

```

## 4 A traditional compiler

We now proceed to write a program for `trans`, in the traditional attribute grammar style, especially as suggested in [4, 11, 19, 26, 28]. This means that we will not be concerned with slicing the computations into a minimal number of passes over the abstract syntax; such a division into passes comes for free by virtue of lazy evaluation. While this section only reviews existing techniques for writing attribute grammars, we write `trans` using the extensible record notation to set the stage for Section 5, where extensible records are a key component of our new modular approach to defining attribute grammars.

First we provide the context-free grammar for the source language:

```

Program: Prog -> Block      List: Block -> SList      Use: Stat -> String
SList0:  SList ->          Local: Stat -> Block      Dec: Stat -> String
SList1:  SList -> Stat SList

```

This context-free grammar is close to the type definitions we stated earlier. Roughly speaking, types correspond to nonterminals, and constructors correspond to production rules. Note, however, that we have explicitly written out productions for statement lists, although these productions are not explicit in the type definitions.

The standard strategy for writing an attribute grammar consists of three steps, namely the definition of *semantic domains*, *semantic functions*, and *translators*.

## 4.1 Semantic domains

For each nonterminal symbol  $S$  we define a corresponding *semantic domain*  $S'$ . The compiler will map values of type  $S$  to values of type  $S'$ . These types will likely include the generated code, as in `type Prog' = Rec (code :: [Instr])`, and will be defined via record types where the fields represent various aspects of the semantics. For other grammar symbols, however, a mere record type will not suffice, because their semantics depends on the context in which they occur. That motivates semantic domains that are functions between record types: the input record describes attributes of the context – these are called *inherited* attributes, and the output record describes resulting attributes of the grammar symbol itself – these are called *synthesized* attributes. For example, we have

```
>type SList' = Rec (level :: Int, env :: Envir) ->
>               Rec (code :: [Instr], locs :: [String])
```

That is, given the lexical level and environment (which maps identifiers to (level,displacement) pairs), a statement list will yield code, which is a list of instructions and a list of local variable names, called locs.

It remains to define a semantic domain for blocks and for statements themselves, which happens to be the same as for statement lists:

```
>type Block' = Rec (level::Int, env::Envir) -> Rec (code::[Instr])
>type Stat' = SList'
```

## 4.2 Semantic functions

Before we can define the semantic functions that make up the compiler, we first need some primitive operations for manipulating environments. An environment is an association list from identifiers to (level,displacement) pairs, and we shall write `Envir` for the type of environments. The two operations `apply` and `add` are defined on environments and have the types `apply :: Envir -> String -> (Int,Int)` and `add :: Int -> [String] -> Envir -> Envir`. The function `apply e x` finds the first occurrence of `x` in `e`, and returns the corresponding (level,displacement) pair. We shall build up the environment by adding all local definitions at a given lexical level. This is the purpose of the function `add`: it takes a level, a list of local definitions, and an environment, and it adds the local definitions to the environment.

We are now in a position to define the semantic functions. For each production  $P: X \rightarrow Y Z$ , we define a *semantic function*  $p: Y' \rightarrow Z' \rightarrow X'$  that combines semantic values of the appropriate type. For example, we define the binary semantic function `slist1 :: Stat' -> SList' -> SList'` that takes the translations of a statement and a statement list, and produces the translation of the composite statement list. The two arguments, and the result appear in reverse order, when compared to the production `SList1`. The type of `list` is also obtained by reversing sides of the corresponding production rule to yield

```

>list :: SList' -> Block'
>list slist blockIn
> = (code = [Enter (#level blockIn) (length (#locs slistOut))]
>         ++ #code slistOut ++ [Exit (#level blockIn)])
>   where slistIn = (level = #level blockIn,
>                   env = add (#level blockIn) (#locs slistOut)
>                           (#env blockIn))
>   slistOut = slist slistIn

```

It is worthwhile to note the seeming circularity in the argument and result of `slist`. Such definitions are only acceptable because of lazy evaluation. If we programmed the same computation in a strict language, we would have to remove such pseudo-circularities by introducing multiple passes over the abstract syntax.

The above definition of `list` illustrates how in the traditional approach to writing attribute grammars, different aspects must all be defined in a single location. The aspects cannot be split apart, forcing the compiler writer to consider all the semantic aspects simultaneously. It is this deficiency that we aim to remedy below.

The definitions of the other semantic functions are similar and we omit details. To avoid confusion, we mention that our notion of ‘semantic function’ is different from that in the attribute grammar literature. There, a semantic function is understood to be the right-hand side of the definition of a single attribute, and what we call a semantic function is simply termed a ‘production’.

### 4.3 Translators

For each nonterminal  $S$ , we define a *translator* of type `transS :: S -> S'` that maps values of type  $S$  to the corresponding semantic domain  $S'$ . For example, the function that translates programs has type `transProg :: Prog -> Prog'` and the translator for statement lists has type `transSList :: SList -> SList'`. Assuming the existence of a semantic function for each production, we can define a translator for each type in the abstract syntax by:

```

>transProg p          = program (transBlock p)
>transBlock b        = list (transSList b)
>transSList []       = slist0
>transSList (s:ss)   = slist1 (transStat s) (transSList ss)
>transStat (Use x)   = use x
>transStat (Dec x)   = dec x
>transStat (Local b) = local (transBlock b)

```

## 5 An aspect-oriented compiler

We now aim to embed the attribute definition language as a combinator library into Haskell. To some extent, we already did that in the previous section, but

to obtain a truly modular design, we propose making nonterminals, attribute definition rules, semantic functions and aspects first-class objects. We then use polymorphic operations on extensible records to give types to these objects and the combining forms for these objects. As we shall see below, the trickiest problem is to find an appropriate type of attribute definition rules.

As in the traditional approach above, we define a translator `trans'` with type `trans' :: Prog -> [Instr]` so that `trans' example` evaluates to the same result as `trans example` above. As in Section 4.3, `trans'` is defined using a collection of translators, one for each production in the abstract syntax. The semantic functions used in these translators are not the named semantic functions `program`, `list`, etc. used in the traditional approach, but are extracted from the fields of an attribute grammar named `ag ()`. For example, the translator function `transProg'` for production `program` is defined as `transProg' p = #program (ag ()) (transBlock' p)`. Thus, `ag ()` is a record with a field for each abstract syntax production which contains its semantic function. The fields of this record have the same names and types used for the semantic functions in the traditional approach. The important distinction is that the semantic functions in `ag ()` are built using an aspect-oriented approach. That is, they are constructed by grouping attribute definitions by aspect instead of by production. (The dummy argument `()` to `ag` and other constructs is required because of a technicality in Haskell's type system, known as the monomorphism restriction.)

## 5.1 Combining aspects

In our example, the aspects are named `levels`, `envs`, `locss` and `codes` and define, respectively, the attributes lexical level, environment, local variables, and target code. Given these aspects, we combine them into an attribute grammar `ag ()` in the following way:

```
>ag () = knit (levels() 'cat' envs() 'cat' locss() 'cat' codes())
```

Here, `knit` and `cat` are functions for combining aspects into attribute grammars and are defined below. Given this framework, it is clear that we can write new aspects and add them into our attribute grammar using these combinators.

## 5.2 Aspect definitions

The semantic function of a production `P` must define each of the *synthesised* attributes of the parent of `P`, and each of the *inherited* attributes of `P`'s children (Section 4.2). Together we refer to these attributes as `P`'s *output* attributes. To produce the output attributes, the semantic function takes as arguments all of `P`'s *input* attributes, that is the synthesised attributes of `P`'s children, and the inherited attributes of `P`'s parent. The trouble with the traditional approach is that we are forced to define all `P`'s output attributes simultaneously — just look at the definition of `list` in Section 4.2. Our new, modular approach is to express

each semantic function as a composition of one or more rules. Each rule for a production P defines a subset of P's output attributes and is implemented as a function which takes the input attributes from the parent and children of P.

Given a context-free grammar, a *rule grammar* is a record whose fields consist of rules, one for each production. Because of the monomorphism restriction mentioned above, we define an *aspect* as a function taking the dummy argument () and returning a rule grammar. Many aspects involve only a tiny subset of the productions. Think, for example, of operator priorities: these only affect productions for expressions. A rule grammar involves all productions, by definition. Therefore, definitions of aspects are written so that only the rules being defined by an aspect are explicitly written and default rules are provided for the rest.

Our first concrete example of an aspect is *lexical level*. The `level` attribute is inherited, and it is explicitly defined in two productions, namely `program` and `local`:

```
>levels ()=(program=(\b p -> ((level = 0 | #i b), #s p)),
>          local=(\b p -> ((level = #level (#i p) +1 | (nolevel(#i b))),
>                          #s p))          | grammar)
>          where nolevel (level=_ | r) = r
>          (program=_ , local=_ | grammar) = none ()
```

As we will see, the default behaviour for rules for inherited attributes is to copy the parent's attribute value to the children. Thus, we don't write explicit rules for the other productions. This is accomplished by the phrase `(program=_, local=_ | grammar) = none()` which first fills the fields in `grammar` (all those except `program` and `local`) with the default copy rules pulled from the identity rule `grammar`, named `none()`. These defaults are added to the definitions of rules for `program` and `local` to create a complete rule grammar. The given rules are written using lambda expressions (the `\` above can be read as  $\lambda$ ); these functions take the input attributes, held in the child argument `b` and parent argument `p`, and return a tuple which adds the attribute `level` to the inherited attributes of the child `b`, and adds no new synthesized attributes to the parent `p`. These parameters pair up the inherited and synthesized attributes of each symbol in a type of nonterminal: `>type NT ai as = Rec (i :: Rec ai, s :: Rec as)`. Note that both arguments to this type definition, `ai` and `as`, are row variables. The fields in these rows are the attributes themselves.

The record generated by a rule keeps track of the attribute definitions made so far; above, we are adding the `level` attribute definition to the attribute definitions already made to the block `b`. In the definition of the rule for `local` we override the default definition of `level`, by removing it with the function `nolevel`, and then adding a fresh `level` field. Apart from the fact that the above definition of `levels` is re-usable, we also find it easier to read: the flow of the level computation over the abstract syntax tree is clear at a glance, especially because the default copy rules allow us to leave out all irrelevant detail.

The *local variables* aspect of the compiler records the local variables declared at each lexical level. The `locs` attribute is synthesised, and it is adjoined to

five rules. Because `locs` is synthesised, we cannot rely on default copy rules, so this aspect is somewhat more complex than the previous one, which dealt with inherited attributes.

```
>locss ()=(slist0 = (\p      ->(locs= [] | #s p)),
>      slist1 = (\a as p ->(#i a, #i as,
>          (locs= #locs (#s a) 'union' #locs (#s as) | #s p))),
>      use    = \a -> (\p->(locs= [] | #s p)),
>      dec    = \a -> (\p->(locs= [a] | #s p)),
>      local  = (\b p ->(#i b, (locs= [] | #s p))) | grammar)
>      where (slist0=_,slist1=_,use=_,dec=_,local=_|grammar) = none()
```

Here we see that `use` and `dec` are special: they are functions that take a string and yield a rule of arity 0. This is the usual way of dealing with grammar symbols (such as identifiers) that fall outside an attribute grammar.

### 5.3 Attribute definition rules

We now show the development of the rules used to compose semantic functions. The type of a rule has been alluded to above as a function which maps a subset of a productions input attributes to a subset of its output attributes. In this section we provide a precise definition of rules. We build a semantic function by composing rules. When we compose rules, the type system will ensure that no attribute is defined twice; when we assert that a composition of rules defines a complete semantic function, the type system will ensure that every attribute that is used is also defined.

For example, we will be able to construct the `list` semantic function of Section 4.2 thus:

```
list=knit1(list_level 'cat1' list_env 'cat1' list_locs 'cat1' list_code)
```

Here `list_level` etc. are rules, `cat1` composes rules, and `knit1` transforms a composed rule into a semantic function. The “1” suffixes refer to the fact that the `List` production has just one child; we have to define variants of `knit` and `cat` for productions with a different number of children.

We seek a mechanism of composing rules into semantic functions which allows the use of default rules, since we have seen that they shorten and clarify aspect definitions. Also, there is no record concatenation operator in type systems for polymorphic extensible records. For these two reasons, we do not concatenate the records generated by rules, but rather use a solution suggested by Rémy [23, 24] to compose the functions that build up those records.

To apply Rémy’s technique in the particular example of attribution rules, a rule also takes as input the existing output attributes which are passed to the rule in the nonterminal symbols. A rule does not return fixed records of defined attributes; instead it transforms existing definitions by adding new fields for new attribute definitions or by replacing field values with new values. The latter is done to overwrite the default rules. For example, consider the rule `list_level`:

```
list_level = \c p -> ( (level = level p | (nolevel (#i c))), #s p)
```

This rule overwrites the definition of `level` of the inherited attributes of the child and leave the synthesised attributes of the parent unchanged. The type of such a unary rule is

```
>type Rule1 child parent childInh parentSyn
> = child -> parent -> (Rec childInh, Rec parentSyn)
```

Here `child` and `parent` are understood to be nonterminals, whereas `childInh` and `parentSyn` are row variables. The type of `list_level` is thus

```
list_level :: (childInh\level,parentInh\level) =>
             Rule1 (NT (level :: Int | childInh) childSyn)
                  (NT (level :: Int | parentInh) parentSyn)
                  (level :: Int | childInh) parentSyn
```

With this definition of rules, it is straightforward to define the concatenation operator as suggested by Rémy's work:

```
>cat1 :: Rule1 (NT ci cs) (NT pi ps) ci' ps' ->
>       Rule1 (NT ci' cs) (NT pi ps') ci'' ps'' ->
>       Rule1 (NT ci cs) (NT pi ps) ci'' ps''
>cat1 f g c p = g (i=ci', s=#s c) (i=#i p, s=ps') where (ci',ps')=f c p
```

This definition encodes the sequential composition of rule `f` followed by `g`. The lifted version of this combinator, named `cat` takes two rule grammars (records with a rule for each production), and it concatenates their corresponding fields. The concatenation operator does of course have an identity element, namely the rule that leaves all attribute definitions unchanged:

```
>none1 :: Rule1 (NT ci cs) (NT pi ps) ci ps
>none1 c p = (#i c, #s p)
```

The lifted version, `none()`, is similar to `cat` and is a record with fields for each production which contains the identity rule of the appropriate arity.

#### 5.4 Semantic functions

Once we have defined all the requisite attribute values by composing rules, we can turn the composite rule into a *semantic function*. In effect, this conversion involves connecting attribute definitions to attribute applications. We shall call the conversion *knitting*. The type of semantic functions of arity 1 is

```
>type Semfun1 childInh childSyn parentInh parentSyn
> = (Rec childInh -> Rec childSyn) -> (Rec parentInh -> Rec parentSyn)
```

The operation `knit1` takes a rule, and it yields a semantic function. The function that results from `knit1` takes the semantics of child `c` (which is of type `Rec ci -> Rec cs`) as well as the inherited attributes for parent `p` (a value of type `Rec pi`). It has to produce the synthesised attributes of `p`. This is achieved by applying the rule, which builds up the synthesised attributes of `p` starting from the empty record and builds up the inherited attributes of `c` starting from the inherited attributes of `p`. This implies that inherited attributes of `p` are copied to `c`, unless otherwise specified. There is no such default behaviour, however, for synthesised attributes.

```
>knit1 :: Rule1 (NT pi cs) (NT pi EmptyRow) ci ps -> Semfun1 ci cs pi ps
>knit1 rule c pi = ps   where (ci,ps) = rule (i=pi,s=cs)(i=pi,s=EmptyRec)
>                          cs         = c ci
```

The type of `knit1` shows that the rule is required to yield the synthesised attributes `ps`, starting from the empty record. Furthermore, the rule must transform the inherited attributes `pi` into the inherited attributes `ci` of the child. It also shows that the child's and parent's inherited attributes given as input to the rule have the same type, namely `pi`. The definition shows that the parent's inherited attributes, `pi`, are given as the default values of the child's inherited attributes in the application of `rule`. Thus, if a rule does not redefine a child's inherited attributes, the default behavior is to copy them from the parent. When the resulting semantic function is applied to the semantics of the child `c`, and to the inherited attributes of the parent, `pi`, it returns the synthesised attributes `ps`. The inherited attributes `ci` of the child, and the synthesised attributes `ps` are the joint result of applying `rule`.

This completes the set of basic combinators for manipulating rules and semantic functions of arity 1. There are similar combinators for other arities, and we omit the details.

## 6 Discussion

### 6.1 Aspect-oriented programming (AOP)

The inability to separate aspects is not exclusive to the area of compiler writing, and it has received considerable attention in other areas of programming. Gregor Kiczales and his team at Xerox have initiated the study of *aspect-oriented programming* in general terms [17], and the notion of *adaptive object-oriented programming* of Karl Lieberherr *et al.* shares many of these goals [21]. Don Batory and his team at UTA have studied ways to describe aspects in software generators that cut across traditional object class boundaries [3]. The present paper is a modest contribution to these developments, by showing how compilers can be structured in an aspect-oriented style. We are hopeful that the techniques we have employed here can be applied to writing aspect-oriented programs in other problem domains as well.

It is worthwhile to point out some deviations from Kiczales’ original notion of AOP. The notion of aspect in this paper is highly restrictive, and only covers those examples where the “weaving” of aspects into existing code is purely name-based, and not dependent on sophisticated program analyses. For example, in Kiczales’ framework, one might have an aspect that maintains an invariant relationship between variables  $x$  and  $y$ . Whenever either of these is updated, the invariant must be restored by making an appropriate change to the other variable. To weave the aspect into existing code, we have to find places where either  $x$  or  $y$  is changed: the techniques in this paper have nothing to say about such sophisticated aspects. In fact, to avoid all forms of program analysis, we require that the original attribute grammar is written as a rule grammar, and not in its knitted form.

Another seeming difference is one of style. In AOP, the traditional method of composing programs is not replaced, but is complemented by the introduction of aspects. The example we used in this paper is misleading, because we took an extreme approach, and sliced up the original attribute grammar completely in terms of aspects, thus abandoning the primary composition method. That was done purely for expository purposes, and there is no reason why one could not write a rule grammar in the traditional style, and then add one or two aspects later. Indeed, that is likely to be the norm when writing larger attribute grammars. Therefore, we do not suggest that the ‘production’ method of composition be replaced by the aspect, but simply augmented by it. Aspects are a useful tool for creating attribute grammars that in many instances is superior to composition by production.

In summary, we expect that the techniques of this paper are relevant to other applications of aspect-oriented programming, in other problem domains, but only those where the weaving is purely name-based. Because our definitions are in a simple functional programming language, one could also view our contribution as a first step towards a semantics of aspects.

## 6.2 Attribute grammar systems

An obvious objection to the work presented here is that many attribute grammar based compiler generators offer the factorisation we seek, but at a purely syntactic level [7, 25, 28]. The programmer can present attribution rules in any order, and a preprocessor rearranges them to group the rules by production. The situation is akin to the dichotomy between macros and procedures: while many applications of procedures can be coded using macros, the concept of a procedure is still useful and important. In contrast to macros, procedures offer sound type checking, and they are independent entities that can be stored, compiled and manipulated by a program. The benefits deriving from having aspects as explicit, first-class entities in a programming language are the same. Adams [1] proposes a similar decomposition method, but lacks the type checking of aspects possible here.

Ganzinger and Giegerich [8] also modularize attribute grammars but by decomposing the translation process into phases. This is a coarser decomposition

than our proposed method, which could be used to decompose the specifications of their phases by aspect instead of by production.

The type system guarantees that all attributes are defined, and that they are defined only once. These guarantees are of course also ensured in specialised attribute grammar systems. Such systems usually also test for cycles in attribute definitions [12, 18]. In moving from a dedicated attribute definition language to a general programming language, this analysis is a feature one has to give up. Cycle checks are only an approximation, however, so they inevitably rule out attribute grammars that can be evaluated without problems.

Although we are still investigating if additional advanced features found in other systems [10, 13, 15, 25, 20] can be mimicked in our setting, a companion [6] to this paper presents a substantial case study that shows how our technique admits concepts such as *local attributes* and *higher-order attribute grammars* in a natural manner. An attribute grammar for a production language's complete semantics has not been completed; however, we are confident (but not yet certain) that this method will scale to handle these larger attribute grammars.

The ultimate aim of this work is to provide a suitable meta-language for rapid prototyping of domain-specific languages in the *Intentional Programming* system under development at Microsoft Research [27] which, although different from attribute grammar systems, faces many of the same issues of modularity, evaluation schemes, and specification language choice.

## References

1. S. R. Adams. *Modular Grammars for Programming Language Prototyping*. PhD thesis, Department of Electronics and Computer Science, University of Southampton, UK, 1993.
2. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.
3. D. Batory, V. Singhal, J. Thomas, S. Dasari, B. Geraci, and M. Sirkin. The GenVoca model of software-system generators. *IEEE Software*, 11(5):89–94, 1994.
4. R. S. Bird. A formal development of an efficient supercombinator compiler. *Science of Computer Programming*, 8(2):113–137, 1987.
5. R. S. Bird. *Introduction to Functional Programming in Haskell*. International Series in Computer Science. Prentice Hall, 1998.
6. O. De Moor. First-class attribute grammars. 1999. Draft paper available from URL <http://www.comlab.ox.ac.uk/oucl/users/oege.demoor/homepage.htm>
7. P. Deransart, M. Jourdan, and B. Lorho. *Attribute grammars — Definitions, systems and bibliography*, volume 322 of *LCNS*. Springer Verlag, 1988.
8. H. Ganzinger and R. Giegerich. Attribute Coupled Grammars. In *Proceedings of the ACM Symposium on Compiler Construction*, 157–170, 1984. Published as *ACM SIGPLAN Notices*, 19(6).
9. B. R. Gaster and M. P. Jones. A polymorphic type system for extensible records and variants. Technical report NOTTCS-TR-96-3, Department of Computer Science, University of Nottingham, UK, 1996. Available from URL <http://www.cs.nott.ac.uk/Department/Staff/mpj/polyrec.html>.
10. R. W. Gray, V. P. Heuring, S. P. Levi, A. M. Sloane, and W. M. Waite. Eli: A complete, flexible compiler construction system. *CACM*, 35:121–131, 1992.

11. T. Johnsson. Attribute grammars as a functional programming paradigm. In G. Kahn, editor, *Functional Programming Languages and Computer Architecture*, volume 274 of *LNCS*, pages 154–173. Springer-Verlag, 1987.
12. M. Jourdan. Strongly non-circular attribute grammars and their recursive evaluation. *SIGPLAN Notices*, 19:81–93, 1984.
13. M. Jourdan, D. Parigot, C. Julié, O. Durin, and C. Le Bellec. Design, implementation and evaluation of the FNC-2 attribute grammar system. In *Conference on Programming Languages Design and Implementation*, pages 209–222, 1990. Published as *ACM SIGPLAN Notices*, 25(6).
14. U. Kastens. Attribute grammars in a compiler construction environment. In *Proceedings of the International Summer School on Attribute Grammars, Applications and Systems*, volume 545 of *LNCS*, pages 380–400, 1991.
15. U. Kastens, B. Hutt, and E. Zimmermann. *GAG: A Practical Compiler Generator*, volume 141 of *LNCS*. Springer Verlag, 1982.
16. U. Kastens and W. M. Waite. Modularity and reusability in attribute grammars. *Acta Informatica*, 31:601–627, 1994.
17. G. Kiczales. Aspect-oriented programming. *ACM Computing Surveys*, 28A(4), 1996. See also: <http://www.parc.xerox.com/spl/projects/aop>.
18. D. E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2:127–146, 1968.
19. M. Kuiper and S. D. Swierstra. Using attribute grammars to derive efficient functional programs. In *Computing Science in the Netherlands CSN '87*, 1987. See: <ftp://ftp.cs.ruu.nl/pub/RUU/CS/techreps/CS-1986/1986-16.ps.gz>.
20. M. Kuiper and J. Saraiva. LRC — A Generator for Incremental Language-Oriented Tools. In K. Koskimies, editor, *7th International Conference on Compiler Construction*, pages 298–301. volume 1383 of *LNCS*. Springer Verlag, 1998.
21. K. J. Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*. PWS Publishing Company, 1996.
22. A. Ohori. A polymorphic record calculus and its compilation. *ACM Transactions on Programming Languages and Systems*, 17(6):844–895, 1995.
23. D. Rémy. Typechecking records and variants in a natural extension of ML. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL '89)*, pages 77–88. ACM Press, 1989.
24. D. Rémy. Typing record concatenation for free. In C. A. Gunter and J. C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics and Language Design*, Foundations of Computing Series. MIT Press, 1994.
25. T. W. Reps and T. Teitelbaum. *The Synthesizer Generator: A system for constructing language-based editors*. Springer-Verlag, 1989.
26. D. Rushall. *An attribute evaluator in Haskell*. Technical report, Manchester University, 1992. See URL: <http://www-rocq.inria.fr/oscar/www/fnc2/AG.html>.
27. C. Simonyi. Intentional programming: Innovation in the legacy age. Presented at IFIP Working group 2.1, 1996. Available from URL <http://www.research.microsoft.com/research/ip/>.
28. S.D. Swierstra, P. Azero and J. Saraiva. Designing and implementing combinator languages. In S. D. Swierstra, editor, *3rd International Summer School on Advanced Functional Programming*, volume 1608 of *LNCS*. Springer Verlag, 1999. See also URL <http://www.cs.uu.nl/groups/ST/Software/index.html>.
29. M. Wand. Type inference for record concatenation and multiple inheritance. *Information and Computation*, 93:1–15, 1991.