Program Transformation with Stratego/XT

Margus Freudenthal

Based on
Program Transformation with Stratego/XT.
Rules, Strategies, Tools, and Systems in
Stratego/XT 0.9.
by Eelco Visser, 2004

What Is Program Transformation?

- Automatic manipulation of source programs
- Used in
  - Compilers
  - Code generators
  - Refactoring tools
  - Migrating tools
  - Reverse engineering tools

What Is Stratego/XT?

- Stratego is a language for
  - Transformation rules
  - Programmable strategies for applying these rules
- XT is a toolset containing
  - Parser generators
  - Pretty-printer generators
  - Grammar engineering tools

Transformation Rule

- Encodes a basic transformation step as a rewrite on an abstract syntax tree
- Examples:
  - ```
    EvalPlus : Plus(Int(i), Int(j)) -> Int(k)
        where <add>(i, j) => k
    ```
  - ```
    LetSplit : Let([d1, d2 | d*], e*) ->
        Let([d1], Let([d2 | d*], e*))
    ```

Representing Terms

- Rewrite rules apply to abstract syntax trees
  - As opposed to parse trees
- Abstract syntax trees are represented by first-order prefix terms
  - E.g., `If(cond, then, else)`
- Syntax tree fragments can be described using the concrete syntax of the object language.
  - ```
    EvalPlus : |[ i + j ]| -> |[ k ]|
        where <add>(i, j) => k
    ```
    ```

Transformation Rule (2)

- Rewrite rules can be broken done into the more basic actions
  - Matching
  - Building
  - Variable scope
- Typically, rewrite rules are context-free
  - Scoped dynamic rewrite rules can be used to generate rewrite rules in runtime
  - Dynamic rules can encode contextual information

Term Rewriting

- Term rewriting is the exhaustive application of a set of rewrite rules to a term until no rule applies anywhere in the term
  - Also called normalization
- Example
  - `Minus(Plus(Int(4), Plus(Int(1), Int(2))), Var("a"))`
  - ... reduces to `Minus(Int(7), Var("a"))` by repeatedly applying EvalPlus

Term Rewriting (2)

- Exhaustive rewriting is used in most rewriting tools
- However, normalizing a term with respect to *all* rules is not always desirable
- Often this is done by using special kind of rules
  - Example: encode evaluation order in rules

Programmable Rewriting

- Stratego makes rewriting strategy *explicit* and *programmable*
- One has to define explicitly
  - Which rules to apply
  - Which strategy to follow
- Example
  - `simplify = innermost(EvalPlus + LetSplit + ...)`

Transformation Strategy

- *Strategy* is an algorithm that transforms a term into another term or fails at doing so
- Combines a set of rules into a complete transformation
  - Orders their application using control and traversal combinators
- Important property: ability to define generic traversals
  - Do not depend on specific data types

Strategy Combinators

- Stratego's approach is to allow building complex strategies from very simple building blocks
  - Sequential composition (s1 ; s2)
  - Deterministic choice (s1 <+ s2; first try s1, only if that fails s2)
  - Non-deterministic choice (s1 + s2; same as <+, but the order of trying is not defined1)
  - Guarded choice (s1 < s2 + s3; if s1 succeeds then commit to s2 else s3)

Strategy Combinators (2)

- Building blocks
  - Testing (where(s); ignores the transformation achieved)
  - Negation (not(s); succeeds if s fails)
  - Recursion (rec x(s))

Strategy Definitions

- `f(x1, ..., xn) = s` — $f(x_1, \ldots, x_n) = s$
  - Define strategy *f*
  - $x_1..x_n$: strategy arguments
- Examples
  - `try(s) = s <+ id`

    Applies strategy *s*, *s*ucceeds even if it fails.
  - `repeat(s) = try(s; repeat(s))`

    Repeats transformation *s* until it fails

Strategy Definitions (2)

- Strategy definitions do not explicitly mention the term to which they are applied
- Instead, they combine term transformations into more complex term transformations

Congruence Operator

- Basically match and apply
  - ```
    control-flow(s) =
        Assign(id, s)
        + If(s, id, id)
        + While(s, id)
    ```
  - `map(s) = [] + [s | map(s)]`
- Defines traversals that are specific to a data type
    ```

Generic Traversals

- Not specific to any data type
- One-pass traversals:
  - `all(s)` – applies s to each subterm of current term
  - `bottomup(s) = all(bottomup(s)); s`
  - `topdown(s) = s; all(topdown(s))`
  - `alltd(s) = s <+ all(alltd(s))`
  - `oncetd(s) = s <+ one(oncetd(s))`

Generic Traversals (2)

- Fixpoint traversals:
  - ```
    innermost(s) =
      bottomup(
       try(s; innermost(s)))
    ```
    ```

Cascading Transformations

- Applying several small transformation steps.
  - ```
    simplify =
      innermost(R1 <+ ... <+ Rn)
    ```
    ```

Staged Transformations

- Transformations are applied in stages
  - ```
    simplify =
        innermost(A1 <+ ... <+ Ak)
      ; innermost(B1 <+ ... <+ Bl)
      ; ...
      ; innermost(C1 <+ ... <+ Cm)
    ```
- Can be combined with cascading transformations
- Because rules are independent from strategies, they can be reused in different stages
    ```

Local Transformations

- Transformations are applied only to subtree of the program where they make the most sense
  - ```
    transformation =
      alltd(
       trigger-transformation
       ; innermost(A1 <+ ... <+ An)
      )
    ```
  - *trigger-transformation* selects one node where the cascading transformation is applied
    ```

First-Class Pattern Matching

- In addition to rules, pattern matching is available as primitives in strategies
  - ```
    EvalPlus : Plus(Int(i), Int(j)) -> Int(k)
      where <add> (i, j) => k
    ```
  - ```
    EvalPlus = {i,j,k: ?Plus(Int(i), Int(j));
      where(!(i,j); add; ?k); !Int(k)}
    ```

Dynamic Rules

- Pure rewriting rules and strategies are context-free
- Passing context through arguments can quickly become tedious
  - List of defined functions
  - List of defined variables
- Stratego offers mutable global state in the form of dynamic rules

Dynamic Rules (2)

- Generated at run-time
- Can access information available from their generation context

Dynamic Rule Example

```
DeclareFun =
  ?fdec@|[ function f(x1*) ta = e1 ]|;
  rules(
    InlineFun :
      |[ f(a*) ]| -> |[ let d* in e2 end ]|
      where <rename> fdec =>
              |[ function f(x2*) ta = e2 ]|
          ; <zip(BindVar)>(x2*, a*) => d*
  )
BindVar :
  (FArg |[ x ta ]|, e) ->
    |[ var x ta := e ]|
```

Dynamic Rule Scope

- Restricts the scope of new definitions of the dynamic rule
  - Rule is removed if execution goes out of scope
- Example:
  - ```
    inline = {| InlineFun:
        try(DeclareFun)
        ; repeat(InlineFun + Simplify)
        ; all(inline)
        ; repeat(Simplify)
        |}
    ```
    ```

Term Annotations

- Abstract syntax of programs is expressed in terms
  - Term = Constructor + list of argument terms
- Sometimes it is useful to attach information to term without changing it
- This information can be stored in annotations
  - Each term has list of annotations
  - Annotations are also terms

Term Annotations (2)

- Annotations can be processed like any other terms
- Example:
  - `TypeCheck : Plus(e1{Int}, e2{Int}) ->`
    `Plus(e1, e2){Int}`
- Can be used for
  - Type checking
  - Strictness analysis
  - Bound-unbound variables analysis

Problems with Annotations

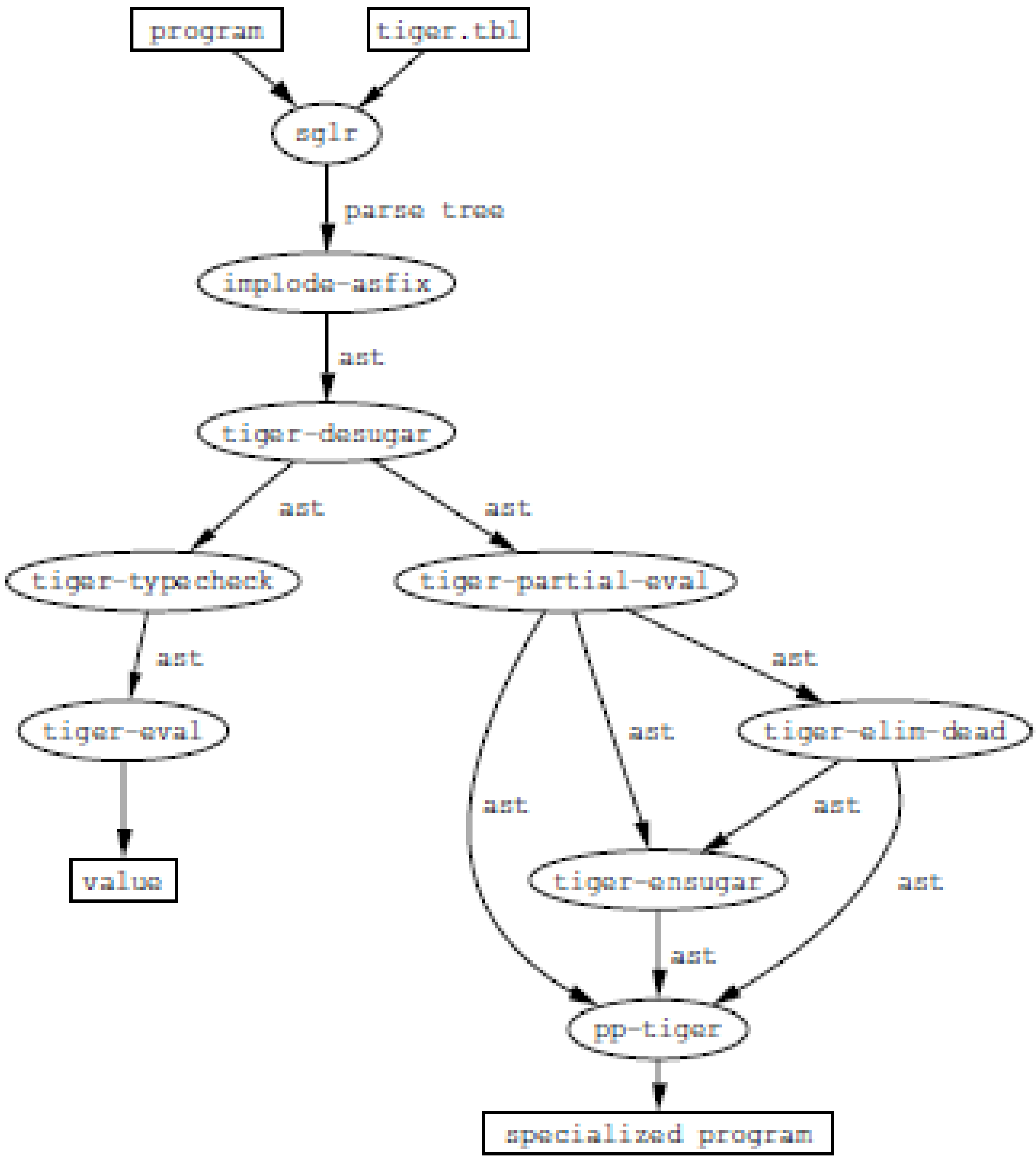
- Transformations are supposed to preserve annotations
  - Stratego's traversals do that
  - When transforming a term, this is not so simple, because this should follow semantics of the annotations
- Should annotations affect equality between terms?

Transformation Tool

- Wraps a composition of rules and strategies into a stand-alone, deployable component
- Can be called from the command-line or from other tools
- Transforms terms into terms
  - All the tools in Stratego/XT toolkit use standard ATerm format for terms

Transformation System

- Composition of tools that performs a complete source-to-source transformation
- Consists of:
  - Parser
  - Pretty-printer
  - Transformation tools



program
tiger.tbl
sglr
parse tree
implode-asfix
ast
tiger-desugar
ast
ast
tiger-typecheck
tiger-partial-eval
ast
ast
tiger-eval
ast
tiger-elim-dead
value
ast
ast
tiger-ensugar
ast
ast
pp-tiger
specialized program

Thank You