Aspect-Oriented Compilers

Oege de Moor
Oxford University Computing Laboratory
oege@comlab.ox.ac.uk

Simon Peyton–Jones
Microsoft Research Cambridge
simonpj@microsoft.com

Eric Van Wyk
Oxford University Computing Laboratory
evanwyk@comlab.ox.ac.uk

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.

Syntax-directed compilers are often specified by means of attribute grammars. Such specifications are typically structured by production — it is hard to structure them by semantic aspects such as 'environment', 'lexical level' and 'type checking'. Even if such structuring is allowed at a syntactic level, it is certainly not possible to parameterise compiler aspects, and to treat them as first-class objects in the specification language.

In this paper we propose a technique for making compiler 'aspects' first-class objects, that can be stored, manipulated and combined. We propose a modest set of combinators that achieve this task in the functional programming language Haskell. The combinator library is an application of recent work on polymorphic type systems for record operations, in particular that of Gaster and Jones, and also of a technique due to Rémy, which types symmetric record concatenation 'for free'. 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* [1, 15, 19]. 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 [12, 20].

Unfortunately, however, compilers written as attribute grammars suffer from a lack of modularity [17]. 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.

Figure 1 highlights the difference between the traditional 'syntactic view' of provided by grouping attribute definitions by production with the 'semantic view' provided by grouping attribute definitions by aspect. This image is a compressed view of the example attribute grammar code used in this paper written in the two styles with the definitions of the 'code' attribute shaded in gray. On the left is the traditional attribute grammar program in which the definitions of the 'code' attribute is distributed across the translation functions associated with each production. On the right is the attribute grammar written as an aspect oriented program. Here, the definitions of the 'code' attributes have been gather into a single aspect. Such a semantic view of an attribute grammar is helpful when additional 'semantic layers' are to be added without affecting existing aspects.

The implementation of compiler aspects that we propose here is illustrated in a variant of the programming language

Figure 1: Syntactic and Semantic views

Haskell, augmented with extensible records. It is this highly flexible type system which allows us to give a type to each aspect. In particular, its use 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]. In fact, the LaTeX 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 LaTeX comments and are thus not visible in the paper.

2 A polymorphic type system for extensible records

In the course of this paper, we shall make extensive use of record operations, and their associated types. Here we introduce the notation that we shall use, which is known as the *Trex* extension of Haskell.

A record maps field names to values. In essence, it is merely an association list, but by using a carefully crafted type discipline, one can avoid run-time errors of the kind "field not present" or "field multiply defined". The type discipline sketched below is due to Mark Jones and Benedict Gaster [8]. We chose this type discipline rather than any of the other proposals in the literature (*e.g.* [26, 28, 27, 34]) because an implementation is readily available.

In the Trex extension 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

```
Num a => Rec (x :: a, y :: Char, z :: String)
```

The phrase `Num a =>` is a type class restriction on the type variable `a` and states that `a` must be one of the Haskell numeric types. The order of fields in a record does not matter, but no field should appear more than once. For each field name, there is a *selection function*, named by prefixing with a `#`. We thus have, for example,

```
#y  (x = 0,  y = 'a',  z = "abc")    =   'a'
```

This is the minimum set of record operations, found in any language that supports records.

Extensible records are needed when we wish to dynamically add a new field to an existing record. For example, the following function `f` extends its argument `r` with a field named `z`:

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

The intention is that, for instance,

```
f (x = 0, y = 'a')  =
          (x = 0, y = 'a', z = "abc")
```

But what is the type of `f`? As said before, no field should appear twice in the same record, so `f` should not be applied to records `r` that already have a `z` field. The type of `f` ought to reflect the absence of `z` in `r`. We write `r\z` to signify a row of fields `r` that does not include `z`. In that notation, the type of `f` reads

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

It is important to realise that `r` is a new kind of type variable, which stands for a *row* of (field label, type) pairs. A row becomes an ordinary type by applying the `Rec` constructor to it. The above type of `f` should therefore be read as follows: for each row `r` that *lacks* `z`, `f` takes a record with fields described by `r` to a record that has one more field, namely `z`, whose value is a string.

Since records can be extended, it is natural to consider a starting point for such extensions, namely the empty record, which is written (rather unattractively) `EmptyRec` in Haskell. Its type is `Rec EmptyRow`.

Pattern matching is an important feature of Haskell, and it also applies to extensible records. We can thus define our own selection functions as follows:

```
sely  ::  Rec (x :: a, y :: b, z:: c) -> b
sely (x=_, y=b, z=_)   =   b
```

or indeed more generally

```
sely'  ::  r\y  =>  Rec (y :: b | r) -> b
sely' (y=b | r)    =     b
```

This last function `sely'` is identical to the built-in selection function `#y`.

Later in this paper, we shall often have occasion to override existing field values. This is achieved by first removing the relevant field, and then extending the reduced record again. For example, the function `newx` overrides the existing field `x` in its argument record, and replaces it by the string `"abc"`:

```
newx :: s\x =>
        Rec (x :: a | s) ->
        Rec (x :: String | s)
newx r = (x="abc" | nox r)
         where nox (x=_ | s) = s
```

3 Algol 60 scope rules

We shall now introduce a motivating example for the remainder of the paper. 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 [17]. 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 abstract syntax:

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

That is, a program consists of a block, and a block is a list of statements. A statement can be one of three things: an applied occurrence of an identifier, a defining occurrence of an identifier, or a local block. An example of a program is

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

Note that `x` is used before it is declared, and that the inner block declares a second variable `y`. Consequently, the second applied occurrence of `y` refers to that inner declaration at lexical level 1, and not to the outer declaration at level 0.

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 is marked with its lexical level and the number of local variables declared in that block. Each block exit is marked with the lexical level only. Finally, 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]
```

At lexical level 0, we have declared two identifiers, namely `x` and `y`. Entry to the block at level 0 is followed by applied occurrences of both `x` and `y`. Then we have a block entry to level 1, and here only one identifier has been declared, namely `y`. That new declaration is referred to, followed by a reference to `x` (which was declared at level 0). Level 1 is exited. There is one more reference to `x` at level 0, and the program concludes by exiting level 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, 12, 20, 30, 32]. 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 need to be a bit more explicit about the context–free grammar for the source language:

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

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

Our strategy for writing a compiler consists of three steps, namely the definition of *semantic domains*, *semantic functions*, and *translators*:

- For each nonterminal symbol `S` we define a corresponding *semantic domain* `S'` (Section 4.1). 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])
  ```

  but for nonterminals other than `Prog` they will also include attributes such as the lexical level.

- For each production `P: X -> Y Z`, we define a *semantic function* `p: Y' -> Z' -> X'` that combines semantic values of appropriate type (Section 4.2). For example, we shall define a function

  ```
  program :: Block' -> Prog'
  ```

  associated with the `Program` production above. For binary productions consider the 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`.

- 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'` (Section 4.3). For example, the function that translates programs has the type

  ```
  transProg :: [Stat] -> Rec (code::[Instr])
  ```

  and the statement translation function lists has the type

  ```
  transSList :: SList -> SList'
  ```

Given the above three components, the definition of the compiler itself reads:

```
>trans = #code . transProg
```

Recall the type of `trans` is `trans :: Prog -> [Instr]`. It now remains to define a semantic domain for each nonterminal, a semantic function for each production, and a translator for each nonterminal.

4.1 Semantic domains

We shall describe each semantic domain via record types, where the fields represent various aspects of the semantics. As we saw above, the semantic domain of programs has only one such aspect, namely the generated code. 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, and the output record describes resulting attributes of the grammar symbol itself. For example, we have

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

That is, given the lexical level and environment (which maps identifiers to (level,displacement) pairs), a block will yield code, which is a list of instructions. Readers who are familiar with attribute grammars will recognise `level` and `env` as inherited attributes, whereas `code` is a synthesised attribute. Assigning attributes to nonterminals is the same activity as designing their corresponding semantic domains [12, 20].

Statement lists are similar to blocks, but here we also compute a list of the local variables that are declared: this aspect is called `locs`. The semantic domain of statement lists is therefore

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

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

```
>type Stat'  = SList'
```

4.2 Semantic functions

Before we can proceed to 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. There are two operations defined on environments: `apply` and `add`.

```
>apply :: Envir -> String -> (Int,Int)
>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, one for each production in the grammar. The semantic function `program` for the corresponding production `Program` defines the `code` aspect of programs, and also the `level` and `env` aspects of its descendant `Block`. Its type is (again note the reversal)

```
>program :: Block' -> Prog'
```

Now, as explained before, `Block'` is a function type, taking a record with `level` and `env` fields to a record with a single `code` field. These type considerations lead to the following definition:

```
>program block = (code = #code blockOut)
>                where blockIn = (level=0, env=[])
>                      blockOut = block blockIn
```

That is, the outermost block of a program has lexical level 0 and an empty environment. The code generated for the program is the code generated for the outermost block.

The type of `list` is again obtained by reversing sides of the corresponding production rule:

```
>list :: SList' -> Block'
```

Recall that the semantic domains of statement lists and blocks only differ in the presence of a `locs` field (of local variables) in statement lists. The local variables have to be added to the environment of the block. These considerations yield the program:

```
>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 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

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)
```

In more realistic examples the difference between the grammar and the syntax types may be greater, and in such cases, one could say that `trans` *parses* the program tree according to the grammar. In our example, we only need to parse statement lists.

4.4 Evaluation of the traditional approach

This section has presented the traditional style of writing compilers as attribute grammars expressed in a functional language. This style does however have much to commend it. Unlike the multi-pass compilers in strict programming languages, we do not need to concern ourselves with a division into a minimal number of passes, and the dependencies between such passes. Because the translation is completely syntax-directed, there is strong guidance on how to proceed, and the result is quite readable and neatly decomposed by production in the abstract syntax. However, its main deficiency lies in precisely that decomposition: the notion of lexical level makes perfect sense, independent of the particular translation problem considered here. Indeed, the environment is a well-defined notion and has nothing to do with the particular kind of code we generate. And yet all these aspects are irretrievably intertwined in the compiler. This was illustrated in the left hand side of Figure 1 by the highlighted and fragmented sections of code defining the 'code' attribute. Although the independence of the aspects is clear, we cannot describe (and re-use) the aspects as separate entities. It would be nicer if we could build up the semantic domains and the semantic functions piecewise, leaving the choice whether to decompose along productions or aspects up to the programmer. The lack of modularity in attribute grammars is a well-known problem and [17] surveys some of the techniques that have been employed to overcome it. Most of these techniques are however of a syntactic nature, and do not allow a separation into modules that can be separately compiled or even separately type checked.

5 An aspect–oriented compiler

So far, we have only reviewed existing mechanisms for writing attribute grammars by using extensible records in order to provide a background for our new modular approach to composing attribute grammars described in this section. The approach is 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 most tricky 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 as before

```
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]
```

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 the attribute grammar `ag ()`. (The dummy argument `()` to `ag` is required on account of a technicality in the type system of Haskell, known as the *monomorphism restriction*.)

```
>trans'
> = #code . transProg'
>   where
>    transProg' p      = #program g (transBlock' p)
>    transBlock' b     = #list g (transSList' b)

>    transSList' []    = #slist0 g
>    transSList' (s:ss)= #slist1 g (transStat' s)
>                                  (transSList' ss)

>    transStat' (Use x)  = #use g x
>    transStat' (Dec x)  = #dec g x
>    transStat' (Local b)= #local g (transBlock' b)
>    g = ag ()
```

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 type of `ag ()` is:

```
Rec ( program:: Block' -> Prog',
      list   :: SList' -> Block',
      slist0 :: Stat',
      slist1 :: Stat' -> SList' -> SList',
      use    :: String -> Stat',
      dec    :: String -> Stat',
      local  :: Block' -> Stat' )
```

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.

5.1 Combining aspects

In our example, the aspects are named `levels`, `envs`, `locss` and `code` and define, respectively, the attributes lexical level, environment, local variable, 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 completely below. Given this framework, it is clear that we can write new aspects and add them into our attribute grammar using these combinators. (The aspects, like `ag`, are affected by the monomorphism restriction and are written as functions which take the dummy argument `()`. This technicality allows us to avoid writing the type signatures of the aspects in the program. While this is not technically difficult, for large grammars it is tedious and far outweighs the inconvenience of writing the dummy argument `()`.)

5.2 Aspect definitions

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.

Before we define the aspects, let us be a bit more precise about attributes. There are two kinds of attribute, namely inherited and synthesised attributes. *Inherited* attributes describe information about the context in which a construct appears. Examples of inherited attributes are `level` and `env`. *Synthesised* attributes describe information computed from attributes of a constructs components, and examples of such attributes are `locs` and `code`.

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`.

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= inh1 (\b p -> (level = 0 | #x b)) ,
>    local  = inh1 (\b p -> (level = level p + 1
>                            | (nolevel (#x b))))
>    | 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 parents 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` 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` seen above to create a complete rule grammar. The given rules are written using lambda expressions (the `\` above can be read as λ) and in the case of the rule for `local`, the defined function takes the parent `p` and child block `b` and generates a record defining the `level` field as 1 more than the level of `p`.

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`. These attribute definitions are stored in the `x` field of the record describing the parent and children of the production `program`. As we will see in section 5.3, each attribute also has an attribute selection function, named by the attribute name, for accessing attributes values in the parent or children. This is seen above in the expression `level p`. 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. The functions `inh1` above and `syn0`, `syn1`, and `syn2` below provide a useful shorthand notation for converting rules which define only inherited or synthesised attributes into the more general type of rule expected by `over`. These functions are defined in the following section. The type of `levels` is given at the end of Section 5.3.3 after all the constituent types have been introduced. 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 crystal 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 two, which both dealt with inherited attributes.

```
>locss ()
> =(slist0= syn0 (\p      ->(locs= []   | #x p)),
>   slist1= syn2 (\a as p ->(locs= locs a
>                                  `union`
>                                  locs as
>                                      | #x p)),
>   use   = \a -> syn0 (\p->(locs= []   | #x p)),
>   dec   = \a -> syn0 (\p->(locs= [a]  | #x p)),
>   local = syn1 (\b p    ->(locs= []   | #x 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 an empty 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. Finally, the order in which we compose rules will not matter.

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.

The rest of this section develops our compositional scheme in detail. We take three bites at the cherry, rejecting two simpler designs before adopting a third. The development is necessarily technical and some readers may with to skip this section on their first reading.

5.3.1 First attempt at defining rules

What is the type of a rule like `list_level`? The simplest thought is this: it takes as arguments all `P`'s input attributes, and produces as output only the inherited `level` attribute for `P`'s child.

```
list_level
 :: (parentInh\level) =>
    Rec childSyn ->
    Rec (level :: Int | parentInh) ->
    (Rec (level :: Int), EmptyRow)

list_level c p = ((level = #level p), EmptyRec)
```

That is, `list_level` takes a record of synthesised attributes from the child (which it does not use), and of inherited attributes from the parent (from which it extracts the level). It produces a partial record of inherited attributes for the child (here containing only a `level` field), and of synthesised attributes for the parent (here empty). It is clear that this rule does not completely define the `List` production.

In general, the shape of a unary rule is this:

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

where all four parameters are understood to be row variables. As a slightly more interesting example than the type of `list_level`, consider the type of `list_code`:

```
list_code ::
 (childSyn\locs, childSyn\code,
 parentInh\level) =>
 Rule1 (locs :: [String],
        code :: [Instr]    | childSyn))

       (level :: Int | parentInh)

       EmptyRow  (code :: [Instr])
```

Indeed, this type reflects that the rule for `code` makes use of two synthesised attributes of the child, namely `locs` and `code`. It also records the dependence on the `level` attribute of the parent itself. Finally, from the last line in the type we can see that this rule defines no inherited attributes of the child, and that it defines precisely one synthesised attribute of the parent, namely `code`. One could give similar types for the other rules that make up the semantic function `list`.

There is a problem with this view of rules, however. As explained before, we need an operator `cat1` to compose rules by taking their union. With the proposed representation of rules, that operator would have to be defined as follows:

```
cat1 ::
 (disjoint inh1 inh2, disjoint syn1 syn2) =>
 Rule1 child parent inh1 syn1   ->
 Rule1 child parent inh2 syn2   ->
 Rule1 child parent (inh1 @ inh2)(syn1 @ syn2)

cat1 f g c p = (inh1 @ inh2, syn1 @ syn2)
               where (inh1,syn1) = f c p
                     (inh2,syn2) = g c p
```

Here (`@`) is symmetric concatenation, both on row variables and on records. Unfortunately the Trex type system does not directly support record concatenation; we are not even able to express evidence for disjointness of row variables as a type constraint (the predicate `disjoint` in the pseudo-code above).

5.3.2 Second attempt at defining rules

This lack of a concatenation operator is in fact a well-known problem when providing type systems for polymorphic extensible records, and a solution has been suggested by Rémy [28, 27]. Instead of directly concatenating records, we compose the functions that build up those records. The idea is very similar to the representation of lists by functions that aims to make list concatenation a constant time operation [11].

To apply Rémy's technique in the particular example of attribution rules, a rule takes two more parameters, which represent the existing output attributes. A rule does not return fixed records of defined attributes; instead it transforms existing definitions. For example, consider the rule `list_level`:

```
list_level cs pi ci ps
     = ((level = #level pi | ci), ps)
```

Its type is

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

That is, this rule contributes the definition of `level` to the inherited attributes of the child, whereas it leaves the synthesised attributes of the parent unchanged.

Note that for a rule with n children, the number of arguments in this representation will become $2*(n+1)$. To have so many arguments is somewhat clumsy, so we shall pair up the inherited and synthesised attributes of each symbol, in a type of nonterminals:

```
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. Consequently we can define a projection function for each attribute in our example, as shown below:

```
>level t = #level (#i t)
>env   t = #env   (#i t)
>locs  t = #locs  (#s t)
>code  t = #code  (#s t)
```

It will later become clear that the definition of NT has to be subtly revised by adding the `x` field for attribute definition contributions that was mentioned above.

Returning to the problem of defining a type of rules, we can now use a subtle adaptation of our earlier definition:

```
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 `list_level` rule can be written

```
list_level c p = ( (level = level p | #i c),
                   #s p)
```

and its type is

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

With this new 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`.

Unfortunately, however, this last definition of `cat1` fails as well, for rather more subtle reasons than our previous attempt. It is quite common for synthesised attributes to be defined in terms of each other. This does not happen in the attribute grammar of Section 4, but we have to cater for the possibility. As an example, consider the two rules

```
f  =  (\c p -> (#i c, (a = #b (#s p)) +1 | #s p))
g  =  (\c p -> (#i c, (b = 0             | #s p))
```

Rule `f` defines attribute `a` *in terms of another synthesised attribute*, named `b`. The type of `f` reflects this dependency:

```
f ::  (ps\b,ps\a) =>
      Rule1 (NT ci cs)
            (NT pi (b :: Int | ps))
            ci
            (a :: Int, b:: Int | ps)
```

The type of `g` is

```
g :: (ps\b) =>
     Rule1 (NT ci cs)
           (NT pi ps)
           ci
           (b :: Int | ps)
```

Attempting to concatenate these rules by `cat1 f g` results in a type error, because the type of `g` insists on the absence of the `b` field, whereas that field *is* present in the result of `f`, on account of its use. Paradoxically, concatenation in reverse order `cat1 g f` does not lead to a type error. Clearly it is unacceptable that composition of rules requires intimate knowledge of their dependencies.

5.3.3 Third attempt at defining rules

Close examination of the above example reveals the true source of the problem: we use the same records for applied occurrences (*i.e.* uses) and for defining occurrences of attributes. Instead of merely having records of inherited and synthesised attributes, we should separately keep track of the newly defined attributes of each nonterminal. We therefore revise our original definition of nonterminals, by adding an extra field named `x`:

```
>type NT ai as ax
>  = Rec (i :: Rec ai, s :: Rec as, x :: Rec ax)
```

The `i` and `s` fields record applied occurrences of attributes. The new `x` field records attribute definitions. If a nonterminal occurs as a child in a production, these are definitions of inherited attributes, and so we would expect `ax` to be a subset of `ai`. By contrast, if a nonterminal occurs as the parent in a production, the `x` field will record definitions of synthesised attributes, and thus `ax` is a subset of `as`.

The implementation of the type of rules is the same as before,

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

but the reading has changed. Now the results are extensions of the `x` fields of the child and parent respectively. Again consider the rule that defines the level of a child to be the level of the parent:

```
list_level c p = ( (level = level p | #x c),
                   #x p)
```

Instead of extending (`#i c`) (as we did before), we now extend (`#x c`). The type of this rule is

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

The type now makes a clear distinction between applied and defining occurrences of `level`.

Concatenation is defined as suggested before, but this time it realises concatenation of `x` fields:

```
>cat1 ::
>  Rule1 (NT ci cs cx) (NT pi ps px) cx'  px'  ->
>  Rule1 (NT ci cs cx')(NT pi ps px')cx'' px'' ->
>  Rule1 (NT ci cs cx) (NT pi ps px) cx'' px''

>cat1 f g c p  = g (i= #i c, s= #s c, x= cx')
>                 (i= #i p, s= #s p, x= px')
>               where (cx',px') = f c p
```

Let us now pause and consider the little counterexample that caused us to abandon the previous proposal for rule concatenation. Rule `f` defines a synthesised attribute `a` in terms of another synthesised attribute called `b`. That second attribute is defined in rule `g`:

```
f  =  (\c p -> (#i c, (a = #b (#s p)) +1 | #x p))
g  =  (\c p -> (#i c, (b = 0             | #x p))
```

The type of `f` now accurately reflects that `b` is used, but not defined in this rule:

```
f :: (ps\b,px\a) =>
      Rule1 (NT ci cs cx)
            (NT pi (b :: Int | ps) px)
            cx
            (a :: Int | px)
```

The type of `g` is

```
g :: (px\b) =>
     Rule1 (NT ci cs cx)
           (NT pi ps px)
           cx
           (b :: Int | px)
```

Concatenation of `f` and `g` in either order will result in a new rule of type

```
f `cat1` g ::
  (ps\b,px\a,px\b) =>
  Rule1 (NT ci cs cx)
        (NT pi (b :: Int | ps) px)
        cx
        (a:: Int, b::Int | px)
```

The new version of rule concatenation is indeed symmetric, as it ought to be. It follows that rules can be composed without knowledge of their dependencies. The concatenation operator does of course have an identity element, namely the rule that leaves all attribute definitions unchanged:

```
>none1 :: Rule1 (NT ci cs cx) (NT pi ps px) cx px
>none1 c p = (#x c, #x p)
```

As promised in Section 5.2, we can now provide the type for the aspect `levels`.

```
() ->
Rec(program::Rule1(NT ci cs cx) (NT pi ps px)
                   (level::Int | cx) px,
    local::  Rule1(NT ci cs (level::Int | cx'))
                   (NT (level::Int | pi') ps' px)
                   (level::Int | cx') px,
    list::   Rule1(NT ci1 cs1 cx1) (NT pi1 ps1 px1)
                   cx1 px1,
    slist0:: Rule0(NT ci2 cs2 cx2) cx2,
    slist1:: Rule2(NT ci3 cs3 cx3) (NT ci4 cs4 cx4)
                   (NT pi2 ps2 px2) cx3 cx4 px2,
    dec:: t->Rule0(NT ci5 cs5 cx5) cx5,
    use:: t->Rule0(NT ci6 cs6 cx6) cx6)
```

We now see clearly that `levels` is a function mapping the dummy argument `()` to a rule grammar. The resulting rule grammar defines `Rule1` rules for the `program` and `local` fields. The type shows that the `program` rule adds a definition of `level` to the childs `x` field and the `local` rule updates the childs `level` field using the `level` field from the parents inherited attributes. The types of the other rules are the default types.

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)
```

All the unary semantic functions of Section 4.2 are values of this type, in particular `list` is.

The operation `knit1` takes a rule, and it yields a semantic function. Let us call the two symbols involved `c` and `p`, short for `child` and `parent` respectively. The function that results from `knit1` takes the semantics of `c` (which is of type `Rec ci -> Rec cs`) as well as the inherited attributes for `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 inherited attributes of `c` starting from the inherited attributes of `p`, and it builds up the synthesised attributes of `p` starting from the empty record. 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 ci cs pi)
>               (NT pi ps EmptyRow)
>               ci
>               ps
>         -> Semfun1  ci cs   pi ps

>knit1 rule c pi
> = ps
>   where (ci,ps) = rule (i=ci,s=cs,x=pi)
>                        (i=pi,s=ps,x=EmptyRec)
>         cs      = c ci
```

The type of `knit1` shows that the rule must transform the inherited attributes `pi` into the inherited attributes `ci` of the child. Furthermore, the rule is required to yield the synthesised attributes `ps`, starting from the empty record. 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`. The arguments of `rule` are nonterminals: the applied occurrences of the attributes are filled in recursively, and the `x` fields are filled in as `pi` and `EmptyRec` respectively.

Note that the output attributes `ci` and `ps` are recursively defined in terms of themselves. It follows that the values of

synthesised attributes can depend on each other — which was the point of including the `x` field in the type of nonterminals. It is also possible for the inherited attributes of the child to be defined in terms of each other. There is a school of thought in the attribute grammar community which holds that such dependencies are bad style, and that one should use so–called *local* attributes instead to avoid repeated attribute computations in the same production. For us, local attributes are just synthesised attributes that are removed from the result of a semantic function, immediately after knitting.

Finally, the start symbol of the grammar does not have any inherited attributes, so we define a specialised knitting function for that:

```
>start f i = knit1 f i EmptyRec
```

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 details.

5.5 Derived combinators

Often rules define only inherited or only synthesised attributes. In those cases, the syntax for manipulating rules is somewhat cumbersome: one has to explicitly state that either set of attributes remains unchanged. It is useful, therefore, to have shorthands for rules that define inherited attributes only. The type of *inheriting rules* is

```
>type IRule1    child    parent   childX
>    =
>     child  ->
>     parent ->
>     Rec childX
```

and the injection injection function from inheriting rules to ordinary rules is given by

```
>inh1 :: IRule1 (NT ci cs cx)
>               (NT pi ps px)
>               cx'                 ->
>        Rule1  (NT ci cs cx)
>               (NT pi ps px)
>               cx'
>               px
>inh1 f c p = (f c p, #x p)
```

As can be seen from these definitions, inheriting rules leave the synthesised attributes of the parent unchanged. The rule `list_level`, which defines the `level` of the child to be the `level` of the parent, could have been written as an inheriting rule:

```
list_level =
    inh1 (\c p -> (level = level p | #x c))
```

Similarly we have a type of *synthesising rules*, with a corresponding injection function `syn1`.

5.6 Lifted combinators

We do not always wish to compose single rules, as the granularity of our compositions would then be too small. Instead we would prefer to compose rules at the level of grammars. This is in fact what we did in Section 5.1 where we defined the attribute grammar `ag ()` by composing aspects, not single rules. One can lift the combinators that we have defined above to rule grammars, applying them field by field.

For example, here is the lifting of `none`:

```
>none()   = (program   = none1,
>            list      = none1,
>            slist0    = none0,
>            slist1    = none2,
>            use       = \a -> none0,
>            dec       = \a -> none0,
>            local     = none1)
```

Again, we see that `use` and `dec` are special, because they take a string `a` and return an empty rule of arity 0.

Concatenation is similarly lifted to operate on all productions simultaneously, by pairing up corresponding fields, and again treating `use` and `dec` as special cases:

```
>cat u v
>  = (program = #program u `cat1` #program v,
>     list    = #list u    `cat1` #list v,
>     slist0  = #slist0 u  `cat0` #slist0 v,
>     slist1  = #slist1 u  `cat2` #slist1 v,
>     use     = \a -> #use u a  `cat0` #use v a,
>     dec     = \a -> #dec u a  `cat0` #dec v a,
>     local   = #local u  `cat1` #local v)
```

Finally, we lift the knitting operation. Note, however, that we need to use the specialised knitting operation `start` for the root production `program`:

```
>knit u
> = (program   = start (#program u),
>    list      = knit1 (#list u),
>    slist0    = knit0 (#slist0 u),
>    slist1    = knit2 (#slist1 u),
>    use       = \a -> knit0 (#use u a),
>    dec       = \a -> knit0 (#dec u a),
>    local     = knit1 (#local u))
```

Admittedly these lifted combinators reveal a shortcoming of our encoding using records. It would be nice if the lifted versions could be defined once and for all, using *map* and *zip* operations on records, taking the underlying context-free grammar as a parameter. It is however impossible to give types to these operations in the type system we have chosen to employ. As a consequence, each of the lifted combinators needs to be modified when a new production is added to the underlying context-free grammar. It is for this reason that we have chosen to lift only the minimum set of operations, and not to lift (for example) `inh1`.

6 Discussion

6.1 Aspect oriented programming

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, such as distributed systems, avionics and database programming. Indeed, Gregor Kiczales and his team at Xerox have initiated the study of *aspect oriented programming* in general terms [18], and the notion of *adaptive object oriented programming* of Karl

Lieberherr *et al.* shares many of these goals [24]. 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 employed here can be applied to writing aspect oriented programs in other problem domains as well.

It is worthwhile to point out a number of deviations from Kiczales' original notion of aspect oriented programming (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, 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, 29, 32]. 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.

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 [19, 13]. 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.

Readers who are familiar with attribute grammar based compiler generators may wonder whether some of the more advanced features found in such systems can be mimicked in our setting [9, 14, 16, 29, 21]. While we are still investigating this issue, a more substantial case study has already shown how our technique admits concepts such as *local attributes* and *higher-order attribute grammars* in a natural manner. That case study is the topic of a companion to the present paper [6]. Although an attribute grammar for the complete semantics of a production language has not been completed, we are confident (but not yet certain) that this method will scale to handle these larger attribute grammars. There is one extension to the type system that will enable us to define many more useful combinators, namely *first-class labels*. There is no theoretical difficulty with that extension, as shown in [8]. The definitions in the present paper would also look prettier if this feature were available. For example, we would be able to define

```
inh1 l f c p = ((l = f c p | # x c),#x p)
```

and hence the syntax for attribute definitions would be much less noisy.

It might appear from our path to the current set of combinators that a type system allowing record concatenation as a primitive would also simplify the code. This would however entail a loss of flexibility: by representing rules as *definition transformers*, we allow overriding of default definitions, which is not an option in the direct encoding using record concatenation.

6.3 Modular interpreters

The functional programming community has lately seen a lot of effort on the topic of *modular interpreters* and *modular compilers* [10, 22, 23]. In those works, the aim is to separate features such as the existence of side effects in the definition of semantic domains. Eugenio Moggi observed that the notion of monads is useful in making such a factorisation [25], and that idea was further explored by Phil Wadler and others in the context of functional programming [33]. The factorisation of compilers shown in this paper is somewhat different in nature. We wish to separate the description of aspects, even when they intimately depend on each other. Such separation is achieved by using the naming features of extensible records, and it is orthogonal to the separation achieved by means of monads.

6.4 Intentional programming

The ultimate aim of this work is to provide a suitable metalanguage for rapid prototyping of domain-specific languages in the *Intentional Programming* (IP) system under development at Microsoft Research [31]. Roughly speaking, an *intention* is a collective name both for productions and aspects as discussed in this paper. The aim of the IP system is to enable applications programmers to tailor programming languages to the needs of a particular domain. Its model of translation is entirely demand-driven, and the rules

for accessing information in the abstract syntax tree are very similar to those in attribute grammars. Until now, it was accepted that attribute grammars, though similar, were not sufficiently modular to be the basis of a meta-language (for specifying translations) in IP. In this paper we have presented a generalisation of attribute grammars that goes some way toward the criteria set out by Charles Simonyi in [31]. Some of the features we have proposed here (for example the default handling of inherited attributes) have a direct counterpart in the latest design of IP. There is currently some doubt whether demand-driven translation can be efficiently implemented in the IP project. Augusteijn (of Philips Research Laboratories) has built an attribute grammar evaluator based on lazy evaluation, and demonstrated its efficiency in industrial-strength applications [2]. We are confident, therefore, that the combination of lazy evaluation with strong typing will prove to be successful in IP as well.

Acknowledgements

The research reported in this paper was sparked by Paul Kwiatkowski, in his presentation of the new reduction engine in IP. Will Aitken explained the design of that reduction engine in more detail during a very fruitful visit to Oxford, and he provided several helpful insights on an early draft of this paper. Charles Simonyi set out the design criteria for aspect-oriented compilers that this work aspires to. Kevin Backhouse, Ivan Sanabria-Piretti, and Doaitse Swierstra pointed out many obscurities and suggested several improvements. Oege de Moor would like to thank Doaitse Swierstra for many illuminating discussions on the subject of attribute grammars over the past decade.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[2] A. Augusteijn. *Functional programming, program transformations and compiler construction.* PhD thesis, Department of Computing Science, Eindhoven University of Technology, The Netherlands, 1993. See also: `http://www.research.philips.com/generalinfo/special/elegant/elegant.html`.

[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 *Lecture Notes in Computer Science.* Springer Verlag, 1988.

[8] 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`.

[9] R. W. Gray, V. P. Heuring, S. P. Levi, A. M. Sloane, and W. M. Waite. Eli: A complete, flexible compiler construction system. *Communications of the ACM*, 35:121–131, 1992.

[10] W. L. Harrison and S. N. Kamin. Modular compilers based on monad transformers. In *Proceedings of the IEEE Conference on Computer Languages 1998.* IEEE Press, 1998.

[11] R. J. M. Hughes. A novel representation of lists, and its application to the function 'Reverse'. Technical report PMG-38, Programming Methodology Group, Chalmers Technological University, Sweden, 1984.

[12] T. Johnsson. Attribute grammars as a functional programming paradigm. In G. Kahn, editor, *Functional Programming Languages and Computer Architecture*, volume 274 of *Lecture Notes in Computer Science*, pages 154–173. Springer-Verlag, 1987.

[13] M. Jourdan. Strongly non-circular attribute grammars and their recursive evaluation. *SIGPLAN Notices*, 19:81–93, 1984.

[14] 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).

[15] 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 *Lecture Notes in Computer Science*, pages 380–400, 1991.

[16] U. Kastens, B. Hutt, and E. Zimmermann. *GAG: A Practical Compiler Generator*, volume 141 of *Lecture Notes in Computer Science.* Springer Verlag, 1982.

[17] U. Kastens and W. M. Waite. Modularity and reusability in attribute grammars. *Acta Informatica*, 31:601–627, 1994.

[18] G. Kiczales. Aspect-oriented programming. *ACM Computing Surveys*, 28A(4), 1996. See also: `http://www.parc.xerox.com/spl/projects/aop`.

[19] D. E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2:127–146, 1968.

[20] M. Kuiper and S. D. Swierstra. Using attribute grammars to derive efficient functional programs. In *Computing Science in the Netherlands CSN '87*, 1987. Available from: `ftp://ftp.cs.ruu.nl/pub/RUU/CS/techreps/CS-1986/1986-16.ps.gz`.

[21] 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 *Lecture Notes in Computer Science*. Springer Verlag, 1998.

[22] S. Liang and P. Hudak. Modular denotational semantics for compiler construction. In H. R. Nielson, editor, *Programming Languages and Systems – ESOP '96, 6th European Symposium on Programming*, volume 1058 of *Lecture Notes in Computer Science*, pages 219–234. Springer Verlag, 1996.

[23] S. Liang, P. Hudak, and M. P. Jones. Monad transformers and modular interpreters. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL '95)*, pages 333–343. ACM Press, 1995.

[24] K. J. Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*. PWS Publishing Company, 1996.

[25] E. Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.

[26] A. Ohori. A polymorphic record calculus and its compilation. *ACM Transactions on Programming Languages and Systems*, 17(6):844–895, 1995.

[27] 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.

[28] 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.

[29] T. W. Reps and T. Teitelbaum. *The Synthesizer Generator: A system for constructing language-based editors*. Texts and Monographs in Computer Science. Springer-Verlag, 1989.

[30] D. Rushall. *An attribute evaluator in Haskell*. Technical report, Manchester University, 1992. Available from URL `http://www-rocq.inria.fr/oscar/www/fnc2/AG.html`.

[31] 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/`.

[32] 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*, Lecture Notes in Computer Science. Springer Verlag, 1999. See also URL `http://www.cs.uu.nl/groups/ST/Software/index.html`.

[33] P. Wadler. Monads for functional programming. In *Program Design Calculi*, volume 118 of *NATO ASI Series F: Computer and systems sciences*. Springer Verlag, 1992.

[34] M. Wand. Type inference for record concatenation and multiple inheritance. *Information and Computation*, 93:1–15, 1991.