0.1 UUAG

Report by:	Jeroen Bransen
Participants:	ST Group of Utrecht University
Status:	stable, maintained

UUAG is the *Utrecht University Attribute Grammar* system. It is a preprocessor for Haskell that makes it easy to write *catamorphisms*, i.e., functions that do to any data type what *foldr* does to lists. Tree walks are defined using the intuitive concepts of *inherited* and *synthesized attributes*, while keeping the full expressive power of Haskell. The generated tree walks are *efficient* in both space and time.

An AG program is a collection of rules, which are pure Haskell functions between attributes. Idiomatic tree computations are neatly expressed in terms of copy, default, and collection rules. Attributes themselves can masquerade as subtrees and be analyzed accordingly (higher-order attribute). The order in which to visit the tree is derived automatically from the attribute computations. The tree walk is a single traversal from the perspective of the programmer.

Nonterminals (data types), productions (data constructors), attributes, and rules for attributes can be specified separately, and are woven and ordered automatically. These aspect-oriented programming features make AGs convenient to use in large projects.

The system is in use by a variety of large and small projects, such as the Utrecht Haskell Compiler UHC (→ **??**), the editor Proxima for structured documents (http://www.haskell.org/communities/05-2010/html/report.html#sect6.4.5), the Helium compiler (http://www.haskell.org/communities/05-2009/html/report.html#sect2.3), the Generic Haskell compiler, UUAG itself, and many master student projects. The current version is 0.9.52.1 (January 2015), is extensively tested, and is available on Hackage. There is also a Cabal plugin for easy use of AG files in Haskell projects.

We recently implemented the following enhancements:

Evaluation scheduling. We have done a project to improve the scheduling algorithms for AGs. The previously implemented algorithms for scheduling AG computations did not fully satisfy our needs; the code we write goes beyond the class of OAGs, but the algorithm by Kennedy and Warren (1976) results in an undesired increase of generated code due to non-linear evaluation orders. However, because we know that our code belongs to the class of linear orderable AGs, we wanted to find and algorithm that can find this linear order, and thus lies in between the two existing approaches. We have created a backtracking algorithm for this which is currently implemented in the UUAG (`--aoag` flag).

Another approach to this scheduling problem that we implemented is the use of SAT-solvers. The scheduling problem can be reduced to a SAT-formula and efficiently solved by existing solvers. The advantage is that this opens up possibilities for the user to influence the resulting schedule, for example

by providing a cost-function that should be minimized. We have also implemented this approach in the UUAG which uses Minisat as external SAT-solver (`--loag` flag).

We are have recently worked on the following enhancements:

Incremental evaluation. We have just finished a Ph.D. project that investigated incremental evaluation of AGs. The target of this work was to improve the UUAG compiler by adding support for incremental evaluation, for example by statically generating different evaluation orders based on changes in the input. The project has lead to several publications, but the result has not yet been implemented into the UUAG compiler.

Further reading

- http://www.cs.uu.nl/wiki/bin/view/HUT/AttributeGrammarSystem
- http://hackage.haskell.org/package/uuagc