Modular Computational Effects

A Comparison of Techniques and Approaches

L. Thomas van Binsbergen
Royal Holloway,
University of London
ltvanbinsbergen@acm.org

1. ATTRIBUTE GRAMMARS

An AG is a context-free grammar with attributes declared for each nonterminal. The attributes represent both input and output for computations 'over' the trees described by the grammar, and can be used for providing context-information or "simulating" effects. Input is provided by *inherited attributes* and output by *synthesised attributes*. When an attribute is both inherited and synthesised we call it a *chained attribute*. Computations are described by giving *semantic definitions* to attributes in terms of other attributes. Semantic definitions are essentially equation relating an attribute occurrence to an expression built from other attributes and functions/operators defined in some host language. It is the task of an *attribute evaluator* to traverse a syntax tree and evaluate the necessary attributes according to their semantic definitions, thus performing the described computations.

1.1 UUAG

I will from now on talk specifically about UUAG, the Utrecht University Attribute Grammar formalism, which I think is the most expressive AG formalism available to functional programmers. This means, among other things, that I use the word datatype rather than nonterminal and constructor rather than production.

1.1.1 Modular Specifications

An UUAG specification consists of three components:

1. Datatype definition
2. Attribute declaration, specific to a datatype
3. Semantic definitions, specific to a constructor

The datatypes are extensible in the sense that new constructors and attributes for that type can be given in isolation (although all constructors of a datatype are grouped together which Haskell code is generated).

The attributes declared for a nonterminal raise certain requirements to be fulfilled by the semantic definitions:

ACM ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

- An inherited attribute i_1 declared for datatype X demands a definition for the attribute occurrence $X_1.i_1$, for each occurrence X1 of X as an argument to some constructor.
- A synthesised attribute $_s1$ declared for datatype X, demands a definition for attribute occurrence $\mathbf{lhs}.s_1$, for each constructor of X (**lhs** stands for left-hand side).

When a definition is missing, the UUAGC (UUAG compiler) generates definitions automatically when possible, relying on so called use- and copy-rules. As such, use- and copy-rules implicitly propagate the values of any unmentioned attributes. When we add a datatype or constructor (more syntax), we do not have to give definitions for attributes declared elsewhere. Similarly, when declaring new attributes (with the purpose of describing the semantics of some syntax) we do not have to worry about other bits of syntax that are not influenced by these attributes. To which extent the use/copy-rule mechanism enables *composing* effects will be the target of our investigation.

The UUAGC generates a single semantic function for each production of a nonterminal, even if semantic definitions are spread across different code fragments or compilation units. As such, some form of fusion takes place automatically. The resulting semantic function may not terminate, if it contains a proper cycle. However, the UUAGC can be asked to perform compile-time scheduling to avoid nontermination, effectively "unfusing" where necessary.

1.1.2 Higher-Order AGs

UUAG supports higher-order Attribute Grammars. The main idea is that the values of attributes may be syntax trees themselves, and that we may evaluate these syntax trees when necessary. We may extend a syntax tree by declaring, as part of the semantic definitions of a constructor, a new argument to that constructor (a new nonterminal occurrence, in context-free grammar terms). This additional argument will be instantiated, at runtime, by the value of some attribute (as specified in the semantic definitions). This extension greatly enhances the expressiveness of UUAG, as arbitrary recursive computations can now be defined.

A common pattern in domain of programming language semantics is to mark a certain argument to a constructor as being *terminal*, ensuring that it will not be evaluated. (A copy of) The argument may then be 'transported' to another location in the syntax tree, where it is evaluated instead. This resembles continuation passing. As I showed at the Haskell eXchange, this feature can be used to handle cut/cutfail.

1.2 Nondeterminsm and State

This sections will contain multiple attempts to rework the knapsack example from [1]. In order to keep the following specifications simple I will not use type-variables and specialise types when necessary.

1.2.1 Syntax of Nondeterminism

Datatype definition:

```
DATA Syntax
    -- terminal (indicated by curly braces) child @sol
  | Sol     sol :: {[Int]}
    -- no solution / failure
  | NoSol
    -- 2 nonterminal children called @left and @right
    -- capturing 'further syntax'
  | Choice left :: Syntax right :: Syntax
```

1.2.2 Semantics of Nondetermism

Attribute declaration:

```
ATTR Syntax
  SYN sols :: {[[Int]]}   -- list of all solutions
```

Semantic definitions:

```
SEM Syntax
  | Sol     lhs.sols = [@sol]
  | NoSol   lhs.sols = []
  | Choice  lhs.sols = @left.sols ++ @right.sols
```

1.2.3 State as semantics

We now use state to count how many choice nodes are considered. We can do so by changing the semantics of *Choice* to increment the state.

```
ATTR Syntax
  CHN counter :: {Int}
SEM Choice
  | Choice left.counter = @lhs.counter + 1
```

The following rules are generated by the copy rules for chained attributes:

```
SEM Choice
  | Choice right.counter = @left.counter
           lhs  .counter = @right.counter
    -- lhs.counter refers to synthesized occurrence
    -- on the left-hand side of =
    -- and to the inherited occurrence
    -- on the right-hand side of =
  | Sol    lhs  .counter = @lhs.counter
  | NoSol  lhs  .counter = @lhs.counter
```

1.2.4 State as syntax

We extend syntax to capture the desire to increment the state. (Off course the below can be generalised to *Set* and *Get*.)

```
ATTR Syntax
  CHN counter :: {Int}
DATA Syntax
  | Increment other :: Syntax
```

We increase the counter without having to consider other syntactic constructs. However, to get the desired outcome a syntax tree needs to be produced with *Increment* as a parent of every *Choice* node. (The burden has shifted from semantic evaluation to syntax construction.)

```
SEM Syntax
  | Increment other.counter = @lhs.counter + 1
```

The following rules, the first of which propagates all solutions, are automatically generated:

```
SEM Syntax
  | Increment lhs  .sols    = @other.sols
              lhs  .counter = @other.counter
  | Choice    left .counter = @lhs.counter
              right.counter = @left.counter
              lhs  .counter = @right.counter
  | Sol       lhs  .counter = @lhs.counter
  | NoSol     lhs  .counter = @lhs.counter
```

1.2.5 Local State

The above gives us "global state". In order to obtain "local state", significant changes have to be made to the above specification.

```
ATTR Syntax
    -- solutions are now pairs of counter and solution
  SYN sols :: {[(Int,[Int])]}
    -- by changing counter to an inherited attribute
    -- another copy-rule is active
  INH counter :: {Int}
SEM Syntax
    -- the following semantic definition describes
    -- pairing state with solutions
  | Sol lhs.sols = [(@lhs.counter, @sol)]
```

The following rules complete the specification:

```
SEM Syntax
  | Increment left .counter = @lhs.counter + 1
              lhs  .sols    = @other.sols
  | Choice    left .counter = @lhs.counter
              right.counter = @lhs.counter
              lhs  .sols    = @left.sols
                              ++ @right.sols
  | NoSol     lhs  .sols    = []
```

2. REFERENCES

[1] N. Wu, T. Schrijvers, and R. Hinze. Effect handlers in scope. *SIGPLAN Not.*, 49(12):1–12, Sept. 2014.