

# Meiage

(preliminary version for Eifl)

Joost Verhoog  
jverhoog@cs.uu.nl

December 2, 2004

## Abstract

We discuss our improvement of UU-AG with Kasten's Chained Scheduling Algorithm, and give a full example.

## 1 Introduction

Meiage is an extension of the UU-AG system. In this report, we assume familiarity with this system.

Many programs that work on trees are essentially attribute grammars evaluators. Attribute grammars can be seen as decorated trees. Each nonterminal is a node in the tree, and each terminal is a leaf. With each node a number of attributes are associated, which can be divided into inherited and synthesized attributes. The semantics of a node in the tree is a function taking the inherited and returning the synthesized attributes.

One can describe an attribute grammar by giving for each node in the tree the value of its synthesized attributes and the values of the inherited attributes of its children, using the node's inherited attributes and the synthesized attributes of its children. Given this rather abstract description, an attribute grammar evaluator is needed, which computes the value of all (or some) attributes in a tree.

In order to do this in a strict setting, an order of evaluation is needed. For an attribute grammar in its original form, this is not always possible, so only subclasses of attributes grammars are considered. Pennings [Pen94] gives an overview of these.

In a non-strict, or lazy, setting, things become a lot easier. The attribute grammar can be described in the original form, depending on the laziness of the language to handle the order of evaluation. This is exactly what the UU-AG

system does. The advantage of this is that the implementation is easy to understand, and that it can handle a large class of attribute grammars. A disadvantage is that it's slower than a strict implementation.

This is the problem that Meiage aims to solve. Meiage stands for 'More Efficient Implementation of Attribute Grammar Evaluators'. We want the best of both worlds, so being able to handle all attribute grammars that can be handled by the current UU-AG system, but using our knowledge of the structure of the attribute grammar to speed up the computation. We can do this by using a technique that is used in the creation of strict attribute grammar evaluators: Kasten's Chained Scheduling Algorithm.

## 2 Kasten's algorithm in UU-AG

### 2.1 Kasten's algorithm

Kasten [Kas80] describes an algorithm for computing the values of the attributes of a tree using visit functions. Its description in pseudo-code in [Pen94] is very illustrative in this respect. The idea is that we try to find an ordering in the attributes, such that every attribute can be computed using only the values of attributes preceding it. We compute visit functions for every nonterminal. Each visit function takes some inherited, and gives some synthesized attributes of the nonterminal. We will not describe the algorithm in detail here. We only show how we implemented it in the UU-AG system.

### 2.2 Adaptation to UU-AG

Because visit functions might need values that were computed by previous visits, and we are in a stateless environment, these values need to be passed to them as parameters. The easiest way to accomplish this is by using a form of continuation: each visit function does not only return its synthesized attributes, but also the next visit function. In this way it can already partially apply it to the values that it needs from this visit and previous visits.

Because the semantic functions that the UU-AG-compiler generates are usually used in parsers, we would like to be able to use Meiage, and not having to change the parsing. This is possible, under the following restrictions:

1. The first visit function has the same name as the semantic function generated by the original UU-AG system.
2. The root is only visited once.

The first restriction is easily satisfied; we just name it that way. We can also obey the second restriction, because in a valid attribute grammar it is always possible to compute the synthesized from the inherited attributes of the root.

### 3 Description of the implementation

The implementation consists of the following steps, which we will discuss below.

1. Find dependencies
2. Compute interfaces
3. Check ordering
4. Generate visit sub-sequences
5. Generate semantic functions
6. Include in distribution

#### 3.1 Find dependencies

A computation of the dependencies is already available in the UU-AG system. It is used for the detection of cycles. It only computes input/output dependencies. This is enough for the detection of cycles, because every cycle consists of a direct output-input dependency and an indirect input-output dependency. To find an ordering of the attributes we need all dependencies, so we added the computations of output/input dependencies to the cycle detection. Of course it still detects cycles now, but we can use its information to derive an ordering of the attributes.

#### 3.2 Compute interfaces

Usually this step is preceded by the explicit generation of a total ordering, but we omit this here, and directly generate interfaces. An interface indicates how many times a node is visited, and for each visit which inherited attributes it takes, and which synthesized attributes it returns. We implemented [Pen94], page 124, step 3 directly. This is the implementation in Haskell:

```

type Uses' = [(Name, [Name])]
type Interface = [[Name], [Name]]
makeInterface :: (Uses', Uses') → Interface
makeInterface ([], []) = []
makeInterface (inh, syn) =

```

```

let (inhpart, inh', syn') = work inh syn
    (synpart, syn'', inh'') = work syn' inh'
in (inhpart, synpart) : makeInterface (inh'', syn'')
work :: Uses' → Uses' → ([Name], Uses', Uses')
work [] passive = ([], [], passive)
work active passive
  = let ready = sinks active
    active' = remVertices ready active
    passive' = remVertices ready passive
    in if ready ≡ [] then error "No sinks left "
      else (ready, active', passive')
remVertices :: [Name] → Uses' → Uses'
remVertices _ [] = []
remVertices vs ((v, uas) : us)
  | v ∈ vs = remVertices vs us
  | otherwise = let uas' = filter rem uas
    rem = (λua → ¬ (ua ∈ vs))
    in (v, uas') : remVertices vs us
sinks :: [(v, [u])] → [v]
sinks uses = map fst (filter isSink uses)
where isSink :: (v, [u]) → Bool
    isSink = null ∘ snd

```

### 3.3 Check ordering

The interfaces induce a total ordering of the attributes of each nonterminal. In moving from the partial ordering in the dependencies to this total ordering, we have introduced some extra dependencies, that can lead to circularity. This is called type-3 circularity. We can check for this by using the circularity test of the UU-AG system. If it finds a cycle, we report the dependencies that we have added. Many times this type of dependencies can be prevented by the programmer by adding fake dependencies, and the messages will assist him in that.

If a cycle is found, we cannot give a strict implementation, so we return the old non-visit-oriented implementation.

This feature has not been implemented yet.

### 3.4 Generate visit sequences

After we have found the interfaces, we can compute the visit sequences. For each production of a nonterminal, we describe the behaviour of each visit to

this nonterminal. This consists of a series of evaluations of attributes, and visits to children. We do this in a backwards fashion.

At the end, we need the values of this visit's synthesized attributes, so we add an evaluation of each of them to the visit sequence. We keep a worklist of attributes that we need to compute. Initially this consists of all attributes that this visit's synthesized attributes depend on. Now we inspect the first element of the worklist. This can be:

- An inherited attribute of this visit. We don't have to do anything.
- An inherited attribute of an other visit. We remember that we need its value in this visit, so we need it as an argument.
- A synthesized attribute of a child. We add a visit to this child to the visit sequence, and add the inherited attributes that are needed for this to the worklist.
- An inherited attribute of a child. We add an evaluation of this to the visit sequence.

We keep doing this until the worklist is empty.

The AG code for this can be found in the file `Ordering.ag`, under `Alternative loc.visits`—

### 3.5 Generate semantic functions

From the visit sequences and the derived extra information, we can easily compute the semantic functions. The type of the semantic functions will be of the following form: they take some attributes that were computed in previous visits (and are needed in this visit or in later visits), visit functions of the children (this can be any visit function) and some inherited attributes. They return some synthesized attributes and a continuation function (if there is a next visit).

### 3.6 Include in distribution

Meiage needs to be included in the distribution. This means that it can be switched on and off with a flag when running the UU-AG compiler. This has not been implemented yet.

## 4 Example

### 4.1 Introduction

In this section we give an example of how Meiage works on the Block AG. This example is taken from [Sar99], in which an explanation of this attribute grammar can be found, together with an illustrative picture of the dependencies (Fig. 3.2 of [Sar99]). We only give the UU-AG code here:

```
DATA Its
  | NilIts
  | ConsIts hd : It tl : Its

DATA It
  | Use name : Name
  | Decl name : Name
  | Block its : Its

DATA Root
  | Root its : Its

ATTR It Its [dcli : Env || dclo : Env]
SEM Its
  | NilIts lhs ◦ dclo = @lhs ◦ dcli
  | ConsIts hd ◦ dcli = @lhs ◦ dcli
    tl ◦ dcli      = @hd ◦ dclo
    lhs ◦ dclo    = @tl ◦ dclo

SEM It
  | Use lhs ◦ dclo = @lhs ◦ dcli
  | Decl lhs ◦ dclo = (@name, @lhs ◦ lev) : @lhs ◦ dcli
  | Block lhs ◦ dclo = @lhs ◦ dcli

ATTR Its It [env : Env ||]
SEM Its
  | ConsIts hd ◦ env = @lhs ◦ env
    tl ◦ env      = @lhs ◦ env

SEM It
  | Block its ◦ dcli = @lhs ◦ env
    its ◦ env      = @its ◦ dclo

ATTR Its It [lev : Int ||]
SEM Its
  | ConsIts hd ◦ lev = @lhs ◦ lev
    tl ◦ lev      = @lhs ◦ lev

SEM It
  | Block its ◦ lev = @lhs ◦ lev + 1

ATTR Its It [|| errs : Err]
SEM Its
  | NilIts lhs ◦ errs = []
```

```

    | ConsIts lhs  $\circ$  errs = @hd  $\circ$  errs ++ @tl  $\circ$  errs
SEM It
    | Use lhs  $\circ$  errs = @name 'mBIn' @lhs  $\circ$  env
    | Decl lhs  $\circ$  errs = (@name, @lhs  $\circ$  lev) 'mNBIn' @lhs  $\circ$  dcli
    | Block lhs  $\circ$  errs = @its  $\circ$  errs
ATTR Root [|| errs : Err]
SEM Root
    | Root its  $\circ$  dcli = []
      its  $\circ$  lev = 0
      its  $\circ$  env = @its  $\circ$  dclo
      lhs  $\circ$  errs = @its  $\circ$  errs

{
type Name = String
type Env = [(Name, Int)]
type Err = [Name]
mBIn :: Name  $\rightarrow$  Env  $\rightarrow$  Err
mBIn id [] = [id]
mBIn id ((n, l) : es) | n  $\equiv$  id = []
  | otherwise = id 'mBIn' es
mNBIn :: (Name, Int)  $\rightarrow$  Env  $\rightarrow$  Err
mNBIn t [] = []
mNBIn t@(n, l) (e : es) | t  $\equiv$  e = [n]
  | otherwise = t 'mNBIn' es
}

```

## 4.2 Dependencies

The cycle detection finds the following dependencies (without cycles):

Nonterminal.Attribute	Depends on
It.dclo	dcli,lev
It.errs	env,dcli,lev
Its.dclo	dcli,lev
Its.errs	dcli,env,lev
Root.errs	
It.dcli	
It.env	dclo
It.lev	
Its.dcli	
Its.env	dclo
Its.lev	

### 4.3 Interfaces

The interfaces that are derived are:

Nonterminal	Interface
Root	$[(\[], [\text{lhs. errs}])] -$
Its	$[(\text{[dcli, lev]}, [\text{dclo}]), (\text{[env]}, [\text{errs}])] -$
It	$[(\text{[dcli, lev]}, [\text{dclo}]), (\text{[env]}, [\text{errs}])] -$

### 4.4 Visit sub-sequences

The visit-subsequences that are generated are

Visit	Sub-sequence	Needed attributes
Root.Root - 0	Eval its.lev Eval its.dcli its.dclo = Visit its 0 (its.dcli, its.lev) Eval its.env its.errs = Visit its 1 its.env Eval lhs.errs	
Its.ConsIts - 0	Eval tl.lev Eval hd.lev Eval hd.dcli hd.dclo = Visit hd 0 (hd.dcli, hd.lev) Eval tl.dcli tl.dclo = Visit tl 0 (hd.dcli, hd.lev) Eval lhs.dclo	
Its.ConsIts - 1	Eval tl.env tl.errs = Visit tl 1 tl.env Eval hd.env hd.errs = Visit hd 1 hd.env Eval lhs.errs	lhs.dcli, lhs.lev
Its.NilIts - 0	Eval lhs.dclo	
Its.NilIts - 1	Eval lhs.errs	
It.Decl - 0	Eval lhs.dclo	
It.Decl - 1	Eval lhs.errs	lhs.lev, lhs.dcli
It.Use - 0	Eval lhs.dclo	
It.Use - 1	Eval lhs.errs	lhs.dcli
It.Block - 0	Eval lhs.dclo	
It.Block - 1	Eval its.lev Eval its.dcli its.dclo = Visit its 0 (its.dcli, its.lev) Eval its.env its.errs = Visit its 1 its.env Eval lhs.errs	lhs.dcli, lhs.lev

The only two differences with [Sar99] (page 67) are:

1. If the value of an attribute is not used, it is not evaluated. This can for example be seen in the visits of Uses. The attribute lev is not used, so it is not evaluated. Because we work in a side-effect free setting, this is correct.
2. If the value of an attribute is needed in a certain visit  $n$  and not in a visit before  $n$ , it is computed in visit  $n$ , even if all the attributes that are needed for its evaluation are present in a previous visit. This can for example be seen in the visits of Decl. The value of the errs attribute is needed in the second visit, so it is evaluated in the second visit. The disadvantage of this is that all the attributes from previous visits that are needed for the evaluation of this attribute must be passed to this visit function. Luckily this does not always case more argument-passing (maybe the passed attributes are needed for the evaluation of other attributes, too), and can even decrease the passing of arguments. An advantage of this method is that if a visit is never made, the attributes is not computed at all.

## 4.5 Semantic functions

From the visit sub-sequences we can generate the semantic functions. Below you see what the output will look like. The names of the variables have been simplified to improve readability.

```

type T_Root_0 = Err
type T_Its_0 = Env → Int → (Env, T_Its_1)
type T_Its_1 = Env → Err
type T_It_0 = Env → Int → (Env, T_Its_1)
type T_It_1 = Env → Err
sem_Root_Root :: T_Its_0 → T_Root_0
sem_Root_Root its_0 =
  let its_lev = 0
      its_dcli = []
      (its_dclo, its_1) = its_0 its_dcli its_lev
      its_env = its_dclo
      its_errs = its_1 its_env
      lhs_errs = its_errs
  in lhs_errs
sem_Its_ConsIts :: T_It_0 → T_Its_0 → T_Its_0
sem_Its_ConsIts it_0 its_0 =
  λdcli lev →
    let tl_lev = lev
        hd_lev = lev
        hd_dcli = dcli

```

```

      (hd_dclo, it_1) = it_0 hd_dcli hd_lev
      tl_dcli = hd_dclo
      (tl_dclo, its_1) = its_0 tl_dcli tl_lev
      lhs_dclo = tl_dclo
      sem_Its_1 = sem_Its_ConsIts_1 dcli lev it_1 its_1
    in (lhs_dclo, sem_Its_1)
sem_Its_ConsIts_1 :: Env → Int → T_It_1 → T_Its_1 → T_Its_1
sem_Its_ConsIts_1 dcli lev it_1 its_1 =
  λenv →
    let tl_env = env
        tl_errs = its_1 env
        hd_env = env
        hd_errs = it_1 env
        lhs_errs = hd_errs ++ tl_errs
    in lhs_errs
sem_Its_NilIts :: T_Its_0
sem_Its_NilIts =
  λdcli lev →
    let lhs_dclo = dcli
        sem_Its_1 = sem_Its_NilIts_1
    in (lhs_dclo, sem_Its_1)
sem_Its_NilIts_1 :: T_Its_1
sem_Its_NilIts_1 =
  λenv →
    let lhs_errs = []
    in lhs_errs
sem_It_Decl :: Name → T_It_0
sem_It_Decl name =
  λdcli lev →
    let lhs_dclo = (name, lev) : dcli
        sem_It_2 = sem_It_Decl_1 lev dcli name
    in (lhs_dclo, sem_It_2)
sem_It_Decl_1 :: Int → Env → Name → T_It_1
sem_It_Decl_1 lev dcli name =
  λenv →
    let lhs_errs = (name, lev) 'mNBIn' dcli
    in lhs_errs
sem_It_Use :: Name → T_It_0
sem_It_Use name =
  λdcli lev →
    let lhs_dclo = dcli
        sem_It_1 = sem_It_Use_1 dcli name
    in (lhs_dclo, sem_It_1)
sem_It_Use_1 :: Env → Name → T_It_1

```

```

sem_It_Use_1 dcli name =
  λenv →
    let lhs_errs = name 'mBIn' env
    in lhs_errs
sem_It_Block :: T_Its_0 → T_It_0
sem_It_Block its_0 =
  λdcli lev →
    let lhs_dclo = dcli
    sem_It_1 = sem_It_Block_1 dcli lev its_0
    in (lhs_dclo, sem_It_1)
sem_It_Block_1 :: Env → Int → T_Its_0 → T_It_1
sem_It_Block_1 dcli lev its_0 =
  λenv →
    let its_lev = lev + 1
    its_dcli = env
    (its_dclo, its_1) = its_0 its_dcli its_lev
    its_env = its_dclo
    its_errs = its_1 its_env
    lhs_errs = its_errs
    in lhs_errs

```

## References

- [Kas80] Uwe Kastens. Ordered attributed grammars. *Acta Informatica*, 13(3):229–256, 1980.
- [Pen94] Maarten Pennings. *Generating Incremental Evaluators*. PhD thesis, Utrecht University, <ftp://ftp.cs.ruu.nl/pub/RUU/CS/phdtheses/Pennings/>, November 1994.
- [Sar99] Joao Saraiva. *Purely Functional Implementation of Attribute Grammars*. PhD thesis, Utrecht University, <http://www.cs.uu.nl/~doaitse/Theses/Saraiva.pdf>, 1999.