Comprehension of Generative Techniques

V. Winter, C. Scalzo, A. Mametjanov, B. Kucera, and A. Jain

Department of Computer Science
University of Nebraska at Omaha

STS 2006

Outline

Outline

Inception

- We want to develop tracing facilities for the HATS software transformation system.
- We want to provide users with an abstract view of the computational model underlying HATS.
- We want to use the above model to help users understand dynamic behavior and link it to its static description.

Outline

History of the Project

- Version 0- Proof of Concept Model
  - What I called the Draft Version
  - Only shows static code
  - Didn't focus on use of system resources
  - Finish on March 23, 2005 by Brent Kucera.
- Version 1- Summer Fun
  - Looked at XML usage to cut back on system resources (88% less)
  - Added more states than pass/fail
  - Had some higher-order context.
  - Finish on July 13, 2006
- Version 2-Current "Fun"
  - Better way of showing trees.
  - Add the concept of subtree hiding.
  - Shows all higher order concepts
  - **Hope** to be done in December 2006

Outline

1. Background
   - Motivation
   - Project History
2. Previous Work
   - Software Visualization
   - Programming Debugging
3. Real Example
   - A Conceptual Overview of TL
   - Demo

Software Visualization

- Flowcharts
- Dynamic Images of Data Structures
- Pretty-Printing (color and format)
- Nassi-Shneiderman diagram*
- Web-based systems*
- parallel program visualization*
- 3-D Computational visualization*

* Due to time these will not be included in this talk

- 1947- Flowcharts
  - Created by Goldstein and von Neumann
  - Show the importance of the path of control though execution
  - Very basic way of showing information
- 1959- Automatic Flowcharts
  - Habit developed a system that drew them from assembly language or Fortran
  - Knuth developed a system in 1963 that also integrated documentation to add extra depth to his flow charts
  - Still very basic way of showing information

Software Visualization 1968 - Images

- Baecker made a debugger for the TX-2 computer that produced images of data structures
- Lead to a system for displaying data structures on a running program
- This system was live and interactive as well.
- Close to something that we would need!!!

Software Visualization 1975 - pretty-printing

- Ledgard cited with coming up with the idea
- Describing the use of spacing, indentation, and layout to make source code easy to read
- Many system where developed for automatic pretty-printing.

```
package languagePackage;

public class foo                         //class heading
{

    final int COOL = 42;                 //COOL Gobal

    public void main()                   //Main func call
    {
        System.out.print("Hello world"); // Haha got you...no hello world here

        int bar = 98 + 2 - COOL;         //Simple Math...

        if(bar != 98 + 2 - COOL)         //If this is ever wrong, Dijkstra help us
        {
            System.out.println(bar);     //Print me some bar

        }
    }

}
```

Debugging

- Pass/Fail (Any)
- Inadmissible (Functional)
- Logical Program Debugging
- Automatic Debugging

Outline

TL is a higher-order strategic programming language in which:

- The application of rules to a term is controlled
  - at the rule level by: matching and conditions.
  - at the strategy level by: combinators.
- The application of a strategy to a collection/sequence of terms is controlled by
  - traversals (TDL) and iterators (FIX)

Specs of TL

- higher-order (labelled) conditional rewrites enabling strategies to be created dynamically
- first-order matching
- a library of standard traversals
- user defined traversals
- most standard strategic binary combinators including: sequential composition (<;), left-biased choice (<+), and right-biased choice (+>).
- a variety of unary combinators, most notably the transient() combinator

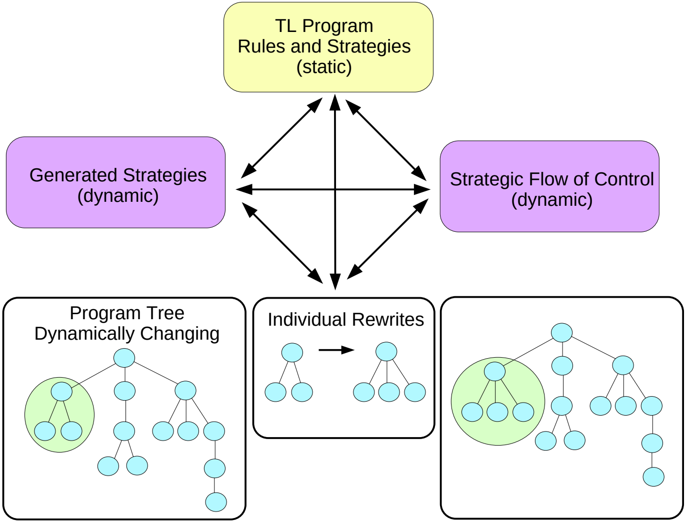
Basis A term is a strategy of type τ_0.

Induction Let let *lhs* and *rhs* denote a strategy of type τ_0 and τ_n respectively. Then

lhs $\rightarrow$ *rhs if cond*

denotes a rule of type τ_{n+1}.

A **strategy** is an expression composed of rules, rule abstractions (i.e., labels), combinators, traversals, and iterators.


TL Program
Rules and Strategies
(static)
Generated Strategies
(dynamic)
Strategic Flow of Control
(dynamic)
Program Tree
Dynamically Changing
Individual Rewrites

An Abstract Strategic Program

$$rule_1 : lhs_1 \rightarrow rhs_1 \text{ if } cond_1$$

$$rule_2 : lhs_2 \rightarrow rhs_2 \text{ if } cond_2$$

$$strategy : TDL\{rule_1 <\!+ \; transient(rule_2)\}$$

Traceable Elements

$$\boxed{rule_1 : lhs_1 \longrightarrow rhs_1 \text{ if } \boxed{cond_1}}$$

$$\boxed{rule_2 : lhs_2 \longrightarrow rhs_2 \text{ if } \boxed{cond_2}}$$

$$\boxed{strategy : TDL\{ \boxed{\boxed{rule_1} <+ \boxed{transient(\boxed{rule_2})}} \}}$$

Issues

- Specification of which “boxes” are of interest with respect to a particular transformational behavior.
- Display of interesting sequences of entities (i.e., boxes).
- The role played by a set of entities with respect to overall transformation.

Outline

$$union : set\_pgm[\![set_{super} \; union \; set_{this}]\!]$$
$$\rightarrow$$
$$set\_pgm[\![set_{super} \; union \; set_{this} \Rightarrow set_{scope\_this}]\!]$$
$$\text{if } set_{scope\_this} \ll BUL\{ \; lcond\_tdl\{get\_elements\}[set_{this}] \; \}(set_{super})$$

$$get\_elements : elements[\![class_{this}.value_1 \; elements_1]\!]$$
$$\rightarrow$$
$$transient(elements[\![class_{super}.value_1 \; elements_3]\!]$$
$$\rightarrow$$
$$elements[\![class_{this}.value_1 \; elements_3]\!]$$
$$<+$$
$$elements[\![\;]\!] \rightarrow elements[\![class_{this}.value_1]\!]$$
$$)$$

My favorite slide of them ALL

End of slides... yep scary live demo, if I have time...

...Or questions if we don't want the live demo!