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 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 ($\rightarrow$ **??**), 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 0.9.39 (October 2011), is extensively tested, and is available on Hackage. Recently, we improved the Cabal support and ensured compatibility with GHC 7.

We are working on the following enhancements of the UUAG system:

First-class AGs We provide a translation from UUAG to AspectAG ($\rightarrow$ **??**). AspectAG is a library of strongly typed Attribute Grammars implemented using type-level programming. With this extension, we can write the main part of an AG conveniently with UUAG, and use AspectAG for (dynamic) extensions. Our goal is to have an extensible version of the UHC.

Ordered evaluation We have implemented a variant of Kennedy and Warren (1976) for *ordered* AGs. For any absolutely non-circular AGs, this algorithm finds a static evaluation order, which solves some of the problems we had with an earlier approach for ordered AGs. A static evaluation order allows the generated code to be strict, which is important to reduce the memory usage when dealing with large ASTs. The generated code is purely functional, does not require type annotations for local attributes, and the Haskell compiler proves that the static evaluation order is correct.

Multi-core evaluation Our algorithm for ordered AGs identifies statically which subcomputations of children of a production are independent and suitable for parallel evaluation. Together with the strict evaluation as mentioned above, which is important when evaluating in parallel, the generated code can automatically exploit multi-core CPUs. We are currently evaluating the effectiveness of this approach.

Stepwise evaluation In the recent past we worked on a stepwise evaluation scheme for AGs. Using this scheme, the evaluation of a node may yield user-defined progress reports, and the evaluation to the next report is considered to be an evaluation step. By asking nodes to yield reports, we can encode the parallel exploration of trees and encode breadth-first search strategies.

We are currently also running a Ph.D. project that investigates incremental evaluation.

Further reading

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