TOOL DEMONSTRATION

Lrc - A Generator for Incremental Language-Oriented Tools

Matthijs Kuiper and João Saraiva
{kuiper,saraiva}@cs.ruu.nl

Department of Computer Science, University of Utrecht,
P.O. Box 80.089, 3508 TB Utrecht, The Netherlands

1 Introduction

LRC is a generator for graphical, easy to use, Language-Oriented Tools. LRC accepts as input an *Higher Order Attribute Grammar* (HAG) [VSK89] specifying a particular language and generates incremental attribute evaluators.

LRC generates tools that have advanced interactive interfaces and these interfaces are *computed.* That is, the interface may depend on the structure being edited and also on computed properties. A change in that structure may cause the interface to change. This provides users with powerful facilities to interact with programs.

One of the key features to handle interactive environments is the ability to perform efficient recomputations after each user interaction. The incremental evaluators produced by LRC are efficient: incremental behaviour is achieved via function caching, an efficient strategy for handling HAGS [PSV92,CP96]. Furthermore, LRC uses several optimizations which increase the evaluators' performance, such as, the *elimination of nodes and copy rules* and *tree deforestation* [SKS97].

The system has been used to develop several kind of applications such as: language based editors, compilers and even simple games. It has also been used to support a compiler construction course. LRC produces portable C and Haskell code. LRC itself and the generated tools can be installed in any system with an ANSI C compiler and the Tcl/Tk toolkit.

The LRC system can be found on the Internet at the address:

`http : //www.cs.ruu.nl/people/matthys/lrc_html/`

2 Components of Lrc

The LRC system consists of three components:

- The **generator**, constructs purely functional attribute evaluators for higher-order attribute grammars. The evaluators consist of strict functions which recursively call each other in order to decorate trees.

- The **function cache**, the cache makes evaluators incremental by storing calls to visit-functions. Incremental behaviour is achieved by reusing results of visit-function calls previously cached.
- The **incremental screen updater**: It is a modern graphical user interface tool. It provides a predefined set of widgets like menus and buttons. These widgets can be easily included in the HAG specification.

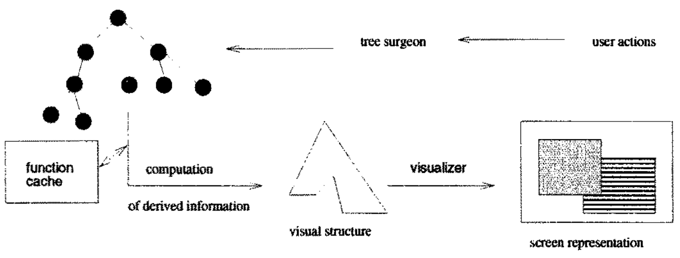


Fig. 1. Operation of tools.

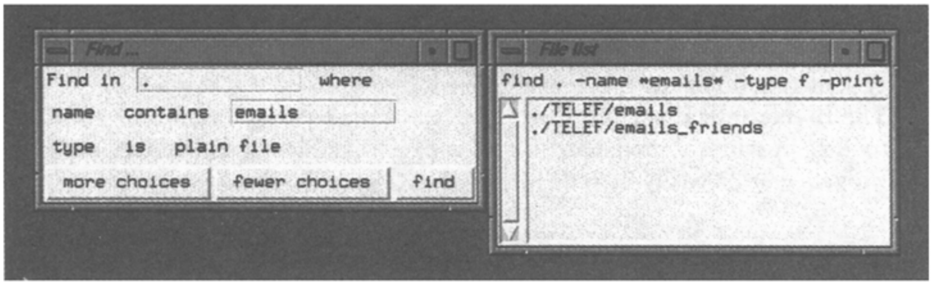
The generator translates the higher order attribute grammar into a set of pure visit-functions. These functions are incrementally evaluated using a function cache. One of the attributes being computed by the visit-functions is an abstract description of the interface. This abstract interface is presented on the screen by a *visualizer*. After a change to the term the abstract interface is incrementally recomputed and then redisplayed by the visualizer. The visualizer is itself incremental and only updates screen parts which have changed (figure 1).

3 An Example

To show the graphical user interface which LRC produces we have defined an interactive interface to the Unix find command. Interactive file finders, like the one on the Macintosh, let users define a predicate on filenames and file contents with the aid of buttons and menus. We have defined a *language of predicates* in LRC's specification formalism and generated a tool from that definition. Figure 2 displays the interface of the tool generated by LRC. The operations offered to a user depend on the predicate under construction. The button labeled with "fewer choices" in window *Find* of figure 2 removes the last choice. This button is only displayed if two or more choices are presented.

4 The Generator

The generator translates higher order attribute grammars into incremental attribute evaluators. These attribute evaluators are based on the visit-sequence



```
Find in .                where
name   contains  emails
type   is   plain file
```

```
find . -name *emails* -type f -print
./TELEF/emails
./TELEF/emails_friends
```

Fig. 2. The Interactive File Finder.

paradigm [Kas80] which perform multiple traversals over the abstract syntax trees. The attribute evaluators are purely functional and consist of a set of strict functions, called *visit-functions*. Each visit-function takes as parameter a tree and a subset of the inherited attributes of the tree's root node and returns a subset of the synthesized attributes of the tree. The entire evaluator consists of visit-functions that recursively call each other in order to decorate the tree.

Attribute values are not stored in the nodes of the tree, they only exist as arguments and results of the visit-functions. Since attributes are not stored in the tree, multiple instances of the same tree can be shared. Sharing not only reduces memory consumption, it also allows for fast equality tests between terms: a pointer comparison suffices. This is particularly important when decorating higher order attributes which may be large attributed trees and which must be compared for equality.

Other features of the Generator:

- *Parallel Evaluation*: LRC includes a parallelism detector which identifies independent visit-function calls (*i.e.*, independent visit operations of the visit sequences). Visit-functions are amenable for parallel evaluation since they do not have side effects.
- *Sliced Attribute Evaluators*: LRC produces sliced attribute evaluators. The slices are produced in respect to the synthesized attributes of the AG's root symbol. A sliced evaluator includes only the computations needed to evaluate the synthesized attributes of the AG's root considered in the slice.
- *Readable Attribute Evaluators*: LRC produces readable attribute evaluators. Moreover, it generates a LaTeX version of the HASKELL based evaluators. The LaTeX evaluators use different colours and fonts for different entities of the evaluators. They also shadow the computations not performed in a AG's slicing, identify the visit-function dependencies, etc.

5 The Function Cache

Incremental attribute evaluation is achieved through the *caching* of calls to the visit-functions. An entry in the *visit-function cache* stores the arguments and results of one visit-function call. When a visit-function is called a lookup in the cache is performed. If the cache contains an entry corresponding to the call then the result in that entry is returned. Otherwise, the visit-function is applied to the arguments and the call is cached.

To prevent unbounded growth of the function cache LRC uses several function cache update algorithms [SKS96].

6 The Incremental Screen Updater

LRC produces modern graphical user interface tools. It defines a set of predefined widgets like menus and buttons. Widgets can be horizontal or vertical combined to form the interface. The text widget is used to offer more traditional textual editing.

The interface of the tool is defined in the higher order attribute grammar. The widgets which are used in the interface and how they are combined is defined in the HAG. The *visual structure* of the tools is computed from the underlying data structure during attribute evaluation.

The incremental screen updater contains a *visualizer* which performs incremental screen updates. After each user interaction the visualizer computes the difference between the actual visual structure presented in the screen with the visual structure that must be presented. This difference is translated into incremental updates of the screen.

References

[CP96] Alan Carle and Lori Pollock. On the optimality of change propagation for incremental evaluation of hierarchical attribute grammars. *ACM Transactions on Programming Languages and Systems*, 18(1):16–29, January 1996.

[Kas80] Uwe Kastens. Ordered attribute grammars. *Acta Informatica*, 13:229–256, 1980.

[PSV92] Maarten Pennings, Doaitse Swierstra, and Harald Vogt. Using cached functions and constructors for incremental attribute evaluation. In M. Bruynooghe and M. Wirsing, editors, *Programming Language Implementation and Logic Programming*, volume 631 of *LNCS*, pages 130–144. Springer-Verlag, 1992.

[SKS96] João Saraiva, Matthijs Kuiper, and Doaitse Swierstra. Effective Function Cache Management for Incremental Attribute Evaluation. Technical report UU-CS-1996-50, Department of Computer Science, Utrecht University, November 1996.

[SKS97] João Saraiva, Matthijs Kuiper, and Doaitse Swierstra. Specializing Trees for Efficient Functional Decoration. In Michael Leuschel, editor, *ILPS97 Workshop on Specialization of Declarative Programs and its Applications*, October 1997. (Also available as Technical Report CW 255, Department of Computer Science, Katholieke Universiteit Leuven , Belgium).

[VSK89] Harald Vogt, Doaitse Swierstra, and Matthijs Kuiper. Higher order attribute grammars. In *ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*, volume 24, pages 131–145. ACM, July 1989.