Contents

UU AG System User Manual

Getting Started

Installing the UUAG system

The easy way: `cabal install uuagc`

Otherwise:

- Have a recent GHC compiler on your system
- Download the UULIB Haskell package
  - From the download page
  - From Hackage
- Unzip and use Cabal to compile and install the UULIB package:

  ```
  > cd <uulib source directory>
  > ghc --make Setup.hs -o setup -package Cabal
  > ./setup configure
  > ./setup build
  > ./setup install
  ```

- Download a source distribution of the UUAG system:
  - From the download page
  - From Hackage
- Unzip and use Cabal to compile and install the UUAG package:

  ```
  > cd <uuagc source directory>
  > ghc --make Setup.hs -o setup -package Cabal
  > ./setup configure
  > ./setup build
  > ./setup install
  ```

Running the UUAG system

We assume that UUAG compiler is installed on your system. If you run the compiler without arguments it will show you a short help message, and a list of options.

```
> uuagc
Usage info:
  uuagc options file ...

List of options:
  -m                        generate default module header
           --module[=name]  generate module header, specify module name
  -d       --data           generate data type definitions
  ...
```

In this user manual all the compiler switches and language features are introduced and explained with examples.

It must be noted that there are two different syntaxes that can be used: the "old" AG syntax and the Haskell like syntax. The latter is preferred for clarity and hence used in this manual. However, in the current version of the compiler the old syntax is default, so to compile the examples the `-H` (or `--haskellsyntax`) option must be used to explicitly enable the Haskell like syntax.

Example 1: Calculate the sum of a tree of integers

Her is a complete UUAG program. It defines binary trees of integers, and how to compute their sum. It also defines a test tree containing some test data, and computes their sum.

```
data Tree
   | Node  left  :: Tree
           right :: Tree
   | Tip   value :: Int

attr Tree
   syn sum :: Int

sem Tree
  | Node  lhs.sum  =  @left.sum + @right.sum
  | Tip   lhs.sum  =  @value

{
main :: IO ()
main = print (show test)

testTree :: Tree
testTree = Node (Tip 1) (Node (Tip 2) (Tip 3))

test :: Int
test = sem_Tree testTree
}
-- output of the program will be "6"
```

The program consists of some UUAG declarations, introduced by the keywords `data`, `attr`, and `sem`. It also contains a plain Haskell part, which is enclosed in braces. All text in braces is not touched by the preprocessor and passed unchanged to the Haskell compiler.

With the `data` declaration we define the syntax of our datatype. A `data` declaration is quite similar to a Haskall `data` declaration, but:

- each field has a name
- all alternatives are preceded with a "|" (even the first one)

With an `attr` declaration we specify attributes, that is the values we want to calculate in our tree walk. We can specify top-down, threaded, and bottom-up attributes, with the constructs `inh`, `chn` and `syn` constructs respectively. Currently only a bottom-up attribute is specified.

With a `sem` declaration we specify the semantics. For each constructor we have an assignment that states how the sum for the left hand side (`lhs`) can be calculated from the fields and/or the attributes of the right hand side. A field can be referred to by an `@` and the field name (as in `@value`), an attribute by selecting from a field with a "dot" notation (as in `@left.sum`). These items can be combined by any Haskell expression (in this case just a `+`).

In the Haskell part we define an example tree structure by using the type `Tree` and the constructors `Node` and `Tip` introduced by the `data` declaration.

A function `sem_Tree` is generated by the preprocessor. It can be used to determine the semantics of our example structure.

The program can be preprocessed, compiled an run by:

```
uuagc -dcfH Example1.ag
ghc Example1.hs
./Example1
```

Example 2: Multiple attributes, and wrappers

The second example program defines four attributes (the sum and maximum of a tree, its front, and a copy of it) and as a test selects one of them to print (the front).

```
data Tree
   | Node  left  :: Tree
           right :: Tree
   | Tip   value :: Int

attr Tree
   syn sum   :: Int
   syn max   :: Int
   syn front :: {[Int]}
   syn copy  :: Tree

sem Tree
  | Tip   lhs.max  =  @value
          lhs.sum  =  @value
  | Node  lhs.sum  =  @left.sum   +   @right.sum
             .max  =  @left.max `max` @right.max

sem Tree
  | Node  lhs.front = @left.front ++ @right.front
          lhs.copy  = Node @left.copy @right.copy
  | Tip   lhs.front = [ @value ]
          lhs.copy  = Tip @value

{
main :: IO ()
main = print (show result)

testTree :: Tree
testTree = Node (Tip 1) (Node (Tip 2) (Tip 3))

test :: T_Tree
test = sem_Tree testTree

result :: [Int]
result = front_Syn_Tree (wrap_Tree test Inh_Tree)
}
-- output of the program will be "[1,2,3]"
```

In the `attr` declaration, we define four bottom-up, or "synthesized" attributes. As of yet, we still don't have top-down, or "inherited" attributes. Types of attributes can be any Haskell type, including the ones generated when preprocessing this this file, but if they are not a simple name they must be escaped by curly braces.

In `sem` declarations we define the values for all four attributes for both constructors. For each constructor we can give multiple assignments at once, but we can also distribute assignments in separate `sem` declarations. Constructors and their assignments may appear in any order. If, for a particular constructor, we have multiple assignments sharing the same target field, we may omit the field name (as is done with `.max` in the `Node` semantics).

In the Haskell part, because we declared four synthesized attributes, calling `sem_Tree` now returns a quadruple. The order of the elements of this quadruple is unspecified. To be able to use it in signatures, the type of the full quadruple is available as `T_Tree` (as an effect of the `-s` option).

As we are interested in a particular element of the quadruple, we run the preprocessor with `-w` option. This will generate a function `wrap_Tree`, which takes:

- the semantics of the tree (which is the quadruple of unknown order)
- an additional argument for initializing inherited attributes (currently just the constant `Inh_Tree`)

and returns

- the four synthesized attributes again, but now as a value of datatype with named fields

From the result of `wrap_Tree`, we can select the desired attribute by name.

Because we need type signatures and wrapper functions to be generated in this example, we add options `-s` and `-w` to the original ones (`-d` for generating datatypes, `-c` and `-f` for generating the "catamorphism" function and the individual semantic functions, and `-H` for Haskell like syntax). So the program can be preprocessed, compiled an run by:

```
uuagc -dcfswH Example2.ag
ghc Example2.hs
./Example2
```

Example 3: Inherited attributes

The third example program defines an attribute that will be a transformed version of the tree. It has the same shape as the original tree, but different values: all values are replaced by a single constant. Again, it is tested on an example tree, in which all values are replaced by 37.

```
data Tree
   | Node  left  :: Tree
           right :: Tree
   | Tip   value :: Int

deriving Tree : Show

attr Tree
   inh replacement :: Int
   syn transformed :: Tree

sem Tree
  | Node  lhs.transformed   = Node @left.transformed @right.transformed
          left.replacement  = @lhs.replacement
          right.replacement = @lhs.replacement
  | Tip   lhs.transformed   = Tip @lhs.replacement


{
main :: IO ()
main = print (show result)

testTree :: Tree
testTree = Node (Tip 1) (Node (Tip 2) (Tip 3))

test :: T_Tree
test = sem_Tree testTree

result :: Tree
result = transformed_Syn_Tree (wrap_Tree test Inh_Tree{replacement_Inh_Tree=37} )
}
-- output of the program will be "Node (Tip 37) (Node (Tip 37) (Tip 37))"
```

As we will use the Haskell `show` function on the `Tree` datatype in the main program, we need to have Haskell `deriving Show` for the `Tree` datatype. In the preprocessor language, this is requested in a separate `deriving` declaration.

In an `attr` declaration, we declare a synthesized attribute *transformed*, that will be a transformed version of the tree. It has the same shape as the original tree, but different values: all values are replaced by a single constant. The value to use as a replacement is information that flows top-down through the tree. For this, we declare an **inherited** attribute *replacement*.

In the `sem` declaration, the definition of synthesized attribute *transformed* is like the *copy* attribute from Example2, except that at a `Tip`, instead of `@value`, the *replacement* value is inserted.

For the inherited attribute *replacement*, the role of the two sides of the assignments are reversed: we may now **use** the value of `@lhs.replacement` at the right side of the equation, but we must **define** the values for the *left* and *right* children.

The inherited attribute needs to be initialized at the top of the tree. This is done in the Haskell part as the second argument to `wrap_Tree`. That argument is a datastructure with named fields for all inherited attributes. The notation with the braces in the definition of `result` is the (not so well-known) Haskell notation for initializing such datastructure values.

Example 4: Multiple datatypes, the self pseudotype, auto-generated rules

This example is a re-work of Example 3, where the value "37" is fixed internally during the tree walk, rather than passing it when calling the tree walk.

For this, we need a non-recursive wrapper-type `Root` around the recursive type `Tree`.

```
data Root
   | Root  tree  :: Tree

data Tree
   | Node  left  :: Tree
           right :: Tree
   | Tip   value :: Int

deriving Root Tree : Show

attr Root Tree
   syn transformed :: self

attr Tree
   inh replacement :: Int

sem Tree
  | Tip   lhs.transformed = Tip @lhs.replacement

sem Root
  | Root  tree.replacement = 37

{
main :: IO ()
main = print (show result)

testTree :: Tree
testTree = Node (Tip 1) (Node (Tip 2) (Tip 3))

testRoot :: Root
testRoot = Root testTree

test :: T_Root
test = sem_Root testRoot

result :: Root
result = transformed_Syn_Root (wrap_Root test Inh_Root)
}
-- output of the program will be "Root (Node (Tip 37) (Node (Tip 37) (Tip 37)))"
```

We can declare attributes for multiple datatypes in a single `attr` declaration. Here we do so for the synthesized attribute *transformed*. For this particular attribute there would be a problem, because a transformed Root has type Root, and a transformed Tree has type Tree. But we can summarize that by using the pseudotype `self`, and still declare the attribute *transformed* in a single declaration.

The inherited attribute *replacement* is only needed in the Tree datatype, so it is declared in a separate `attr` declaration.

An additional effect of using the pseudotype `self` is that semantic rules are defined implicitly, that as a default generate a copy of the datastructure. So we don't have to specify explicitly that a transformed Node consists of the Node constructor applied to the transformed versions of its children, and that a transformed Root is Root re-applied to the transformed version of its content tree.

The only semantic rule we **do** need to write, is where we disagree with the default behaviour: at a Tip, we don't want a copy, but insert the replacement value.

The non-recursive wrapper type Root is a good place to initialize inherited attributes. So in another `sem` declaration the value of *replacement* is set to be 37. The Haskell part is thus freed of the responsibility to initialize the inherited attribute: the non-recursive wrapper type `Root` not even has one. So the call to start the tree walk is as simple as in Example 2.

Another rule `sem` that you might expect is:

```
sem Tree
  | Node  left.replacement  = @lhs.replacement
          right.replacement = @lhs.replacement
```

This is not needed though, because rules that copy inherited attributes unchanged to the children are inserted automatically when nothing other is specified. This defaulting mechanism is known as (one of the forms of) the **copy rule**.

Example 5: Two-pass traversals

This example is another variation on Example 4, where again we generate a tree of the same shape as the original one, with values replaced by a constant. The constant however is not taken to be 37, but the sum of all values in the original tree.

Appearantly, this would require a two-pass traversal of the tree: one pass that determines the sum, and another one to construct the copy-with-replaced-values. But the program does not really look very different than Example 4:

```
data Root
   | Root  tree  :: Tree

data Tree
   | Node  left  :: Tree
           right :: Tree
   | Tip   value :: Int

set Every = Root Tree

deriving Every : Show

attr Every
   syn transformed :: self
```

```
attr Tree
   inh replacement :: Int
   syn sum         :: Int

sem Tree
  | Tip   lhs.transformed = Tip @lhs.replacement
          lhs.sum         = @value
  | Node  lhs.sum         = @left.sum + @right.sum

sem Root
  | Root  tree.replacement = @tree.sum

{
main :: IO ()
main = print (show result)

testRoot :: Root
testRoot = Root (Node (Tip 1) (Node (Tip 2) (Tip 3)))

test :: T_Root
test = sem_Root testRoot

result :: Root
result = transformed_Syn_Root (wrap_Root test Inh_Root)
}
-- output of the program will be "Node (Tip 6) (Node (Tip 6) (Tip 6))"
```

As an added convenience, we introduce the `set` declaration in this example. A `set` declaration is a way to abbreviate sets of datatypes. The name can be used instead of an explicit enumeration of the set in other (`deriving`, `attr` and `sem`) declarations.

In the `attr` declarations, we re-introduce synthesized attribute *sum* here for datatype Tree (see Example1). Notice that Tree has two synthesized attributes:

- `sum: Int` (which is obviously declared)
- `transformed: Tree` (because `Tree` is part of the `Every` set, and `self` is instantiated to `Tree`)

In the `sem` declaration for `Root` in Example4 we initialized the replacement value with 37. But nothing prevents us from letting the replacement value depend on other synthesized attributes, such as *sum*.

In effect, we now have defined a two-pass tree traversal. In the first pass, the *sum* of all values is calculated. In the second pass, the *transformed* tree is build, using the *sum* as the *replacement* value. combined, we have synthesized a tree of the same shape as the original, where every value is replaced by the sum of all values. The nice thing is that you need hardly be aware that this is actually a two-pass traversal. You can freely use synthesized attributes to define inherited attributes, that may in turn be needed to define other synthesized attributes. Your only concern should only be that the dependency is not circular, but whether one, two, or six passes are necessary is automatically determined.

Building AG files with Cabal

The UUAG system consists of the preprocessor `uuagc` and a Cabal plugin `uuagc-cabal`. The UUAG preprocessor takes several UUAG files (.ag) and produces a Haskell file (.hs). Instead of invoking the preprocessor manually as we described above, we can also let Cabal take care of that through the Cabal plugin. Take the following steps.

In the `Setup.hs`, add an import declaration for the cabal plugin and use the special user-hook:

```
import Distribution.Simple (defaultMainWithHooks)
import Distribution.Simple.UUAGC (uuagcLibUserHook)
import UU.UUAGC (uuagc)

main = defaultMainWithHooks (uuagcLibUserHook uuagc)
```

Furthermore, add in the same directory a file called `uuagc_options`, which must specify for each AG file, with what features the corresponding Haskell file is to be build:

```
file : "src/MyModule.ag"
options : data, catas, semfuns, signatures, pretty, rename, module "MyModule", wrappers
```

Cabal then keeps track of dependencies between files, and ensures that files are rebuild after changes.

Language Constructs

This section gives an overview of the UUAG language. The following sections show the syntax of each construct by example. A complete reference in EBNF of the UUAG language can be found in the appendix.

data declaration

A `data` declares a nonterminal, and a number of productions for a nonterminal. Each production is labelled with a constructor name. Constructors must be unique within a nonterminal, but (unlike in Haskell) the same constructor name is allowed in other nonterminals. Giving multiple `data` declarations for the same nonterminal is allowed, provided that the constructor names in the declarations do not clash. The fields of each production all have a name and a type. The type can be a nonterminal or a Haskell type. All fields of the same constructor must have different names.

Valid `data` declarations:

```
data Tree | Bin  left :: Tree  right :: Tree
          | Leaf value :: Int

data Decl | Fun  name :: String  args :: {[String]}  body :: Expr
```

Several abbreviations exist for `data` declarations. Fields with the same type can be declared by listing their names separated by commas. Also the field name can be left out, in which case the name is defaulted to the type name with the first letter converted to lowercase. It is only allowed to leave out the field name if the type is an uppercase type identifier. You also need to make sure that the default name does not clash with the name of another field. The following example show correct abbreviations:

```
data Tree | Bin  left,right :: Tree   -- two fields with the same type
          | Leaf               Int    -- field name implicitly is 'int'
```

The following `data` declaration is wrong:

```
data Tree | Bin Tree Tree    -- two nameless fields have the same type, so would have the same name
          | Leaf {(Int,Int)} -- type of nameless field is not a simple name, so no name can be deduced for
```

Polymorphic abstract syntax is allowed by adding type variables to `data` declarations. The variables are required to be lower-case identifiers. The type variables are only usable somewhere inside a Haskell type (hence the curly braces). Use parentheses around nonterminals to be able to pass a list of Haskell types to the nonterminal. The following example shows a binary tree storing values of type *a* or *b* depending on the depth of the leaf:

```
data Eq {a}, Show {b} => Tree a b
  | Bin left,right :: (Tree {b} {a})
  | Leaf value :: {a}
```

If a nonterminal is defined using several `data` declarations, the list of type variables is the concatenation of the type variables of the individual declarations in the order of appearance.

These `data` declarations define the structure of the Abstract Syntax Tree. An AST can be constructed for a nonterminal `D` (data type) using alternatives `C` (constructor) in Haskell by applying some arguments to the constructor `D_C`, returning a value of type `D` when all arguments are applied. Alternatively, a deforested AST (the denotation of the AST) can be constructed by using the functions `sem_D_C` instead of the constructor `D_C`, returning a value of type `T_D` when all arguments are applied.

attr declaration

An `attr` declaration declares attributes for one or more nonterminals. Each attribute is inherited, chained, or synthesized and has a name and a type. A chained attribute is just an abbreviation for an attribute that is both inherited and synthesized. The names of all inherited attributes declared by `attr` statements must be different. The same holds for synthesized attributes.

Valid `attr` declarations are:

```
attr Tree
   inh depth   :: Int
   chn minimum :: Int
   syn out     :: {[Bool]}
attr Tree
   inh count :: Int
   syn count :: Int
attr Decl inh environment :: {[(String,Type)]}
attr Decl syn code :: Instructions
```

In attribute declarations the abbreviations for fields in a `data` declaration are not allowed, so all attributes must have an explicit name and they can not be grouped.

A `use` clause can be added to the declaration of a synthesized or chained attribute, to trigger a special kind of copy rule. The first expression must be an operator, and the second expression is a default value for the attribute. For example:

```
data Tree
   | Bin left,right :: Tree
   | Leaf value :: Int
attr Tree
   syn value use {+} {0} :: Int -- compute sum of values
```

An attribute can be declared to be of type `self`. The type `self` is a placeholder for the type of the nonterminal for which the attribute is declared. For example:

```
attr Tree Expr
   syn copy :: self
```

The `attr` statement above declares an attribute *copy* of type *Tree* for nonterminal *Tree*, and an attribute *copy* of type *Expr* for nonterminal *Expr*. Declaring a synthesized attribute of type `self` triggers a special copy-rule, that constructs a copy of the tree.

To refer to type variables of polymorphic nonterminals in the types of attributes, use the name of the type variable prefixed with an *@*. Type variables only need to be prefixed when they occur in the type of an attribute. The type of the attribute has to be a Haskell type (in curly braces). For example, to declare a synthesized attribute that is a list of all the values in the tree:

```
data Tree a | Bin left,right :: (Tree {a})
            | Leaf value :: {a}
attr Tree
   syn elements :: {[@a]}
```

Attribute declarations can also be given in `data` or `sem` statements after the name of the nonterminal. For example:

```
data Tree | Bin left,right :: Tree
          | Leaf Int
attr Tree
   syn min :: Int
```

can be combined into:

```
data Tree syn min :: Int
   | Bin left,right :: Tree
   | Leaf Int
```

sem

In a sem construct one can specify semantic rules for attributes. For each production the synthesized attributes associated with its corresponding nonterminal and the inherited attributes of its children must be defined. If there is a rule for a certain attribute is missing, the system tries to derive a so called copy-rule.

Semantic rules are organised per production. Semantic rules for the same production can be spread over multiple sem statements. This has the same meaning as they were defined in a single sem statement.

A *fieldref* is lhs or loc or a field name. To refer to a synthesized attribute of the nonterminal associated with a production the special fieldref lhs is used together with the name of the attribute. To refer to an inherited attribute of a child of a production the field name of the child is used together with the attribute's name. The special fieldref loc is used to define a variable that is local to the production. It is in the scope of all semantic rules for the production.

The expressions in semantic rules are code blocks, i.e. Haskell expressions enclosed by { and }. They may contains references to values of attributes and fields. These references are all prefixed with an @ symbol to distinguish them from Haskell identifiers. To refer to the value of a field one uses the name of the field. References to attributes are similar to the ones on the left-hand side of a semantic rule such as field.attr. The difference is that they now refer to the synthesized attributes of the children and the inherited attributes of the nonterminal associated with the production. Local variables can be referenced using their name, optionally prefixed with the special loc field.

Valid definitions:

```
attr Tree
   inh gmin   :: Int
   syn min    :: Int
   syn result :: Tree
sem Tree
   | Bin  left.gmin  = { @lhs.gmin }
     -- "left.gmin" refers to the inherited attribute "gmin"
     -- of the child "left"
   | Bin  right.gmin = { @lhs.gmin }
     -- "@lhs.gmin" refers to the inherited attribute "gmin"
     -- of nonterminal "Tree"
   | Bin  loc.min    = { min @left.min @right.min }
     -- "min" is a new local variable of the constructor "Bin"

sem Tree
  | Bin  lhs.result = { Bin @left.result @right.result }
    -- "@left.result" refers to the synthesized attribute "result"
    -- of child "left"
  | Bin  lhs.min    = { @min }
    -- "@min" refers to the local variable "min"
  | Leaf lhs.result = { Leaf @lhs.gmin }
    -- "@lhs.gmin" refers to the inherited attribute "gmin"
    -- of nonterminal "Tree"
  | Leaf lhs.min    = { @int }
    -- "@int" refers to the value of field "int" of "Leaf"
```

For the sem construct there exist a number of abbreviations. As for data statements one can write attribute declarations after the name of the nonterminal. Furthermore semantic rules for the same production can be grouped, mentioning the name of the production only once. For example:

```
sem Tree
   | Bin  left.gmin  = { @lhs.gmin }
          right.gmin = { @lhs.gmin }
          loc.min    = { min @left.min @right.min }
```

In a similar way semantic rules for the same fieldref can be grouped. For example:

```
sem Tree
  | Bin  lhs.result = { Bin @left.result @right.result }
            .min    = { @min }
```

When the same semantic rule is defined for two productions of the same nonterminal they can be combined by writing the names of both productions in front of the rule. For example:

```
sem Tree
  | Node1 lhs.value = { @left.value + @right.value }
  | Node2 lhs.value = { @left.value + @right.value }
```

can be abbreviated as follows:

```
sem Tree
  | Node1 Node2 lhs.value = { @left.value + @right.value }
```

Finally the curly braces around expressions may be left out. The layout of the code is then used to determine the end of the expression as follows. The column of the first non-whitespace symbol after the '=' symbol is the reference column. All subsequent lines that are indented the same or further to the right are considered to be part of the expression. The expression ends when a line is indented less than the reference column. An advantage of using layout is that problems with unbalanced braces are avoided.

When using polymorphic abstract syntax, the type variables often need to be constrained by the Ord class in order to put values of this type in sets or maps. Add context using:

```
data Tree a
sem Ord {a}, Show {a} => Tree
```

It is also possible to add such a context at `data` or `attr` declarations.

A rule introduced by a `sem` introduces dependencies between attributes. It is sometimes desirable to add additional dependencies between attributes. For example, when using the `-O` option to derive a static multi-visit rule ordering, scheduling of attributes is done greedily, with as much attributes per visit as possible. By adding additional dependencies, we end up with smaller visits.

To add an additional dependency that fld1.at1 is evaluated strictly before fld2.at2, write:

```
sem D
  | C  fld1.at1 < fld2.at2
```

You may use local attributes as well.

There are two meta-fields called `first__` and `last__`. When using `@first__.x`, the first field of the alternative is substituted that has a synthesized attribute `x`. Similarly, with `@last__.x` the last field with a synthesized attribute `x` is selected. At the moment, these meta-fields can only be used in right-hand sides.

There is special notation for higher-order attributes (also called nonterminal attributes). A higher-order attribute is a local attribute that acts as if it is an additional child of the production. It has either type `D` or type `T_D` (where `D` is some AST declared by a `data` declaration). For example:

```
data D | C Int
data R | R
sem R
  | R  inst.x :: D
       inst.x = C 3
```

Now, it is as if `x` is a child of `R`, although the value is defined by a rule instead of obtained from the constructor. The type signature is obligatory. You can also use the "deforested" types (T_D), and build the deforested tree yourself (using e.g. sem_C). Please note that nonterminal-names should not start with a "T_".

You need to supply inherited attributes of `D` and can use synthesized attributes of `D`:

```
attr D
   inh p :: Int
   syn q :: Int
sem R
  | R  x.p = 3
       loc.z = @x.q + 1
```

Like normal children, higher-order attributes participate with copy rules. Higher-order attributes are considered as children after the normal children of a constructor, in the order of lexical appearance.

If a higher-order attribute shadows an existing child, its definition is actually a function from the semantics of the original child to the new semantics:

```
data D1 | C1   k :: D1
data D2 | C2

sem D1 | C1  inst.k :: D2
             inst.k = \semD1 -> sem_D2_C2 ...
```

Sometimes it is needed to specify type signatures for local attributes:

```
sem D
  | C     loc.x :: {type}
```

type

The `type` construct is convenient notation for defining list based types. Apart from a convenient notation the `type` construct has effect on the code generated. Instead of generating data constructors *Cons* and *Nil* Haskell's list constructors `:` and `[]` are used.

Examples of `type` constructs:

```
type IntList = [ Int ]
type Trees   = [ Tree ]
```

The following other type synonyms are supported:

```
type MbInt = maybe Int
type EIntBool = either Int Bool
type Trip = (Int, Bool, Char)
type M1 = map {String} D   -- curly braces required (key must be a haskell type)
type M2 = intmap D
```

Nonterminals are also allowed as argument:

```
type MbExpr = maybe Expr
attr MbExpr Expr
   syn tp :: Type

sem MbExpr
  | Just     lhs.tp = @just.tp
  | Nothing  lhs.tp = undefined
```

- A list has productions: `Cons` with field `hd` and field `tl`, `Nil` without any fields.
- A `maybe` has productions: `Just` with field `just`, `Nothing` without any fields.
- An `either` has productions: `Left` with field `left`, `Right` with field `right`.
- A `map` has productions: `Entry` with field `key`, `val`, and `tl`, `Nil` without any fields.

- A tuple has productions: `Tuple` with fields `x1`, `x2`, etc..

include

Other UUAG files can be included using the following construct:

```
include "filename.ag"
```

The filename must appear in double quotes. The suffix of the file `.ag` or `.lag`) should not be omitted. The file should contain valid UUAG declarations. These statements are inlined in the place of the `include` directive.

Code Block

A code block is a piece of Haskell code enclosed by curly braces. There exist three kinds of code blocks: top-level, type, and expression code blocks. A top-level code block contains Haskell declarations, such as `import` declarations, and function and type definitions. A name can be writen before a top-level code block. The code blocks are sorted by their names, and appended to the code generated by the UUAG system. A special name `imports` is used to mark code blocks containing `import` declarations. These are copied to the start of the generated code, as Haskell only allows `import` declarations at the beginning of a file.

An example of two code blocks, an import declaration and a function definition:

```
imports
{
import List
}

quicksort
{
-- simple implementation of quicksort:
qsort :: Ord a => [a] -> [a]
qsort [] = []
qsort (x:xs) = let (l,r) = partition (<=x) xs
               in qsort l ++ [x] ++ qsort r
}
```

A type code block contains a Haskell type and may be used as type in `data`, `type`, and `attr` declarations. Examples:

```
data Module
   | Module name :: {Maybe String} body :: Declarations

type Points = [ {(Int,Int)} ]

attr Expr
   inh env :: {[(String,Int)]}
```

Finally expression code blocks contain a Haskell expression and occur as the right-hand side of attribute definitions in `sem` statements. Apart from normal Haskell code they may contain references to attributes. These references are prefixed with an `@` symbol, to distinguish them from ordinary Haskell identifiers. Examples:

```
sem Tree syn min :: {Int}
  | Node lhs.min = { min @left.min @right.min } -- an expression code block
```

The contents of a block is the plain text between an open and a close brace. The text in a code block is not interpreted by the UUAG system. Curly braces occurring inside the Haskell code must be balanced. This includes curly braces in comments, and in string and character literals.

An example of a code block containing a nested pair of braces:

```
{
f a b c = let { d = b*b - 4*a*c
              ; result1 = (-b + sqrt d) / 2*a
              ; result2 = (-b - sqrt d) / 2*a
              ; result  | d >  0 = [result1, result2]
                        | d == 0 = [result1]
                        | d < 0  = []
              }
          in result
}
```

All curly braces Haskell constructs, such as `do` and `let` are normally matched. However, curly braces in string, or character literals may cause problems. The balancing rule forbids code blocks such as:

```
{
openbrace = "{"
}
```

This problem can be fixed by inserting a matching brace in comments. In the following code the curly braces are balanced:

```
{
openbrace = "{"
-- }, just to balance braces
}
```

Comments

One-line comments start with two dashes (`--`) and end at the end of the line. Multi-line comments start with `{-` and end with `-}`. As in Haskell comments can be nested.

```
{-
Definition of a datatype for binary trees
-}
data Tree
   | Leaf val :: Int
   | Node left :: Tree right :: Tree -- a node has two subtrees
```

Names

Names start with a letter followed by a (possibly empty) sequence of letters, digits, underscores and prime signs. A name for a nonterminal or constructor must start with an upper-case letter. A name of a field or attribute must start with a lower-case letter. The following words are reserved and cannot be used as names: `data`, `ext`, `attr`, `sem`, `type`, `use`, `loc`, `lhs`, and `include`.

Valid names:

```
-- nonterminals or constructors:
Node
Expression
Tree_Node
-- field names or attributes:
left
long_name
field2
```

Multi-productions

Sometimes you want to have multiple productions for an AST node. This situation arises when an AST node represents some form of choice, typically for some form of proof search. For example for the implementation of type systems with type rules that are not syntax-directed, or interpreters of an operational semantics with evaluation rules that are not syntax-directed. Alternatively, this may also arise when the grammar is ambiguous and the AG evaluation should disambiguate. Here, we show how this can be done with AGs.

The above situation is the case for the following proposition language. We first explain this language, then look at multiple evaluation-productions for the `Or` node. The abstract syntax is:

```
data Prop
  | Con  bool  :: Bool             -- constants |True| and |False|
  | Emb  fun   :: {Env -> Prop}    -- embed Haskell function
  | Let  nm    :: String           -- binding a (non-recursive)
         expr  :: Prop             -- name to a prop
         body  :: Prop
  | And  left  :: Prop             -- the logical and of two props
         right :: Prop
  | Or   left  :: Prop             -- the logical or of two props
         right :: Prop

type Env = map String Bool
```

An `Emb` node is semantically equal to `fun` applied to the environment (introduced later). Its actual purpose is that the technique presented here also works for higher-order children, as we see later. For now, it is a fancy way to define the dereference of a value in the environment. For example, via the smart-constructor `var`:

```
var :: String -> Prop
var nm = Prop_Emb (Prop_Con . findWithDefault (error "not found") nm)
```

Given an initial environment [("f", False), ("t", True)], the following is an example proposition equal to `False`:

```
tree = Prop_Or (Prop_And big $ var "f") (var "f")
big  = Prop_And (var "t") big
```

In this example `big` is an infinite proposition. Its value does not affect the outcome, because of the `Prop_And` with `var "f"`.

Consider a syntax-directed set of evaluation rules for `Prop` written as an AG. For that, we add an inherited attribute `env` (the environment) and a synthesized attribute `outcome` (denotes the boolean value of the proposition).

```
attr Prop
   inh env :: Env
   syn outcome :: Bool

sem Prop
  | Con  lhs.outcome = @bool
  | Emb  inst.k :: Prop
         inst.k = @fun @lhs.env
  | Let  body.env    = insert @nm @expr.outcome @lhs.env
  | And  lhs.outcome = @left.outcome && @right.outcome
  | Or   lhs.outcome = @left.outcome || @right.outcome
```

For an `Emb` prop, the outcome is determined by the computed prop `k`. For a `Let` prop, we insert in the environment of the `body`, the outcome of the `expr` bound to `nm`. Copy rules take care of the rest.

When we run the above program on the `tree`, it does not produce a value. The problem is causes by both the "&&"-operator and the "||"-operator. They do not distribute the evaluation over their operands well. They are strict in the first argument, therefore the left-operand is explored first. We would like a more balanced evaluation.

Take a closer look at the productions for `And` and `Or`: these can actually be split in four conditional productions:

```
sem Prop
  | And1  lhs.outcome = False           (if @left.outcome=False or @right.outcome=False)
  | And2  lhs.outcome = True            (otherwise)
  | Or1   lhs.outcome = @left.outcome   (if @right.outcome=False)
  | Or2   lhs.outcome = @right.outcome  (if @left.outcome=False)
```

If we could write our AG this way, a clever AG evaluator could use some backtracking-technique combined with an attempt to reduce the AST nodes that received the least amount of evaluation so far. However, this may not always be a good strategy. For example, when both `@left.outcome` and `@right.outcome` are `False`, the choice between `Or1` and `Or2` is ambiguous, and we may prefer one over the other. Also, in some cases we may want to be biassed in a particular subtree, potentially based results computed at runtime. Hence: we would like to define this strategy ourselves, at runtime.

Since the problem for the `And` props is similar to that of the `Or` props, we forget about the former and consider only the latter. We do not really want to express multiple productions for an AST node, because (as mentioned above) the AG evaluator should not magically make the choice for us. However, we can encode the multiple productions as a single production where a choice is made between children via a function `best`:

```
sem Prop
  | Or  merge left right as result :: Prop = best
```

Read `merge` as 'choose' here: `best` indicates which of the two children `left` and `right` is chosen (explained later). The chosen child is provided as a child named `result` (hence, of type `Prop`, which has to be explicitly written). Note that as a consequence, no rule may refer to the the synthesized attributes of `left` and `right`, and neither define the inherited attributes of `result`.

The function `best` gets the synthesized results of `left` and `right` as parameter, and must provide the synthesized results for `result`. The type of `best` is thus `T_Prop -> T_Prop -> T_Prop`. Since `Prop` has only once synthesized attribute `outcome` of type `Bool`, `best` is actually a function of type `Bool -> Bool -> Bool`. For example, when `best = const`, it always chooses the `left` child. For our example, as initial attempt, we define `best` as follows:

```
best left_outcome right_outcome
  | not right_outcome = left_outcome
  | not left_outcome  = right_outcome
```

However: we now actually made the problem worse! We now effectively evaluate both children always. We can do slightly better with the following definition as second attempt:

```
best left_outcome right_outcome
  | left_outcome  = left_outcome
  | right_outcome = right_outcome
  | otherwise     = False  -- may also take left_outcome/right_outcome
```

Actually, we are now back at the beginning: to make a choice, we inspect the outcome of the left-child, thus the evaluation is still left-biased.

The problem is that the `best` as defined above makes a choice based on the final results of the computation, which is too late. What we really would like to do is to let every child return some form of progress report that gives insight in the amount of effort evaluation took so far, and base our choice on that. We can achieve this in a various ways. For instance, via a combination of lazy evaluation with an additional attribute. However, we do not discuss the entire design space here. Instead we present immediately our solution, which provides a breadth-first evaluation with online results. To enable this behavior, provide an additional flag `--breadthfirst` to `uuagc`, and import the Haskell module `import Control.Monad.Stepwise.AG` (package `stepwise` on Hackage).

The 'progress report' for a child is a stream of `EvalInfo` (user-defined) entries, that ends with special entry containing the values of its synthesized attributes. For our example, we just want some estimation of the amount of work that is done for a child, so each progress-entry is just a value that denotes a bit of work:

```
data EvalInfo = Work
```

These entries need to be generated somewhere, and we act on them in the `best` function.

To start with the latter: the signature of `best` changes to: `Comp EvalInfo I_T_Prop -> Comp EvalInfo I_T_Prop -> Comp EvalInfo I_T_Prop`. It takes a computation `Comp` for each child, and must return a computation for the choice between these types. The type `Comp` is provided by the imported module, as well as the function `oneStep`:

```
oneStep :: Comp EvalInfo t -> Outcome EvalInfo t

data Outcome i t
  = Step  i (Comp i t)
  | Done   (Syn t)
```

It evaluates the computation until it either emits a progress report (returned via outcome `Step`, including the residual computation), or the computation is finished and returns the synthesized attributes of the child (denoted as `Syn t`, i.e. the type `Bool`).

In `best`, we distribute work over the two children equally. We emit a Work-report if both children did a bit of work, and lazily choose between the residual computations. Otherwise, if one of the children finishes, we commit ourselves to one of them:

```
best l_comp r_comp = best' (oneStep l_comp) (oneStep r_comp)
  where  best' (Step Work l_next) (Step Work r_next)  -- both do a step
            = Step Work (best l_next r_next)
         best' (Done True) _  = l_comp  -- one of them fails or succeeds
         best' _ (Done True)  = r_comp
         best' (Done False) _ = r_comp
         best' _ (Done False) = l_comp
```

If you (lazily) want the synthesized results of a child, apply the function `lazyEval :: Comp EvalInfo t -> Syn t` to its computation. This is not the same as running `oneStep` until it emits a `Fin`, because the latter uses a strict evaluation scheme.

Finally, how to emit these `Work` steps. We can use the `merge` syntax introduced above. Suppose that we consider reaching an `Or` node to be abit of evaluation, we could emit a `Work` report before doing the comparisons:

```
sem Prop
  | Or  merge left right as result :: Prop = \l r -> info Work (best l r)
```

The function `info :: EvalInfo -> Comp EvalInfo t -> Comp EvalInfo t` is provided by the imported module and emits the `Work` progress report before the progress reports resulting from `best l r`.

What if we also want to emit some progress for e.g. the constant leafs, where there is no `merge` to piggy-back on? Then we simply introduce one:

```
sem Prop
  | Con  merge dummy :: Dummy = info Work $ final ()
         dummy.handle < lhs.outcome

data Dummy | Dummy
attr Dummy syn handle :: ()
```

`merge` takes a list of children as inputs, which may be empty. We must still provide a computation for `Dummy`, which we can do using the function `final`, which wraps values for synthesized attributes into a trivial computation. Finally, we need to take care that the outcome of the dummy-child is actually needed at some point, otherwise the progress report is never emited, which is the purpose of the 'fake dependency' handle-before-outcome.

The mechanism of injecting progress reports is flexible, because we can precisely pinpoint when to inject these reports. There are some improvements possible here though.

With the code structered in this way, we effectively compute the value of a proposition in a breadth-first way. The example presented here shows an example with inherited and synthesized attributes, a higher-order child (`Emb`), an inherited attr for a child that depends on the syn attr of another child (`Let`). In fact, all features required by our greatest user, the compiler UHC, are supported. The administration needed to represent the breadth-first computation takes additional memory and cpu time. Early benchmarks against the UHC compiler seem to suggest that the overhead is negglible when the mechanism is not used.

The implementation in UUAG has some practical limitations. Some of them can be eliminated through additional effort. The limitations are:

- Works in combination with many of the commonly-used commandline options, but likely breaks with some rarily used options.
- In case of a multi-visit AG, the choice is made for the first visit. Only those attributes are available.
- The type of the result child of a merge must equal the type of the input children.
- There can only be one type of progress report, hardcoded to be `EvalInfo`. It must be in scope of the generated Haskell module.
- The AG must be ordered (pass the -O flag).

Auto-generated rules

When a definition for an attribute is missing, the UUAG can often derive a rule for it. These automatic rules, also known as copy rules, are based on name equality of attributes. They save a lot of otherwise trivial typing, thus making your programs easier to read by just leaving the essential parts in the code. If in the list of rules for a constructor a rule for an attribute *attr1* is missing then UUAG system tries to derive a rule for this attribute. This is done by looking for an attribute *attr2* with the same name as *attr1* in the sets of synthesized attributes of the children of the constructor and in the set of inherited attributes of the nonterminal it belongs to. If such an attribute *attr2* is found then the value of *attr1* is set to the value of *attr2*. This section first shows two examples and then defines a generalisation that captures both(and others). There are also two special copy rules, the `use`, and `self` rules, which are explained at the end of this section.

The copy rule

Very often one needs to pass a value from a node to all its children. Consider for example the following code, in which a inherited attribute *gmin* is declared.

```
data Tree | Bin left,right :: Tree
          | Leaf val :: Int

attr Tree
   inh gmin :: Int
```

In this example rules for the syntesized attribute *gmin* of children of the constructor *Bin* are missing. This is however no problem. The nonterminal *Tree* has an inherited attribute with the same name and the UUAG system automatically inserts the following rules:

```
sem Tree
  | Bin left.gmin  = @lhs.gmin
        right.gmin = @lhs.gmin
```

This kind of copy-rule is very convenient for copying an inherited attribute to all nodes in a top-down fashion.

Another kind of copy-rule is a co-called chain-rule. For a chain rule an attribute that is both inherited as well as synthesized is chained from left to right through all children of a constructor. Consider for example the following code that numbers all leaves in a *Tree* from left to right.

```
attr Tree
   chn label :: Int

sem Tree
  | Leaf lhs.label = @lhs.label+1
```

Because the attribute *label* is declared inherited as well as synthesized the UUAG system derives the following rules for the constructor *Bin*:

```
sem Tree
  | Bin left.label  = @lhs.label
```

```
        right.label = @left.label
        lhs.label   = @right.label
```

Generalised copy rule

The UUAG system implements a more general copy rule of which the examples above are instances. If a rule is missing for an inherited attribute *n* of a child *c*, the UUAG system searches for an attribute with the same name(*n*). The UUAG system searches for a suitable candidate in the following lists:

1. local attributes
2. synthesized attributes of children on the left of *c*
3. inherited attributes
4. fields

The search takes place in the order defined above, and the first occurrence of *n* is copied. Thus local attributes have preference over others. When there are multiple occurrences of *n* in the list of synthesized attributes of the children the rightmost is taken.

When a rule for a synthesized attribute is missing the search for a candidate with the same name takes place in a similar fashion. In the second step all children are searched, again taking the rightmost candidate if more than one is found.

`use` rules

A `use` rule can be derived for a synthesized attribute whose declaration includes a `use` clause. A `use` clause consists of two expressions; the first is an operator, and the second is a default value. Suppose *s* is a synthesized attribute of *n*, that is declared with a `use` clause. If for a constructor *c* of *n* a definition of *s* is missing, a rule is derived as follows. Collect all synthesized attributes of constructor *c* 's children with the same name as *s*. If this collection is empty the default value declared in the `use` clause is taken. If this collection contains only a single attribute, then the value of this attribute is copied. Otherwise the values of the attributes are combined using the operator and the result is used to define *s*.

For example:

```
data Tree
   | Bin left,right :: Tree
   | Single val :: Int
   | Empty

attr Tree
   syn sum use {+} {0} :: Int
sem Tree
  | Single lhs.sum = @val
```

The UUAG system derives the following rules:

```
sem Tree
  | Bin   lhs.sum = @left.sum + @right
  | Empty lhs.sum = 0
```

`self` rules

The type `self` in an attribute declaration is equivalent to the type of the nonterminal to which the attribute belongs. A synthesized `self` attribute can for example be used if one wants a local copy of a tree, or wants to transform it. The `self` attribute then holds the transformed version of the tree. A `self` attribute usually holds a copy of the tree, except for a few places where a transformation is done. The semantic rules required for constructing a copy of a tree call for each production the corresponding constructor function on the copies of the children. The UUAG system implements a special copy rule to avoid writing these trivial rules. For each production of a nonterminal with a synthesized **self** attribute *n*, the UUAG system generates a local attribute containing the application of the corresponding constructor to the **self** attributes of the children with the same name as *n*. The value of the synthesized attribute is set to this local attribute.

For example for:

```
data Tree
   | Bin left,right :: Tree
   | Leaf val :: Int

attr Tree
   syn copy :: self
```

the following semantic rules are generated:

```
sem Tree
  | Bin  loc.copy = Bin @left.copy @right.copy
         lhs.copy = @copy
  | Leaf loc.copy = Leaf @val
         lhs.copy = @copy
```

The default definitions for the local and sythesized `self` attributes can be overriden by the programmer.

The following program is a complete attribute grammer for the Repmin problem using as many copy rules as possible. For constructing the transformed a `self` attribute *result* is used. Note that only for the production *Leaf* an explicit definition of this attribute is given. The definition for *Bin* is provided by an automatic rule.

```
data Tree
   | Bin left,right :: Tree
   | Leaf val :: Int

data Root
   | Root Tree

attr Tree
   inh gmin :: Int
   syn lmin use {`min`} {0} :: Int
attr Root Tree
   syn result :: self

sem Tree
  | Leaf lhs.lmin   = @val
            .result = Leaf @lhs.gmin
sem Root
  | Root tree.gmin = @tree.lmin
```

Unique numbering

We have special syntax for obtaining unique "numbers" (doesn't have to be a number strictly speaking, as long as it has an identity) at various places at the abstract syntax tree. The following miniature tutorial explains how to use this special syntax.

Prerequisite: a threaded counter:

```
attr Tree
   chn counter :: Int
```

Can be of any type that you want, with any name that you want, and you can pass it through the tree in whatever way you want, except that it *MUST* be a *CHAINED* attribute at those nonterminals whose constructors use the special syntax below.

Usage: using special type signatures:

```
sem Tree
  | Node
     loc.num1 :: uniqueref counter
     loc.num2 :: uniqueref counter
```

Now you have two local attributes that will have a unique number (in the identity-space of counter, and with the same type as counter). So, you can then just use the local attributes:

```
sem Tree
 | Node
     loc.ids = [ @loc.num1, @loc.num2 ]
```

Is this all? No, there is yet one prerequisite. You must define a function `nextUnique`, which gets a value with the same type as the counter, and returns a tuple containing the next counter and one unique number. For example, for `Int`:

```
nextUnique :: Int -> (Int, Int)
nextUnique u = (u+1, u)
```

This function must be in scope of the semantic functions. The type does not have to be `Int`, it can be something more complicated. The name of this function can be changed using the `--uniquedispenser=funname` commandline option.

Rules for defining these local numbers are automatically generated. For the above example, a sequence of `nextUnique` calls will be generated to obtain the unique numbers for a node in the abstract syntax three, starting with `@lhs.counter`, and subsequently the result of the previous call. After obtaining the numbers, the counter is passed to the first child which has the `counter` atttribute, or `@lhs.counter` otherwise.

Appending to a synthesized attribute

Consider the following problem. Suppose that we want to count the number of internal nodes in a tree with nodes of different order:

```
data Tree
  | Node0
  | Node1  sub1 : Tree
  | Node2  sub1, sub2 : Tree
  | Node3  sub1, sub2, sub3 : Tree
```

With a use-rule with `+` and `0`, we automatically get the aggregation of the subtrees and leafs:

```
attr Tree
   syn count use {+} {0} :: Int
```

All we still have to do is add one to this `count` attribute for each internal node. We would like to express this with a single rule, but this is unfortunately not possible:

```
sem Tree | Node1 Node2 Node3
  lhs.count = 1 + ???
```

At the place of the `???` we would like to have the value that the use-rule would produce for lhs.count. We cannot fill in `???` manually because it is different for `Node1`, `Node2`, and `Node3`.

Our solution is the possibility to transform the value for a synthesized attribute:

```
sem Tree | Node1 Node2 Node3
  +count = (+1)
```

In this example, the automatically computed value for `count` is transformed by (+1).

In general, consider a synthesized attribute `x` of type `T` and a rule for it:

```
attr ... syn x :: T
sem ... | ...   lhs.x = expr
```

Suppose that we want to transform this value before it is passed on the the LHS, but do not want to change the existing rule. We disallow to refer to `lhs.x` from other rules, but provide the possibility to transform this value:

```
sem ... | ...
  +x = f
```

Expression `f` has type `T -> T`. It may refer to other attributes.

As result, we actually produce the rule:

```
sem ... | ...  lhs.x = f expr
```

If there is more than a single `+x`, with functions f1...fn respectively, we apply these functions after each other in some arbitrary order.

Override semantics of children with `around`

To override or change the semantics of a child, you can use an around-rule:

```
sem MyData
  | Constr
      around child = \semOfChild -> modifiedSemOfChild
```

For example, you can add additional constructors on top of a child:

```
data MyData
  | Con     body :: MyData
  | Filter  body :: MyData

sem MyData
  | Con     around body = \b -> sem_MyData_Filter b
  | Filter  ...
```

We can see these extra constructors as filter nodes. With such filtern nodes, you have a relatively easy mechanism to conditionally override several inherited and synthesized attributes of a child, while still having access to the values computed by the original rules (which may be copy rules).

The right hand side of an around rule may contain attributes. When cycle checking is enabled, such an attribute may not depend on a synthesized attribute of the child on the left hand side of the rule.

Using old syntax

An AG-file consists of AG-parts and Haskell-parts. While for purposes besides programming, like inclusion in papers or teaching, the Haskell-mode syntax is desired, the old syntax might be used to to stress that the UUAG system is a preprocessor for Haskell. For example, keywords are in capitals, and there are many notational conveniences which work when defining many attributes and have large AG-files.

The following example is written in this syntax:

```
DATA D
  | C  x : Int

ATTR D [ a : Bool b : Int | | c : String ]

SEM D
  | C  lhs.c = show @lhs.a ++ show (@lhs.b + @x)
```

All AG-keywords are in upper case. To declare inherited and synthesized attributes, you need to use the special `[ ... | ... | ... ]` notation. The `[ ... | ... | ... ]` construct is a three-place delimiter for top-down (inherited), threaded, and bottom-up (synthesized) attributes, respectively. Furthermore, attributes are seperated by spaces instead of comma's, and the type signatures requires single colons instead of a double colons.

Using the Kennedy-Warren ordering algorithm

With the `--kennedywarren` flag the UUAG compiler will generate more efficient code for Ordered Attribute Grammars. At compile time the dependencies are used to construct an evaluation algorithm with multiple visit sequences. In applications where efficiency is an issue it is recommended to turn this flag on. However, not all other options are compatible with this flag.

Pragmas

There are several pragmas that influence the code generation process. You can specify a pragma at block-level using the keyword `pragma` followed by the name of the pragma in lowercase. Pragmas in a source file overrule commandline options.

optimize

```
pragma optimize
```

Generates more efficient Haskell code (multi-pass) for Ordered Attribute Grammars. A dependency analysis is performed to determine this order. For this to work, you need to:

- Ensure that the dependency analysis is able to determine an order. This means that the AG must be cycle free. The analysis is safe but incomplete, so some more may be needed, such as adding manual dependencies to get rid of *induced cycles*.
- Add type signatures for local attributes that are needed in more than one pass.
- The analysis assumes that each AG rule is strict in its attributes. Therefore, ensure that the additional greedyness does not affect the outcome of the program.

bangpats

```
pragma bangpats
```

Uses the GHC bang pattern extension to force evaluation of attributes. This increases the eagerness of the AG code.

sepsemmods

Generates a separate Haskell module for each semantic function. Example:

```
module {MyAG}
{} -- exports for the main Haskell module, keep empty for none
{
import qualified Data.Set as Set
} -- imports to be included in all modules for semantic functions for this AG

-- causes a module MyAG.hs to be generated
-- and causes a module MyAG_common.hs to be generated
pragma sepsemmods

data D1 | C1     -- causes a module MyAG_D1 to be generated
data D2 | C2     -- causes a module MyAG_D2 to be generated
```

```
sem D1 | C1 ...
sem D2 | C2 ...

{
-- code to be included in MyAG_common.hs
}

imports
{
-- import code to be included in MyAG_common.hs
}

attach D2
{
-- code to be included in MyAG_D2.hs
}

imports attach D1
{
-- import code to be included in MyAG_D1.hs
}
```

The generated Haskell code for an AG can be big. This pragma causes the code to be partitioned in smaller modules, which are easier to digest by GHC (especially when compiling with -O2). Additionally, the modules of unchanged semantic functions will not be compiled again, saving compilation times.

This pragma cannot always be used. To prevent cyclic modules, some wrapper code is not accessible from semantic functions. Code generated for higher-order functions may not compile in combination with this feature. Also, type signatures for semantic functions need to be generated to resolve type class defaulting problems, where one particular modules expects an `Int` where some other module expects and `Integer`.

genlinepragmas

```
pragma genlinepragmas
```

Generate GHC `LINE` pragmas. GHC errors will then be given in terms of the locations in the AG source code, instead of the Haskell source code.

gencostcentres

```
pragma gencostcentres
```

Adds cost centers to each AG rule. This gives preciser profiling information.

gentraces

```
pragma gentraces
```

Adds `trace` expressions to the right hand sides of all rules. This gives information about the execution of the AG, although it may be hard to interpret due to lazy evaluation. The results are easier to interpet in combination with the `optimize` pragma.

newtypes

```
pragma newtypes
```

Generates `newtype=` data types instead of `type` synonyms for the types of semantic functions (like `T_Nonterminal`).

strictdata

```
pragma strictdata
```

When generating data types, declares all fields to be strict.

strictwrap

```
pragma strictwrap
```

Generate wrappers that force evaluation of the parameters passed as inherited attributes, and of the synthesized attributes before returning them as a tuple.

datarecords

Generates record data types instead of conventional data types. To enable, specify the commandline option: `--datarecords`, or use `pragma datarecords` in the source code. This feature is implied when the commandline option -a is used.

The fields are named `x_N_C`, where `x` is the name of the child/symbol, `N` the name of the nonterminal, and `C` the name of the constructor.

optpragmas

To put Haskell pragmas above the module header, a Haskell code block can be labelled as optpragmas as follows:

```
optpragmas
{
-- This goes verbatim above the module header
{-# LANGUAGE ScopedTypeVariables #-}
}
```

Generated code

Module header

There are two ways in which a module header can be specified.

The first option is to use the `module` syntax in the source file as follows:

```
module {ModuleName}
{
-- exported functions for this Haskell module, keep empty for all
}
{
-- imports for this module
}
```

This generates a Haskell module header with the name, exports and imports in the right place.

The other possibility is to provide the option `-m` or `--module=[name]` to the UUAG compiler. If a name is provided to the `--module` flag then this name is used as module name, otherwise the module name will be the filename without the suffix `.ag` or `.lag`.

Data types

When the flag `--data` or `-d` is passed to the UUAG compiler, then a data type definition is generated for each nonterminal introduced in a `data` declaration and a type synonym is generated for each nonterminal introduced in a `type` declaration. The UUAG system allows different nonterminals to have constuctors with the same names. For Haskell data types this is not allowed. To prevent clashes between constuctors of different data types the flag `--rename` or `-r` can be specified. All constructors will then be prefixed with their corresponding nonterminal(and an underscore).

For example for this fragment of UUAG code:

```
data Expr
   | Var name :: String
   | Apply fun :: Expr arg :: Expr
   | Tuple elems :: Exprs
   | ...

type Exprs = [Expr]

data Type
   | Var name :: String
   | Apply fun :: Type arg :: Type
   | ...
```

the following Haskell code is generated when the flags `--data` and `--rename` are switched on:

```
data Expr
   = Expr_Var String
   | Expr_Apply Expr Expr
   | ...

type Exprs = [Expr]

data Type
   = Type_Var String
   | Type_Apply Type Type
   | ...
```

If the `--rename` flag is not provided it is the responsibility of the programmer to make sure that are constructors are uniquely named.

Semantic functions

The semantic domain of a nonterminal is a mapping from its inherited to its synthesized attributes. When the flag `--semfuns` or `-f` is switched on, the UUAG compiler generates for each nonterminal a type synonym representing its semantic domain, and for each constructor a semantic function. A semantic function takes the semantics of its children as arguments and returns the semantics of the corresponding nonterminal. A semantic function is named as follows: `prefix_nonterminal_constructor`. The default prefix is `sem`, another prefix can be supplied with the `--prefix==name` flag.

Consider the following code fragment:

```
data Tree
   | Bin left,right :: Tree
   | Leaf val :: Int

attr Tree
  inh lmin :: Int
  inh gmin :: Int
  syn lmin :: Int
  syn result :: Tree

sem Tree
  | Bin  lhs.result = Bin @left.result @right.result
  | Leaf lhs.lmin   = min @lhs.lmin @val
         lhs.result = Leaf @lhs.gmin
```

The semantic domain of the nonterminal *Tree* is defined as follows:

```
type T_Tree = Int -> Int -> (Int,Tree)
```

The inherited attributes are arguments and the synthesized attributes are packed together in a tuple as result. The UUAG system lexicographically sorts the attributes, hence the first *Int* stands for the inherited attribute *gmin*, and the second for the inherited attribute *lmin*. If the flag `--newtypes` is switched on, a `newtype` declaration is generated for the semantic domain instead of a `type` synonym.

The types of the generated semantic functions for the constructors *Bin*, and the *Leaf* are the following:

```
sem_Tree_Bin  :: T_Tree -> T_Tree -> T_Tree
sem_Tree_Leaf :: Int              -> T_Tree
```

Note that the semantics of a child that has a Haskell type is simply the value of that child. When the flag `--signatures` or `-s` is switched on then the type signatures of the semantic functions are actually emmited in the generated code.

Catamorphisms

When the flag `--catas` or `-c` is supplied, the the UUAG compiler generates catamorphisms for every nonterminal. A catamorphism is a function that takes a (syntax) tree as argument and computes the semantics of that tree. The catamorphism for a nonterminal *nt* is named `sem_nt` . As for semantic functions the prefix is `sem` by default, and can be changed with the `--prefix==name` flag. For example the type of the catamorphism for the nonterminal *Tree* is:

```
sem_Tree :: Tree -> T_Tree
```

When the flag `--signatures` or `-s` is switched on then the type signatures of the catamorphisms are actually emmited in the generated code.

Wrappers

The result of a semantic function or a catamorphism is a function from inherited to synthesized attributes. To be able to use such a result, a programmer needs to know the order of all the attributes. Wrapper functions for the semantic domains can be generated to provide access to the attributes by their names. When the flag `--wrappers` or `-w` is switched on the following is generated for each semantic domain:

- a record type with named fields for the inherited attributes
- a record type with named fields for the synthesized attributes
- a wrapper function that transforms a semantic domain in a function from a record of inherited attributes to a record of synthesized attributes

The two record types for a nonterminal `nt` are called `nt_Inh` and `nt_Syn`, for the inherited and synthesized attributes, respectively. The labels of the records are the names of the attributes suffixed with the name of the record type. The generated wrapper function is named `wrap_nt`.

For the nonterminal *Tree* in the example above the following record types are generated:

```
data Tree_Inh = Tree_Inh{ lmin_Tree_Inh :: Int
                        , gmin_Tree_Inh :: Int
                        }
data Tree_Syn = Tree_Syn{ lmin_Tree_Syn   :: Int
                        , result_Tree_Syn :: Tree
                        }
```

The generated wrapper function has the following type:

```
wrap_Tree :: T_Tree -> Tree_Inh -> Tree_Syn
```

Using the generated wrapper code the function *repmin* can be defined as follows:

```
repmin :: Tree -> Tree
repmin tree = let synthesized = wrap_Tree (sem_Tree tree) inherited
                  inherited   =
                        Tree_Inh
                        { lmin_Tree_Inh = infty
                        , gmin_Tree_Inh = lmin_Tree_Syn synthesized
                        }
                  infty       = 1000
              in result_Tree_Syn synthesized
```

Clean language backend

The UUAG compiler can also generate Clean code. For example, the first example can be written as follows:

```
module {Example1}
{}
{
from StdClass import class + (..)
from StdInt import instance + Int
import StdMisc
}

data Tree
   | Node  left  :: Tree
           right :: Tree
   | Tip   value :: Int

attr Tree
   syn sum :: Int

sem Tree
  | Node  lhs.sum  =  @left.sum + @right.sum
  | Tip   lhs.sum  =  @value

{
Start = test

testTree :: Tree
testTree = Node (Tip 1) (Node (Tip 2) (Tip 3))

test :: Int
test = sum_Syn_Tree (wrap_Tree (sem_Tree testTree) Inh_Tree)
}
```

When compiled with `uuagc --kennedywarren --cleanlang -dcfswmH` this results in a Clean program printing 6.

More examples

Falco Peijnenburg maintains a github repo with some examples also using cabal.

Compiler flags

short option	long option	description
`-m`	`--module[=name]`	generate module header, specify module name
`-d`	`--data`	generate data type definition
	`--datarecords`	generate record data types
	`--strictdata`	generate strict data fields (when data is generated)
	`--strictwrap`	generate strict wrap fields for WRAPPER generated data
`-c`	`--catas`	generate catamorphisms
`-f`	`--semfuns`	generate semantic functions
`-s`	`--signatures`	generate signatures for semantic functions
	`--newtypes`	use newtypes instead of type synonyms
`-p`	`--pretty`	generate pretty printed list of attributes
`-w`	`--wrappers`	generate wappers for semantic domains
`-r`	`--rename`	rename data constructors
	`--modcopy`	use modified copy rule
	`--nest`	use nested tuples
	`--syntaxmacro`	experimental: generate syntax macro code (using knit catas)
`-o file`	`--output=file`	specify output file
`-v`	`--verbose`	verbose error message format
`-h`	`--help`	get (this) usage information
`-a`	`--all`	do everything (-dcfsprm)
`-P search path`	`--=search path`	specify seach path
	`--prefix=prefix`	set prefix for semantic functions, default is `sem_`
	`--self`	generate self attribute
	`--cycle`	check for cyclic definitions
	`--version`	get version information
`-O`	`--optimize`	optimize generated code (--visit --case)
	`--visit`	try generating visit functions
	`--seq`	force evaluation using function seq (visit functions only)
	`--unbox`	use unboxed tuples
	`--bangpats`	use bang patterns (visit functions only)
	`--case`	Use nested cases instead of let (visit functions only)
	`--strictcase`	Force evaluation of the scrutinee of cases (in generated code, visit functio
	`--strictercase`	Force evaluation of all variables bound by a case statement (in generated
	`--strictsem`	Force evaluation of sem-function arguments (in generated code)
	`--localcps`	Apply a local CPS transformation (in generated code, visit functions only
	`--splitsems`	Split semantic functions into smaller pieces
	`--Werrors`	Turn warnings into fatal errors
	`--Wignore`	Ignore warnings
	`--Wmax=<max errs reported>`	Sets the maximum number of errors that are reported
	`--dumpgrammar`	Dump internal grammar representation (in generated code)
	`--dumpcgrammar`	Dump internal cgrammar representation (in generated code)
	`--gentraces`	Generate trace expressions (in generated code)
	`--genusetraces`	Generate trace expressions at attribute use sites (in generated code)
	`--gencostcentres`	Generate cost centre pragmas (in generated code)
	`--genlinepragmas`	Generate GHC LINE pragmas (in generated code)
	`--sepsemmods`	Generate separate modules for semantic functions (in generated code)
`-M`	`--genfiledeps`	Generate a list of dependencies on the input AG files
	`--genvisage`	Generate output for the AG visualizer Visage
	`--aspectag`	Generate AspectAG file
	`--nogroup=attributes`	specify the attributes that won't be grouped in AspectAG
	`--extends=module`	specify a module to be extended
	`--genattrlist`	Generate a list of all explicitly defined attributes (outside irrefutable patte
	`--forceirrefutable[=file]`	Force a set of explicitly defined attributes to be irrefutable, specify file co
	`--uniquedispenser=name`	The Haskell function to call in the generated code

	`--lckeywords`	Use lowercase keywords (sem, attr) instead of the uppercase ones (SEM, A
	`--doublecolons`	Use double colons for type signatures instead of single colons
`-H`	`--haskellsyntax`	Use Haskell like syntax (equivalent to --lckeywords and --doublecolons --ge
	`--reference`	Use reference attributes
	`--monadic`	Experimental: generate monadic code
	`--ocaml`	Generate Ocaml code
	`--cleanlang`	Generate Clean code
	`--breadthfirst`	Experimental: generate breadth-first code
	`--breadthfirst-strict`	Experimental: outermost breadth-first evaluator is strict instead of lazy
	`--visitcode`	Experimental: generate visitors code
	`--kennedywarren`	Use Kennedy-Warren's algorithm for ordering
	`--statistics=FILE`	to append to Append statistics to FILE
	`--checkParseRhs`	Parse RHS of rules with Haskell parser
	`--checkParseTys`	Parse types of attrs with Haskell parser
	`--checkParseBlocks`	Parse blocks with Haskell parser
	`--checkParseHaskell`	Parse Haskell code (recognizer)
	`--nocatas=list`	of nonterms Nonterminals not to generate catas for
	`--nooptimize`	Disable optimizations
	`--parallel`	Generate a parallel evaluator (if possible)
	`--monadicwrapper`	Generate monadic wrappers
	`--helpinlining`	Generate inline directives for GHC
	`--dummytokenvisit`	Add an additional dummy parameter to visit functions
	`--tupleasdummytoken`	Use conventional tuples as dummy parameter instead of a RealWorld stat
	`--stateasdummytoken`	Use RealWorld state token as dummy parameter instead of conventional t
	`--strictdummytoken`	Strictify the dummy token that makes states and rules functions
	`--noperruletypesigs`	Do not generate type sigs for attrs passed to rules
	`--noperstatetypesigs`	Do not generate type sigs for attrs saved in node states
	`--noeagerblackholing`	Do not automatically add the eager blackholing feature for parallel progra
	`--noperrulecostcentres`	Do not generate cost centres for rules
	`--nopervisitcostcentres`	Do not generate cost centres for visits
	`--noinlinepragmas`	Definitely not generate inline directives
	`--aggressiveinlinepragmas`	Generate more aggressive inline directives
	`--latehigherorderbinding`	Generate an attribute and wrapper for late binding of higher-order attribu

:::