Data Structure Free Compilation

João Saraiva[1,2] and Doaitse Swierstra[1]
{saraiva,swierstra}@cs.uu.nl

[1] Department of Computer Science,
University of Utrecht, The Netherlands

[2] Department of Computer Science,
University of Minho, Braga, Portugal

Abstract. This paper presents a technique to construct compilers expressed in a strict, purely functional setting. The compilers do not rely on any explicit data structures, like trees, stacks or queues, to efficiently perform the compilation task. They are constructed as a set of functions which are directly called by the parser. An abstract syntax tree is neither constructed nor traversed. Such deforestated compilers are automatically derived from an attribute grammar specification. Furthermore this technique can be used to efficiently implement any multiple traversal algorithm.

1 Introduction

Traditionally, compilers are organized in two main phases: the *parsing phase* and the *attribute evaluation phase*, with an abstract syntax tree as the intermediate data structure. The *parser constructs* the *abstract syntax tree* and the *attribute evaluator decorates* that tree, *i.e.*, it computes *attribute* values associated to the nodes of the tree. In most implementations the attribute evaluator walks up and down in the tree, while in the mean time decorating it with attribute values. The abstract syntax tree guides the evaluator and stores attributes that are needed on different traversals of the compiler.

This paper presents a new technique for constructing compilers as a set of strict, side-effect free functions. Furthermore the compilers are completely *deforestated*, *i.e.*, no explicit intermediate data structure (*e.g.*, abstract syntax tree) has to be defined, constructed, nor traversed. The parser directly calls attribute evaluation functions, the so-called *visit-functions*. Moreover all the attributes are handled in a canonical way: they just show up as arguments and results of visit-functions.

Because our attribute evaluators are independent of any particular data structure definition, they are more generic than classical attribute evaluators. They are highly reusable and new semantics can easily be added to the attribute evaluators, even when separate analysis of compiler components is considered. For example, new productions can be incorporated to an existent compiler without changing its attribute evaluator. The visit-functions which implement the new productions are simply added to the compiler.

Although it is possible to apply these techniques in hand-written compilers, it is much easier to generate them from an *Attribute Grammar* [Knu68].

Our techniques were developed in the context of the incremental evaluation of (Higher-order) attribute grammars: efficient incremental behaviour is achieved by memoization of visit-function calls [PSV92].

In Section 2 we briefly introduce attribute grammars, present a simple attribute grammar and describe attribute evaluators based on the visit-sequence paradigm. In Section 3 λ-attribute evaluators are introduced. Section 4 deals with parse-time attribute evaluation. Section 5 discusses other applications of our techniques and section 6 briefly discusses the current implementation. Section 7 contains the conclusions.

2 Attribute Grammars

The compilers considered in this paper are specified through an Attribute Grammar (AG) [Knu68] which belong to the class of Ordered Attribute Grammars [Kas80]. These AGs have proven to be a suitable formalism for describing programming languages and their associated tools, like compilers, language based editors, etc. From an AG a *parser* and an *Attribute Evaluator* (AE) can automatically be derived.

This section introduces an attribute grammar which acts as the running example throughout this paper. Using it, we present the concept of *visit-sequences* [Kas80] which are the basis of our techniques.

2.1 The Block Language Example

This section presents a analyser for an extremely small language, called BLOCK, which deals with the *scope* of variables in a *block structured language*. An example BLOCK program is:

```
blk main : ( use y;
             blk (dcl w; use y; use w);
             dcl x; dcl x; dcl y; use w;
           );
```

This language does not require that declarations of identifiers occur before their first use. Furthermore an identifier from a global scope is visible in a local scope only if is not hidden by an a declarations with a same identifier in a more local scope. In a block an identifier may be declared at most once. The above program contains two errors: at the outer level the variable x has been declared twice and the use of the variable w has no binding occurrence at all.

Because we allow a *use-before-declare* discipline, a conventional implementation of the required analysis naturally leads to a program that traverses each block twice: once for processing the declarations of identifiers and constructing an environment and once for processing the uses of identifiers using the computed environment to check for the use of non-declared names. The uniqueness of names is checked in the first traversal: for each newly encountered declaration

it is checked whether that identifier has already been declared in this block. In that case an error message is computed. Since we need to distinguish between identifiers declared at different levels, we introduce an inherited attribute *lev* indicating the nesting level of a block. The environment is a list of bindings of the form $(name, lev)$.

In order to make the problem more interesting, and to demonstrate our techniques, we require that the error messages produced in both traversals are to be merged in order to generate a list of errors which follows the sequential structure of the program.

Figure 1 presents the attribute grammar defining the BLOCK language. We use a *standard* AG notation: Productions are labelled with a name for future references. Within the attribution rules of a production, different occurrences of the same symbol are denoted by distinct subscripts. Inherited (synthesized) attributes are denoted with the down (up) arrow $\downarrow$ ($\uparrow$). As usual in AGs we distinguish two classes of terminals: the *literal symbols* (*e.g.*, `':'`, `'decl'`, etc) which do not play a role in the attribution rules and the *pseudo terminal symbols* (*e.g.*, `name`), which are non-terminal symbols for which the productions are implicit (traditionally provided by an external lexical analyser). Pseudo terminal symbols are syntactically referenced in the AG, *i.e.*, they are used directly as values in the attribution rules. The attribution rules are written as HASKELL-like expressions. The semantic functions *mustbein* and *mustnotbein* define usual symbol table lookup operations.

```
root Prog
Prog     <↑ errors >
Prog → RootP (Its)
         Its.lev      = 0
         Its.dcli     = []
         Its.env      = Its.dclo
         Prog.errors  = Its.errors

Its      <↓ lev, ↓ dcli, ↓ env, ↑ dclo, ↑ errors >
Its → NilIts ()
         Its.dclo     = Its.dcli
         Its.errors   = []
    | ConsIts (It ';' Its)
         It.lev       = Its1.lev
         Its2.lev     = Its1.lev
         It.dcli      = Its1.dcli
         Its2.dcli    = It.dclo
         It.env       = Its1.env
         Its2.env     = Its1.env
         Its1.errors  = It.errors++Its2.errors

It       <↓ lev, ↓ dcli, ↓ env, ↑ dclo, ↑ errors >
It → Use ('use' name)
         It.dclo      = It.dcli
         It.errors    = name `mustbein` It.env
    | Decl ('dcl' name)
         It.dclo      = (name, It.lev) : It.dcli
         It.errors    = (name, It.lev) `mustnotbein` It.dcli
    | Block ('blk' '(' Its ')')
         Its.lev      = It.lev + 1
         Its.dcli     = It.env
         Its.env      = Its.dclo
         It.dclo      = It.dcli
         It.error     = Its.errors
```

Fig. 1. The Block Attribute Grammar.

2.2 Structured Visit-Sequences

The attribute evaluators considered in this paper are based on the visit-sequence paradigm [Kas80].

A visit-sequence describes, for a node in the tree, the sequence of states the node will go through when the abstract syntax tree is decorated. The essential property is that this sequence depends solely on the production at the node, and not on the context in which it occurs, hence we denote $vis(p)$ to denote the visit-sequence associated to production p. In a visit-sequence evaluator, the number of visits to a non-terminal is fixed, and independent of the production. We denote the number of visits of non-terminal X by $\mathtt{v}(X)$. Each visit i to a node labelled with a production for a non-terminal X has a fixed *interface*. This interface consists of a set of inherited attributes of X that are available to visit i and another set of synthesized attributes that are guaranteed to be computed by visit i. We denote these two sets by $A_{inh}(X,i)$ and $A_{syn}(X,i)$, respectively.

Visit-sequences are the outcome of attribute evaluation scheduling algorithms. They can be directly used to guide the decoration of a classical attribute evaluator [Kas91]. Visit-sequences, however, are the *input* of our generating process. It is then convenient to use a more structured representation of the visit-sequences. Thus, they are divided into *visit-sub-sequences* $vss(p,i)$, containing the instructions to be performed on visit i to the production p.

In order to simplify the presentation of our algorithm, visit-sub-sequences are annotated with *define* and *usage* attribute directives. Every visit-sub-sequence $vss(p,i)$ is annotated with the *interface* of visit i to X: $\mathtt{inh}(\alpha)$ and $\mathtt{syn}(\beta)$, where α (β) is the list of the elements of $A_{inh}(X,i)$ ($A_{syn}(X,i)$). Every instruction $\mathtt{eval}(a)$ is annotated with the directive $\mathtt{uses}(bs)$ which specifies the attribute occurrences used to evaluate a, *i.e.*, the occurrences that a depends on. The instruction $\mathtt{visit}(c,i)$ causes child c of production p to be visited for the ith time. We denote child c of p by p_c and the father (*i.e.* the left-hand side symbol of p) by p_0. The visit uses the attribute occurrences of $A_{inh}(p_c,i)$ as arguments and returns the attribute occurrences of $A_{syn}(p_c,i)$. Thus $\mathtt{visit}(c,i)$ is annotated with $\mathtt{inp}(is)$ and $\mathtt{out}(os)$ where is (os) is the list of the elements of $A_{inh}(p_c,i)$ ($A_{syn}(p_c,i)$).

Figure 2 presents the structured and annotated visit-sub-sequences[1] for the productions RootP and Block.

[1] The visit-sequences were obtained using the *Chained Scheduling Algorithm* [Pen94]. Chained scheduling is a variant of *Kastens' Ordered Scheduling Algorithm* [Kas80]. It was designed with the aim at minimizing the number of attributes that must be passed between traversals and, in this way, improving the behaviour of functional attribute evaluators. Chained scheduling chooses the attribute evaluation order such that every attribute is computed as early as possible. The visit-sequences of figure 2 are similar to the ones produced by Kastens' algorithm. The only exception is the schedule of the instructions $\mathtt{eval}(Its.lev)$. Kastens' algorithm schedules this instruction to the second visit-sub-sequence of production Block. In that case, the occurrence $It.lev$ must be retained for the second sub-sequence. A detailed analysis of both scheduling algorithms can be found in [Pen94] (chapter 5).

```
plan RootP
begin 1 inh() ,
  eval   (Its.lev)
         uses(),
  eval   (Its.dcli)
         uses(),
  visit  (Its, 1)
         inp(Its.lev, Its.dcli)
         out(Its.dclo),
  eval   (Its.env)
         uses(Its.dclo),
  visit  (Its, 2)
         inp(Its.env)
         out(Its.errors),
  eval   (Prog.errors)
         uses(Its.errors)
end 1    syn(Prog.errors)
```

```
plan Block
begin 1 inh(It.lev, It.dcli)
  eval   (Its.lev)
         uses(It.lev),
  eval   (It.dclo)
         uses(It.dcli)
end 1    syn(It.dclo)
begin 2 inh(It.env)
  eval   (Its.dcli)
         uses(It.env),
  visit  (Its, 1)
         inp(Its.dcli, Its.lev)
         out(Its.dclo),
  eval   (Its.env)
         uses(Its.dclo),
  visit  (Its, 2)
         inp(Its.env)
         out(Its.errors),
  eval   (It.errors)
         uses(Its.errors)
end 2    syn(It.errors)
```

Fig. 2. Structured Visit-Sequences: the attribute occurrence $\boxed{Its.lev}$ is defined in the first traversal of Block and is used in the next one.

3 Deriving λ-Attribute Evaluators

This section shows how to derive *purely functional and strict* attribute evaluators, starting from an available set of visit-sequences. The derived attribute evaluators are presented in Haskell. We use Haskell because it is a compact, well-defined and executable representation for our λ-attribute evaluators. We start by describing our techniques informally and by analysing a simple example. After that, we present the formal derivation of λ-attribute evaluators and we derive the evaluator for the Block language.

The λ-attribute evaluators consist of a set of *partial parameterized visit-functions*, each performing the computations of one traversal of the evaluator. Those functions return, as one of their results, the visit-functions for the next traversal. Performing the visit corresponds to *totally parameterising* the visit-functions and, once again returning the function for the next traversal. The main idea is that for each visit-sub-sequence we construct a function that, besides mapping inherited to synthesized attributes, also returns the function that represents the next visit. Any state information needed in future visits is passed on by partially parameterising a more general function. The only exception is the final visit-function which returns synthesized attributes.

Consider the following simplified visit-sub-sequences for production $X \rightarrow$ PROD (Y Z) (the annotations `inp` and `out` of the `visit` instructions are omitted since they are not relevant for this example):

plan PROD
begin 1 `inh`$(X.inh_1)$
  `visit` $(Y, 1)$
  `eval` $\cdots$
    `uses`$(X.inh_1, \cdots)$,
  `visit` $(Y, 2)$
  `eval` $(X.syn_1)$
    `uses`$(\cdots)$,
end 1 `syn`$(X.syn_1)$

begin 2 `inh`$(X.inh_2)$
  `visit` $(Z, 1)$
  `eval` $(X.syn_2)$
    `uses`$(X.inh_1, \cdots)$
end 2 `syn`$(X.syn_2)$

Observe that, the inherited attribute $X.inh_1$ must be explicitly passed from the first visit of X (where it is defined) to the second one (where it is used). The non-terminal Y is visited twice in the first visit to X. These two visit-sub-sequences above are implemented by the following two visit-functions:

$\lambda_{Prod^1}\ \lambda_{Y^1}\ \lambda_{Z^1}\ \boxed{inh_1} = ((\lambda_{Prod^2}\ \boxed{inh_1}\ \lambda_{Z^1}), syn_1)$
`where` $(\boxed{\lambda_{Y^2}}, \ldots) = \lambda_{Y^1}\ \ldots$
$(\ldots) = \boxed{\lambda_{Y^2}}\ \ldots$
$syn_1 = \cdots$

$\lambda_{Prod^2}\ \boxed{inh_1}\ \lambda_{Z^1}\ inh_2 = (syn_2)$
`where` $(\ldots) = \lambda_{Z^1}\ \ldots$
$syn_2 = f(\boxed{inh_1}, \ldots)$

$\boxed{inh_1}$ defined in λ_{Prod^1}
used in λ_{Prod^2}

$\boxed{\lambda_{Y^2}}$ partial parameterized in the first traversal and totally parameterized in the second one.

The visit-functions λ_{Y^1} and λ_{Z^1} define the computations of the first traversal of non-terminal symbols Y and Z. The attribute occurrence $X.x$ is passed from the first to the second traversal as a hidden result of λ_{Prod^1} in the form of an extra argument to λ_{Prod^2}. Note that **no** reference to visits for non-terminal symbol Y is included in λ_{Prod^2} since all the visits to Y occur in the first visit to P. Observe also that the function λ_{Z^1} is directly passed to the second visit to X, where the first visit to Z is performed.

The λ-attribute evaluators can be automatically derived from the visit-sub-sequences, by performing an *attribute lifetime analysis*: for each attribute occurrence it is known in which visit it is *defined* and in which visit(s) it is *used*. Thus, let us introduce two predicates `def` and `use`. The predicate `def`(p, a, v) denotes whether attribute a of production p is defined in visit v. Likewise, `use`(p, a, v) denotes whether attribute a of production p is used in visit v:

$$\begin{aligned}
\texttt{def}(p,a,v) &= \texttt{eval}(a) \in vss(p,v) \vee \texttt{inh}(\ldots,a,\ldots) \in vss(p,v) \\
&\quad \vee\ \texttt{out}(\ldots,a,\ldots)\ \in\ vss(p,v) \\
\texttt{use}(p,a,v) &= \texttt{uses}(\ldots,a,\ldots) \in vss(p,v) \vee \texttt{syn}(\ldots,a,\ldots) \in vss(p,v) \\
&\quad \vee\ \texttt{inp}(\ldots,a,\ldots)\ \in\ vss(p,v)
\end{aligned}$$

Pseudo terminal symbols may also be used as normal attribute occurrences within the attribute equations of the AG (like the symbol `name` of the BLOCK AG). Consequently, we need to perform a lifetime analysis of those symbols too. Thus, we extend the above predicates to work on terminal symbols too. The terminal symbols, denoted by Σ, are not defined in the attribute equations, but at parse-time. So, we assign visit number 0 to the parser. The predicate `def` is extended as follows:

$$\texttt{def}(p, a, 0) = a \in \Sigma$$

An attribute or pseudo terminal symbol of a production p is *alive* at visit i, if it is defined in a previous visit and it is used in visit i or later. For each production p and for each of its visits i, with $1 \leq i \leq \texttt{v}(p_0)$, we define the set $alive(p, i)$ which contains the *live* occurrences on visit i. It is defined as follows:

$$alive(p, i) = \{\ a \mid \texttt{def}(p, a, k)\ \wedge\ \texttt{use}(p, a, j)\ \wedge\ k < i \leq j\}$$

Let us concentrate now on the analysis of the visits to the non-terminal symbols of the grammar. Let $alive\_visits(p, c, v)$ denote the list of visits to child c of production p, which have to be performed in visit-sub-sequence v to p or in later ones. This list is defined as follows:

$$alive\_visits(p, c, v) = [\ \texttt{visit}(c, i) \mid \texttt{visit}(c, i)\ \in\ vss(p, j)\ ,\ v \leq j \leq \texttt{v}(p_0)]$$

Consider the visit-sub-sequences of production PROD. For the first sub-sequence we have the following visits: $alive\_visits(\text{PROD}, 1, 1) = [\texttt{visit}(p_1, 1), \texttt{visit}(p_1, 2)]$ and $alive\_visits(\text{PROD}, 2, 1) = [\texttt{visit}(p_2, 1)]$. That is, in the first visit to PROD or later ones the non-terminal symbol Y is visited twice and the symbol Z is visited once. Note that according to the visit-sub-sequences the single visit to Z is performed in the second visit of PROD. Consider now the visit-function λ_{Prod^1}. Observe that its arguments contain the reference to the first traversal of Y only (argument λ_{Y^1}). The function for the second traversal is obtained as a result of λ_{Y^1}. Observe also that the reference to the visit to Z is passed on to the second traversal of PROD, where it is called. That is, the arguments of the visit-function contain a reference to the earliest visit (function) which has to be performed for all *alive* non-terminal symbols.

In order to derive our visit-functions we need references (the visits-functions) to the earliest visit-function: all following references are returned by evaluating the previous ones. Thus, we define the function $inspect(p, v)$ which takes the head of the list returned by *nt_vis* (*i.e.*, the following visit), for all non-terminal symbols of production p. This is a partial function, since the list returned by *nt_vis* may be empty. This occurs when no further visits to a non-terminal symbol are performed. This function is defined as follows:

$$inspect(p, v) = \quad \{\ hd\ alive\_visits(p, c, v)\ :\ alive\_visits(p, c, v) \neq []\ \wedge\ p_c \in N\ \}$$

, where hd is the usual operation that returns the head of a list and N denotes the set of non-terminal symbols.

We describe now the derivation of the λ-attribute evaluator. For each production p and for each traversal i of non-terminal symbol p_0 a visit-function λ_{p^i} is derived. The arguments of this visit-function are:

1. The attribute occurrences which are alive at visit i, $alive(p,i)$,
2. The deforestated visit-functions derived for the right-hand side symbols of p which are inspected in traversal i or later, $inspect(p,i)$, and
3. The inherited attributes of traversal i, *i.e.*, $A_{inh}(p_0,i)$.

The result is a tuple of which the first element is the partial parameterized function for the next traversal and the other elements are the synthesized attributes, *i.e.*, $A_{syn}(p_0,i)$. Thus, the visit-functions have the following signature:

$$\lambda_{p^i} :: <type\_pp\_args(p,i)> \ \mathcal{T}(inh\_1) \rightarrow \cdots \rightarrow \mathcal{T}(inh\_k) \rightarrow (\mathcal{T}(\lambda_{p^{i+1}}), \mathcal{T}(syn\_1), \ldots, \mathcal{T}(syn\_l))$$

, with $\{inh\_1, \ldots, inh\_k\} = A_{inh}(p_0,i)$, $\{syn\_1, \ldots, syn\_l\} = A_{syn}(p_0,i)$. $\mathcal{T}(a)$ should be interpreted as the derived type for element a. The fragment $<type\_pp\_args(p,i)>$ denotes the type of the elements in $alive(p,i)$ and in $inspect(p,i)$. This fragment is defined as follows:

$$<type\_pp\_args(p,i)> = \mathcal{T}(a_1) \rightarrow \cdots \rightarrow \mathcal{T}(a_m) \rightarrow \mathcal{T}(\lambda_{vt_1}) \rightarrow \mathcal{T}(\lambda_{vt_n}) \rightarrow$$

, for all a_i such that $a_i \in alive(p,i)$ and for all vt_i such that $vt_i \in inspect(p,i)$.

The visit-function which performs the last traversal of a non-terminal does not return any partial parameterized visit-function. Its signature is:

$$\lambda_{p^n} :: <type\_pp\_args(p,i)> \ \mathcal{T}(inh\_1) \rightarrow \cdots \rightarrow \mathcal{T}(inh\_k) \rightarrow (\mathcal{T}(syn\_1), \ldots, \mathcal{T}(syn\_l))$$

Let us now derive the code of the visit-function λ_{p^i}. It looks as follows:

$$\begin{array}{l} \lambda_{p^i} \ <par\_par(p,i)> \ <inherited(i)> = \\ \quad ((\lambda_{p^{i+1}} \ <par\_par(p,i+1)>), <synthesized(i)>) \\ \texttt{where} <body(i)> \end{array}$$

and the visit-functions which performs the last traversal is:

$$\begin{array}{l} \lambda_{p^n} \ <par\_par(p,n)> \ <inherited(i)> = (<synthesized(i)>) \\ \texttt{where} <body(n)> \end{array}$$

where the code fragments defining the inherited and synthesized attributes look as follows:

$$\begin{array}{ll} <inherited(i)> & = inh\_1 \ inh\_2 \ldots inh\_k \\ <synthesized(i)> & = syn\_1, syn\_2, \ldots, inh\_l \end{array}$$

The code fragment $<par\_par(p,j)>$ denotes the *partial parameterisation* of the next visit-function.

$$<par\_par(p,j)> = a_1 \ldots a_m \ \lambda_{vt_1} \ldots \lambda_{vt_n}$$

The body $<body(i)>$ of each visit-function λ_{p^i} is generated according to the instructions of the visit-sub-sequence $vss(p,i)$. Every attribute equation of the form

$$\begin{array}{l}\texttt{eval}\ (p_q.a) \\ \qquad \texttt{uses}(attroccs)\end{array}$$

, defining an attribute occurrence $p_q.a = f\ (attroccs)$ of production p, generates an equation

$$(a_q) = f\ (attroccs)$$

Attribute $p_r.a$ occurring in $attroccs$ is replaced by a_r. Local attribute occurrences of productions are copied literally to the body of the respective visit-functions.

Every instruction $\texttt{visit}(c,i)$ defining the visit i to non-terminal occurrence p_c introduces a call. Two cases have to be distinguished:

If $i < \texttt{v}(p_c)$ then the call returns the partial parameterized function for the next traversal. The following equation is generated:

$$(\lambda_{p_c^{i+1}}, syn\_1_c, \ldots, syn\_j_c) = \lambda_{p_c^i}\ inh\_1_c \ldots inh\_l_c$$

If $i = \texttt{v}(p_c)$ then only the synthesized attributes are computed by the function call.

$$(syn\_1_c, \ldots, syn\_j_c) = \lambda_{p_c^i}\ inh\_1_c \ldots inh\_l_c$$

, with $\{inh\_1, \ldots, inh\_j\} = A_{inh}(p_c, i)$ and $\{syn\_1, \ldots, syn\_l\} = A_{syn}(p_c, i)$.

Let us return to the Block AG and derive the visit-function for the most intricate production: the production Block. First we compute the set $alive$ and the visit-trees for each visit to that production.

$$\begin{array}{ll} alive(\text{Block}, 1) & = \{\ \} \\ alive(\text{Block}, 2) & = \{\ lev_2\ \} \\ inspect(\text{Block}, 1) & = \{\ Its^1\ \} \\ inspect(\text{Block}, 2) & = \{\ Its^1\ \} \end{array}$$

As expected, the attribute occurrence $It.lev$ must be passed from the first to the second traversal. The two visit-functions derived for this production are:

$\lambda_{Block^1}\ <par\_par(\text{Block}, 1)>\ lev_1\ dcli_1 = ((\lambda_{Block^2}\ <par\_par(\text{Block}, 2)>, dclo_1)$
$\texttt{where}\ <body(1)>$

λ_{Block^2} $<par\_par(\textsc{Block}, 2)>$ $env_1 = (errors_1)$
where $<body(2)>$

, where the fragments $<par\_par>$ are:

$$<par\_par(\textsc{Block}, 1)> = \lambda_{Its^1}$$
$$<par\_par(\textsc{Block}, 2)> = lev_2\ \lambda_{Its^1}$$

The body of the visit-functions is trivially derived from the corresponding visit-sub-sequences (see figure 2): we present only the body of the visit-function for the second traversal to the production Block.

$$\begin{array}{lll} <body(2)> = & dcli_2 & = env_1 \\ & (\lambda_{Its^2}, dclo_2) & = \lambda_{Its^1}\ dcli_2\ lev_2 \\ & errors_2 & = \lambda_{Its^2}\ dclo_2 \\ & errors_1 & = errors_2 \end{array}$$

The complete λ-attribute evaluator derived from the Block attribute grammar is presented in figure 3 (some copy rules were trivially removed from the AE code).

```
λRootP¹ λIts¹ = errors₂
where lev₂ = 1
      dcli₂ = []
      (λIts², dclo₂) = λIts¹ dcli₂ lev₂
      errors₂ = λIts² dclo₂

λConsIts¹ λIt¹ λIts₂¹ dcli lev =
          ((λConsIts² λIt² λIts₂²), dclo₃)
where (λIt², dclo₂) = λIt¹ dcli lev
      (λIts₂², dclo₃) = λIts₂¹ dclo₂ lev

λNilIts¹ dcli lev = ((λNilIts²), dcli)
λConsIts² λIt² λIts₂² env = errors
where errors₂ = λIt² env
      errors₃ = λIts₂² env
      errors = errors₂ ++ errors₃

λNilIts² env = []
```

```
λBlock¹ λIts¹ dcli lev =
        ((λBlock² lev₂ λIts¹), dcli)
where lev₂ = lev + 1

λDecl¹ name dcli lev = ((λDecl² errors), dclo)
where dclo = (name, lev) : dcli
      errors = (name, lev) `mustnotbein` dcli

λUse¹ name dcli lev = ((λUse² name), dcli)

λBlock² lev₂ λIts¹ env = errors₂
where (λIts², dclo₂) = λIts¹ env lev₂
      errors₂ = λIts² dclo₂

λDecl² errors env = errors

λUse² name env = errors
where errors = name `mustbein` env
```

Fig. 3. The complete λ-attribute evaluator for the Block Language.

As a result of our techniques all visit-functions have become combinators, *i.e.*, they do not refer to global variables. The type of the λ-attribute evaluator is the type of the visit-function of the root symbol:

$$\lambda_{RootP^1} :: ([a] \rightarrow Int \rightarrow ([a] \rightarrow b, [a])) \rightarrow b$$

This evaluator returns the attribute errors (type b) and it has one function as argument: the visit-function which performs the first visit to the non-terminal symbol *Its*. This function has the initial environment (type $[a]$) and the level (type Int) as arguments and it returns a pair: the function for the second visit to *Its* (with type $[a] \rightarrow b$) and the total environment.

As a result of generating HASKELL code we inherit many useful properties of this language. The λ-attribute evaluator of figure 3, for example, is completely polymorphic. In this evaluator nothing is defined about the type of the identifiers of the language. The identifiers are provided by an external lexical analyser. They can be a sequence of characters, a single character or even a numeral. The AE can be reused in all those cases, provided that the semantic functions *mustbein* and *mustnotbein* are defined on that type too.

This approach has the following properties:

- The λ-attribute evaluators have the tendency to be more polymorphic.
- The evaluators are *data type independent* and, thus, new semantics can be easily added: for example, new productions can be incorporated to a compiler without having to change the evaluator. This property will be explained in section 4.
- Attribute instances needed in different traversals of the evaluator are passed between traversals as results/arguments of partial parameterized visit-functions. No additional data structure is required to handle them, like trees [Kas91,PSV92,SKS97] or stacks and queues [AS91].
- The resulting evaluators are higher-order attribute evaluators. The arguments of the evaluators visit-functions are other AE visit-functions.
- The visit-functions find *all the values* they need in their arguments.
- No pattern matching is needed to detect the production applied at the node the evaluator is visiting.
- The visit-functions are strict in all their arguments, as a result of the order computed by the AG ordered scheduling algorithm.
- Efficient memory usage: data not needed is no longer referenced. References to grammar symbols and attribute instances can efficiently be discarded as soon as they have played their semantic role.
- The code of the attribute evaluator is shorter because no data structures are defined.

4 Parse-Time Attribute Evaluation

Traditional attribute grammar systems construct an abstract syntax tree during the parsing of the source text. This tree is used later to guide the attribute evaluator. For some classes of attribute grammars the construction of the abstract

syntax tree may be avoided and the attribute evaluation may be performed in conjunction with the parsing (L-attributed grammars). In this case, it is the parser which guides the attribute evaluation. Such a model has several advantages, namely speed and space requirements. Methods exist which make one-pass attribute evaluation during parsing possible [ASU86].

Parse-time attribute evaluation is achieved as a by-product of our AG implementation: the parser directly calls the visit-functions which perform the first traversal of the λ-attribute evaluator.

Consider again the production BLOCK. The classic fragment of the parser derived from the AG which defines this production and constructs the corresponding tree node looks as follows[2]:

$$\begin{array}{l} \mathit{It} : \texttt{blk '(' } \mathit{Its} \texttt{ ')'} \\ \quad \{ \textsc{Block} \ \ \$3 \ \ \} \end{array}$$

The type of the parser derived from this specification is a function from a string (*i.e.*, the source text) to the type of the term defined by the production.

$$\mathit{parser\_It} :: [\mathit{Char}] \ \rightarrow \ \mathit{It}$$

, where *It* is a declared data type.

Using our techniques the parser derived from the AG generates a call to the attribute evaluator visit-functions which perform its first traversal. Our parser looks as follows:

$$\begin{array}{l} \mathit{It} : \texttt{blk '(' } \mathit{Its} \texttt{ ')'} \\ \quad \{ \lambda_{Block^1} \ \ \$3 \ \ \} \end{array}$$

The deforestated visit-functions are partially parameterized with the arguments available at parse-time. Those arguments are the other visit-functions which are partially parameterized when parsing the grammar symbols of right-hand side of the production. No explicit abstract syntax tree is constructed.

Consider the visit-function λ_{NilIts^1} which returns the visit-function λ_{NilIts^2}. The function λ_{NilIts^2} is a constant function: it does not depends on its arguments. That is, it does not use the inherited attribute *env* and always returns an empty list (*i.e.*, it evaluates the synthesized attribute *errors*). As result, λ_{NilIts^2} can be computed at parse-time.

Generally, every visit-function, derived from a visit-sub-sequence i which does not have inherited attributes (annotation `inh`) or which does not use its inherited attributes, can be evaluated in visit $i-1$. It has all the arguments it needs available on the previous visit. Observe that the visit-functions derived for productions applied to non-terminal symbols which only have synthesized attributes can be evaluated at parse-time. This is particularly important when implementing processors that produce code as the the input is being processed, *i.e.*, for implementing online algorithms.

[2] We use HAPPY [Mar97] notation, an Yacc equivalent for Haskell.

Suppose that we want to extend the BLOCK language with named blocks. That is, the BLOCK AG is extended with the following production:

$$It \rightarrow \text{NAMEDBLK}\ (\texttt{'blk'}\ \texttt{name}\ \texttt{':'}\ \texttt{'('}\ Its\ \texttt{')'})$$

In traditional AG implementations, the attribute evaluator would have to be modified, since the type of the abstract syntax tree changes. Our implementation, however, is independent of the abstract tree data type. The attribute evaluator of figure 3 can be reused, without any modification, to implement the AG extension. The only part of the compiler that has to be modified is the parser: the new production must be included, obviously. Furthermore the visit-functions $\lambda_{NamedBlk^i}$ which implement the different visits to the production have to be added to the compiler as a separate module. The new parser fragment looks as follows:

$$\begin{array}{l} It\ :\ \texttt{blk}\ \texttt{name}\ \texttt{':'}\ \texttt{'('}\ Its\ \texttt{')'} \\ \quad \{\ \lambda_{NamedBlk^1}\ \ \$2\ \ \$5\ \ \} \end{array}$$

The signature of the visit-functions $\lambda_{NamedBlk^i}$ must follow the partitions of the non-terminal symbol It (*i.e.*, the symbol on the left-hand side of the production).

This property of our AG implementation is particularly important when designing language processors, in a component based style: AG components and the respective evaluators can be easily reused and updated, even when separate analysis and compilation of such components is considered [Sar].

5 Applications

This section describes how our techniques are used in the context of *Higher-Order Attribute Grammars*, *Incremental Attribute Evaluation*, *Composition of Attribute Grammars* and *Lazy Attribute Evaluation*.

Higher-Order Attribute Grammars (HAG) [VSK89]: the techniques described in this paper were developed in the context of the (incremental) evaluation of HAGs. HAGs are an important extension to the classical AG formalism: attribute grammars are augmented with *higher-order attributes*. Higher-order attributes are attributes whose value is a tree with which we associate attributes again. Attributes of these so-called *higher-order trees*, may be higher-order attributes again. Higher-order attribute grammars have two main characteristics: first, when a computation can not easily be expressed in terms of the inductive structure of a tree, a better suited structure can be computed first, and secondly, every computation (*i.e.*, inductive semantic function) can be modeled through attribute evaluation. Typical examples of the use of higher-order attributes are mapping a concrete syntax tree into an abstract one and modelling symbol table lookups.

A higher-order attribute grammar may have several higher-order attributes (*i.e.*, higher-order trees). Thus, an attribute evaluator for HAG may contain

a possibly large number of higher-order trees. As a result the efficiency of the attribute evaluator may be affected by the construction and destruction of those trees. The technique described in this paper can be used to implement higher-order attribute grammars [Sar]. The higher-order attributes are represented by their initial visit-functions.

Incremental Attribute Evaluation: one of the key features of our AG implementation is that the attribute evaluators are constructed as a set of strict functions. Consequently, an incremental attribute evaluator can be obtained through standard function caching techniques [PSV92]. The incremental behaviour is achieved by storing in a cache calls to the attribute evaluator functions and by reusing their results when such functions are later applied to the same arguments. This is the most efficient and elegant approach for the incremental evaluation of HAGs [Pen94,CP96]. Previous techniques, however, rely on additional data structures, *e.g.*, *a binding tree*, to handle attribute instances needed in different traversals of the evaluator [Pen94]. A large number of calls to tree constructor functions may have to be cached since the number of binding trees is quadratic in the number of traversals. Such an approach, albeit optimal in the number of reevaluations, can result in a substantial decrease of performance of the incremental evaluator due to the fast growth, and consequent overhead, of the cache [SKS96]. Using λ-attribute evaluators no constructor functions exist (*i.e.*, abstract tree nor binding tree constructor functions) and thus no constructor functions have to be cached! The calls to the visit-functions are the only calls actually cached. The incremental evaluators have less cache overhead [Sar].

Composition of Attribute Grammars: consider a compiler organized as follows: it has two AGs of the form $ag_1 :: T_1 \rightarrow T_2$ and $ag_2 :: T_2 \rightarrow T_3$. That is, it has two AGs which are *glued* by the intermediate tree T_2. Using traditional AG techniques the tree T_2 would have to be constructed. Using our techniques the attribute evaluator of ag_1 directly calls the deforestated visit-functions of the ag_2 attribute evaluator, like in a *normal* multiple traversal AE. As result, no intermediate tree is constructed. This strategy holds even when separate analysis (compilation) of both AGs is considered. In [Sar] this composition of attribute grammar components is presented.

Lazy Attribute Evaluation: attribute grammars can be easily and elegantly implemented in a programming language with lazy semantics [KS87,Joh87,SA98]. The techniques described here are orthogonal to the lazy mapping of attribute grammars. See [Sar] for the formal derivation of deforestated and lazily implementation of attribute grammars.

6 Implementation

The techniques described in this paper have been implemented in the Lrc system [KS98], a purely functional attribute grammar system. The Lrc processes Higher-Order Attribute Grammars, written in a super-set of Ssl, the *synthesizer*

specification language [RT89], and produces purely functional attribute evaluators.

We have developed a new back-end to the LRC in order to generate HASKELL based attribute evaluators. A (coloured) LaTeX version of such attribute evaluators is also generated by the LRC system. Actually the HASKELL code presented in this paper (including the AE of figure 3) was automatically produced by LRC from a SSL specification. The deforestation of HAGs and the lazy implementation of attribute grammars, discussed in section 5, have also been implemented.

Several small and medium size λ-attribute evaluators have been translated into C in order to use the caching mechanism of the LRC system and to achieve incremental evaluation. The automatic generation of λ-attribute evaluators in the C language is currently being incorporated to LRC.

7 Conclusions

This paper introduced a new technique for compiler construction. The compilers are constructed as a set of strict and purely functional visit-functions. All explicit data structure definition, construction and traversals have been removed. As a result of our technique the λ-attribute evaluators are totally generic and can easily be reused and updated across different applications. Because constructor funtions are never used, and all case statements have been "compiled way", one might in general expect better performance, since the flow of information is now clearly represented in the structure of the paremeters and results of the visit-functions. Thus, many compiler optimization techniques become enabled. Furthermore parse-time attribute evaluation is achieved as a by-product: the parser directly calls the visit-functions.

A simple language was analysed and the respective compiler was automatically derived from an attribute grammar. A mapping from attribute grammars into strict and purely functional attribute evaluator was defined. This mapping has been implemented in the LRC system.

The technique described in this paper is not restricted to the context of compiler construction only. It can be used to efficiently implement any algorithm which performs multiple traversals over a recursive data structure. It was used, for example, to implement a pretty printing combinator library [SAS98], which is a four traversal algorithm and that would have been extremely complicated to construct by hand.

References

[AS91] Rieks Akker and Erik Sluiman. Storage Allocation for Attribute Evaluators using Stacks and Queues. In H. Alblas and B. Melichar, editors, *International Summer School on Attribute Grammars, Applications and Systems*, volume 545 of *LNCS*, pages 140–150. Springer-Verlag, 1991.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison Wesley, 1986.

[CP96] Alan Carle and Lori Pollock. On the optimality of change propagation for incremental evaluation of hierarchical attribute grammars. *ACM Transactions on Programming Languages and Systems*, 18(1):16–29, January 1996.

[Joh87] Thomas 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, September 1987.

[Kas80] Uwe Kastens. Ordered attribute grammars. *Acta Informatica*, 13:229–256, 1980.

[Kas91] Uwe Kastens. Implementation of Visit-Oriented Attribute Evaluators. In H. Alblas and B. Melichar, editors, *International Summer School on Attribute Grammars, Applications and Systems*, volume 545 of *LNCS*, pages 114–139. Springer-Verlag, 1991.

[Knu68] Donald E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, June 1968.

[KS87] Matthijs Kuiper and Doaitse Swierstra. Using attribute grammars to derive efficient functional programs. In *Computing Science in the Netherlands CSN'87*, November 1987.

[KS98] Matthijs Kuiper and João Saraiva. Lrc - A Generator for Incremental Language-Oriented Tools. In Kay Koskimies, editor, *7th International Conference on Compiler Construction*, volume 1383 of *LNCS*, pages 298–301. Springer-Verlag, April 1998.

[Mar97] Simon Marlow. *Happy User Guide*. Glasgow University, December 1997.

[Pen94] Maarten Pennings. *Generating Incremental Evaluators*. PhD thesis, Department of Computer Science, Utrecht University, The Netherlands, November 1994. `ftp://ftp.cs.ruu.nl/pub/RUU/CS/phdtheses/Pennings/`.

[PSV92] Maarten Pennings, Doaitse Swierstra, and Harald Vogt. Using cached functions and constructors for incremental attribute evaluation. In M. Bruynooghe and M. Wirsing, editors, *Programming Language Implementation and Logic Programming*, volume 631 of *LNCS*, pages 130–144. Springer-Verlag, 1992.

[RT89] T. Reps and T. Teitelbaum. *The Synthesizer Generator*. Springer, 1989.

[SA98] S. Doaitse Swierstra and Pablo Azero. Attribute Grammars in a Functional Style. In *Systems Implementation 2000*, Berlin, 1998. Chapman & Hall.

[Sar] João Saraiva. *Purely Functional Implementation of Attribute Grammars*. PhD thesis, Department of Computer Science, Utrecht University, The Netherlands. (In preparation).

[SAS98] Doaitse Swierstra, Pablo Azero, and João Saraiva. Designing and Implementing Combinator Languages. In *Third International Summer School on Advanced Functional Programming, Braga, Portugal*, 1998.

[SKS96] João Saraiva, Matthijs Kuiper, and Doaitse Swierstra. Effective Function Cache Management for Incremental Attribute Evaluation. Technical report UU-CS-1996-50, Department of Computer Science, Utrecht University, November 1996.

[SKS97] João Saraiva, Matthijs Kuiper, and Doaitse Swierstra. Specializing Trees for Efficient Functional Decoration. In Michael Leuschel, editor, *ILPS97 Workshop on Specialization of Declarative Programs and its Applications*, pages 63–72, October 1997. (Also available as Technical Report CW 255, Department of Computer Science, Katholieke Universiteit Leuven , Belgium).

[VSK89] Harald Vogt, Doaitse Swierstra, and Matthijs Kuiper. Higher order attribute grammars. In *ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*, volume 24, pages 131–145. ACM, July 1989.