0.1 UUAG

Report by:	Arie Middelkoop
Participants:	ST Group of Utrecht University
Status:	stable, maintained

UUAG is the *Utrecht University Attribute Grammar* system. It is a preprocessor for Haskell which makes it easy to write *catamorphisms*. I.e. functions that do to any datatype what *foldr* does to lists. You define tree walks 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. Recently, we enhanced these aspect-oriented programming features. It is now possible to add rules that transform a value of a synthesized attribute, or to transform a child and its inherited and synthesized attributes.

The system is in use by a variety of large and small projects, such as the Utrecht Haskell Compiler UHC ($\rightarrow$ **??**), the editor Proxima for structured documents, the Helium compiler ($\rightarrow$ **??**), the Generic Haskell compiler, UUAG itself, and many master student projects. The current version is 0.9.19 (April 2010), is extensively tested, and is available on Hackage.

We are working on the following enhancements of the UUAG system, building on attribute grammar research of the past in a modern setting:

Parallel evaluation We aim to evaluate attributes in parallel to take advantage of current multi-core processors. The static dependencies between attributes, allow us to identify parts that are independent and schedule them simultaneously in a safe and efficient way.

Incremental evaluation We generate code that adapts incrementally to changes in the input data. The goal is to reuse the outcome of previous computations, by recomputing only those parts of the computation that are affected by the changes.

Fixpoint evaluation When static dependencies between attributes are circular, results can still be computed through lazy evaluation when the dynamic dependencies are not. We are now incorporating a fixed-point evaluation scheme that computes a result even in case of a dynamic cycle.

Furthermore, we investigate extensions of the AG formalism to make AGs more suitable to express type inferencing. We made prototype implementations that

facilitates the dynamic construction of inference trees, to use AGs for bidirectional type rules and constraint solving.

Further reading

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