The Correspondence of Term Rewriting and Attribute Grammars

Shirley Goldrei

SAPLING II, June 12th 2007, Macquarie University

Two basic abstractions for computation on trees

- These are used in compiler generation system for defining semantic analysis and code generation
  1. Term Rewriting
  2. Attribute Grammar
- Examples of compiler generation systems
  - Stratego/XT
  - Eli

Attribute Grammars - the basic idea

- Start with a tree
- Annotate the nodes with named values (attributes)
- Attributes are computed by functions on attribute values in other nodes
- Given a grammar and these attribute dependencies an evaluator can be automatically generated

Term Rewriting - the basic idea

- Start with a tree
- Rules say “Everywhere you find this subtree replace it with this other subtree” (and repeat)
- The order in which rules are applied and the tree is searched is either fixed or you explicitly define them

So what's so good about these abstractions?

- Attribute Grammars
  - Great for analysis tasks such as name or type analysis where you don't want the tree structure to change
- Term Rewriting
  - Great for simplifying expressions, optimisation and other task where you want to effect changes to the tree structure

[plrg] macquarie

MACQUARIE UNIVERSITY~SYDNEY

What's so good II

- Attribute Grammars
  - You never have to think about the order of tree traversals directly
- Term Rewriting
  - Unfortunately either traversal order is fixed or you are responsible for ensuring the rewrites are confluent by programming the traversals yourself

Now you want to write your own compiler

- You start with a tree
- You want to do some analysis
  - Say, check for programmer errors
- You might want to transform the source tree
  - Say, into an intermediate language
- Do some more analysis
  - Say, some dataflow analysis
- Do some more transformation
  - Say, some optimisation
- etc. etc. etc.

What are you going to use?

What are you going to use?

- Attribute Grammars?

What are you going to use?

- Attribute Grammars?
- Term Rewriting?

What are you going to use?

- Attribute Grammars?
- Term Rewriting?
- Your favourite general purpose language?

What are you going to use?

- Attribute Grammars?
- Term Rewriting?
- Your favourite general purpose language?
  - Most people choose this last option

What are you going to use?

- Attribute Grammars?
- Term Rewriting?
- Your favourite general purpose language?
  - Most people choose this last option
- An improved abstraction that combines the benefits of the others?

Steps to combining AG and TR

- Show a correspondence between the two abstractions

  Informal

  - Translate Term Rewriting into a Higher Order Attribute Grammar Specification

  Formal

  - Describe both abstractions in terms of a single calculus

First Strategy

- With an appropriate syntax and an automatic translation from Rewriting to HAG we can use an AG evaluator with minimal change

Operations of Rewriting

(e.g. Stratego - Bravenboer et. al. 2005)

Operator	Success/Failure	Effect on Tree	Effect on Env.
id	always succeeds	none	none
fail	always fails	none	none
build !t	always succeeds	replace current subtree	none
match and bind ?t0(t1..tn)	succeeds if the current term “matches” the given term in a refined environment	none	augmented with bindings if match succeeds
binary sequential composition s1;s2	succeed iff both sides succeed	left side followed by the right side	combined effect of both sides
binary left choice combinator	succeeds if either side succeeds	left side or right side	either lhs bindings are added or rhs
non-deterministic choice	as above	as above	as above

Example

GRAMMAR:

```
prog: Program -> Expr
plus: Expr -> Expr Expr
times: Expr -> Expr Expr
const: Expr -> idn
```

REWRITE RULE:

```
... ?times( e1, const(“0”)) ; !const(“0”) ...
... times( e1, const(“0”)) -> const(“0”) ...
```

ATTRIBUTE GRAMMAR SPECIFICATION:

```
RULE times: Expr ::= Expr Expr $ Expr COMPUTE
  Expr[1].s1_e1 = Expr[2].GENTREE; //binding
  Expr[1].s1_match = Expr[3].s1_match;
  Expr[4].GENTREE = IF (Expr[1].s1_match)
                              THEN mkConst(“0”) ELSE mkNOTREE;
END;

RULE const: Expr ::= idn COMPUTE
   Expr.s1_match = EQUALS( idn, "0");
END;
```

Tree Walking Operations

(e.g. Stratego - Bravenboer et. al. 2005)

Operator	Success/Failure	Effect on Tree	Effect on Env.
one(s)	succeeds if s succeeds on one child	first child subtree is replaced	augmented by all bindings in s
some(s)	succeeds if s succeeds on at least one child	all children on which s succeeds are replaced	augmented by all bindings in s
all(s)	succeeds if s succeeds on all children	replace all children subtrees	augmented by all bindings in s
congruence	...	...	...
recursive closure rec(s)	succeeds if s succeeds	...	...

Example cont.

GRAMMAR:

```
prog: Program -> Expr
plus: Expr -> Expr Expr
times: Expr -> Expr Expr
const: Expr -> idn
```

REWRITE RULE:

```
one(times( e1, const("0")) -> const("0"))
```

ATTRIBUTE GRAMMAR SPECIFICATION:

```
RULE prog: Program ::= Expr  COMPUTE
  Program.s4_succeed = Expr[1].s2_match
END;

RULE times: Expr ::= Expr Expr $ Expr COMPUTE

    ... as before...
```

Second Strategy

a little more formal

- Extend the denotational semantics of Gondow and Katayama 2000
- Build on their semantics of Higher Order Attribute Grammars in terms of Cardelli record calculus
- Add a semantics of rewriting in the same calculus
- Implement prototype in Haskell

References

- Bravenboer, M., van Dam, A., Olmos, K., and Visser, E. Program transformation with scoped dynamic rewrite rules. Tech. Rep. UU-CS-2005-005, Institute of Information and Computing Sciences, Utrecht University, 2005.
- Gondow, K., and Katayama, T. Attribute grammars as record calculus - a structure oriented denotational semantics of attribute grammars by using Cardelli's record calculus. *Informatica (Slovenia) 24*, 3 (2000).