From Pattern Matching to Simple Strategies


Eelco Visser

Institute of Information & Computing Sciences
Utrecht University,
The Netherlands

February 14, 2006
Reading Assignments

Reading for first week (infrastructure)
- Stratego/XT Tutorial: Part II

Reading for this lecture (3)
- Stratego/XT Tutorial: Chapters 16, 10, 11, 12
- FI’05: Sections 1, 2, 3

Reading for next lecture (4)
- Stratego/XT Tutorial: Chapters 13, 14, 15, 17
- FI’05: Section 3.5

All assigned reading is material for exam questions

Read to understand, not just to make lab assignments
Previously

syntax definition

parser generator
parse table
parse
program

tree grammar generator
tree grammar
tree
transform
tree

pretty-printer generator
pretty-print table
pretty-print
tree
program
Realizing Program Transformations

How to realize transformations of (abstract syntax) trees?
Stratego is a language for program transformation based on term rewriting with programmable rewriting strategies.
## Term Rewriting

### Conventional Term Rewriting

- Rewrite system = set of rewrite rules
- Redex = reducible expression
- Normalization = exhaustive application of rules to term
  - (Stop when no more redices found)
- Strategy = algorithm used to search for redices
- Strategy given by engine
## Term Rewriting

### Conventional Term Rewriting
- Rewrite system = set of rewrite rules
- Redex = reducible expression
- Normalization = exhaustive application of rules to term
- (Stop when no more redices found)
- Strategy = algorithm used to search for redices
- Strategy given by engine

### Strategic Term Rewriting
- Select rules to use in a specific transformation
- Select strategy to apply
- Define your own strategy if necessary
- Combine strategies
### The Annotated Term Format

<table>
<thead>
<tr>
<th>Term</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application</td>
<td>Void(), Call(t, t)</td>
</tr>
<tr>
<td>List</td>
<td>[], [t, t, t]</td>
</tr>
<tr>
<td>Tuple</td>
<td>(t, t), (t, t, t)</td>
</tr>
<tr>
<td>Integer</td>
<td>25</td>
</tr>
<tr>
<td>Real</td>
<td>38.87</td>
</tr>
<tr>
<td>String</td>
<td>&quot;Hello world&quot;</td>
</tr>
<tr>
<td>Annotated term</td>
<td>t{t, t, t}</td>
</tr>
</tbody>
</table>
A **transformation strategy**

- transforms current term into a new term or fails
- may bind term variables
- may have side-effects (I/O, call other process)
A transformation strategy

- transforms current term into a new term or fails
- may bind term variables
- may have side-effects (I/O, call other process)

Stratego Shell: An Interactive Interpreter for Stratego

\(<\text{current term}>\)
A transformation strategy

- transforms current term into a new term or fails
- may bind term variables
- may have side-effects (I/O, call other process)

Stratego Shell: An Interactive Interpreter for Stratego

<current term>
stratego> <strategy expression>
<transformed term>
A transformation strategy

- transforms current term into a new term or fails
- may bind term variables
- may have side-effects (I/O, call other process)

Stratego Shell: An Interactive Interpreter for Stratego

```plaintext
<current term>
stratego> <strategy expression>
<transformed term>
stratego> <strategy expression>
command failed
```
Part I

Building and Matching Terms

Building and Matching Terms

Atomic actions of program transformation:

1. Creating (building) terms from patterns
2. Matching terms against patterns
Building Terms

**Build Term**

- Syntax: !t
- Replace current term by term t

```stratego> !Int("10")
Int("10")```
Building Terms

### Build Term

- **Syntax:** ! \( t \)
- **Replace current term by term** \( t \)

```plaintext
stratego> !Int("10")
Int("10")
stratego> !Plus(Var("a"),Int("10"))
Plus(Var("a"),Int("10"))
```
Build Pattern

- Syntax: !p
- Replace current term by instantiation of pattern p
- A pattern is a term with meta-variables

```stratego>
:binding e
```
```
e is bound to Var("b")
```

```stratego>
!Plus(Var("a"), e)
Plus(Var("a"), Var("b"))
```
Matching Terms

### Match Term
- Syntax: `?t`
- Match current term against term `t`
- Succeed if equal, fail if unequal

```
Plus(Var("a"), Int("3"))
```

```
stratego> ?Plus(Var("a"), Int("3"))
```

```
stratego> ?Plus(Int("3"), Var("b"))
```

command failed
### Match Variable

- **Syntax:** ?x
- **Match current term (t) against variable x**

<table>
<thead>
<tr>
<th>Plus(Var(&quot;a&quot;),Int(&quot;3&quot;))</th>
<th>stratego&gt; ?e</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>command failed</td>
</tr>
</tbody>
</table>
### Matching Terms

#### Match Variable

- **Syntax:** `?x`
- **Match current term** `(t)` **against variable** `x`
- **Binds variable** `x` **to** `t` **in the environment**

```plaintext
Plus(Var("a"),Int("3"))
stratego> ?e
stratego> :binding e
e is bound to Plus(Var("a"),Int("3"))
```
Matching Terms

**Match Variable**

- **Syntax:** `?x`
- **Match current term** \( t \) **against variable** \( x \)
- **Binds variable** \( x \) **to** \( t \) **in the environment**
- **Variable can only be bound once (or to the same term)**

\[
\text{Plus(Var("a"),Int("3"))}
\]

```stratego>
?e
stratego> :binding e
e is bound to Plus(Var("a"),Int("3"))
stratego> !Int("17")
stratego> ?e
command failed
```
Matching Terms

**Match Pattern**

- Syntax: \(?p\)
- Match current term \((t)\) against pattern \(p\)
- Succeed if there is a substitution \(\sigma\) such that \(\sigma(p) = t\)

```
Plus(Var("a"),Int("3"))
```

```
stratego> ?Plus(e,_)  
```
Matching Terms

**Match Pattern**

- Syntax: \(?p\)
- Match current term \((t)\) against pattern \(p\)
- Succeed if there is a substitution \(\sigma\) such that \(\sigma(p) = t\)
- Wildcard \(_\) matches any term

```
Plus(Var("a"), Int("3"))
```

```
stratego> ?Plus(e, _)
```
Matching Terms

Match Pattern

- Syntax: \(?p\)
- Match current term \(t\) against pattern \(p\)
- Succeed if there is a substitution \(\sigma\) such that \(\sigma(p) = t\)
- Wildcard \(\_\) matches any term
- Binds variables in \(p\) in the environment

\[
\text{Plus(Var("a"),Int("3"))}
\]

\textbf{stratego}>
\[
?\text{Plus}(e,\_)
\]

\textbf{stratego}>
\[
:binding\ e\ 
e\ is\ bound\ to\ Var("a")
\]
Matching Terms

Non-Linear Patterns

- Patterns may be non-linear: multiple occurrences of same variable
- Each occurrence matches same term

```
Plus(Var("a"),Int("3"))
stratego> ?Plus(e,e)
command failed
```
Matching Terms

Non-Linear Patterns

- Patterns may be non-linear: multiple occurrences of same variable
- Each occurrence matches same term

```
Plus(Var("a"),Int("3"))
stratego> ?Plus(e,e)
command failed
stratego> !Plus(Var("a"),Var("a"))
stratego> ?Plus(e,e)
```
Non-Linear Patterns

- Patterns may be non-linear: multiple occurrences of same variable
- Each occurrence matches same term

```
Plus(Var("a"),Int("3"))
stratego> ?Plus(e,e)
command failed
stratego> !Plus(Var("a"),Var("a"))
stratego> ?Plus(e,e)
stratego> :binding e
 e is bound to Var("a")
```
Basic transformations are combinations of match and build

Combination requires

1. Sequential composition of transformations
2. Restricting the scope of term variables

Syntactic abstractions (sugar) for typical combinations

1. Rewrite rules
2. Apply and match
3. Build and apply
4. Where
5. Conditional rewrite rules
Combining Match and Build

Sequential Composition

- Syntax: $s_1; s_2$
- Apply $s_1$, then $s_2$
- Fails if either $s_1$ or $s_2$ fails
- Variable bindings are propagated

```plaintext
Plus(Var("a"), Int("3"))

stratego> ?Plus(e1, e2); !Plus(e2, e1)
Plus(Int("3"), Var("a"))
```
Anonymous Rewrite Rule (Sugar)

- Syntax: \((p_1 \rightarrow p_2)\)
- Match \(p_1\), then build \(p_2\)
- Equivalent to: \(?p_1; !p_2\)

\[
\text{Plus}(	ext{Var("a")}, \text{Int("3")})
\]

\[
\text{stratego}\> \ (\text{Plus}(e_1, e_2) \rightarrow \text{Plus}(e_2, e_1))
\]

\[
\text{Plus}(\text{Int("3")}, \text{Var("a")})
\]
Combining Match and Build

Apply and Match (Sugar)

- Syntax: $s => p$
- Apply $s$, then match $p$
- Equivalent to: $s; ?p$

Build and Apply (Sugar)

- Syntax: $<s> p$
- Build $p$, then apply $s$
- Equivalent to: $!p; s$

stratego> <addS>("1","2") => x
"3"
stratego> :binding x
x is bound to "3"
Combining Match and Build

### Term Variable Scope

- **Syntax:** \( \{x_1, \ldots, x_n : s \} \)
- **Restrict scope of variables** \( x_1, \ldots, x_n \) **to** \( s \)

\[
\text{Plus(Var("a"),Int("3"))}
\]

```
stratego> (Plus(e1,e2) -> Plus(e2,e1))
Plus(Int("3"),Var("a"))
stratego> :binding e1
   e1 is bound to Var("a")
```

```
stratego> \{e3,e4:(Plus(e3,e4) -> Plus(e4,e3))\}
Plus(Var("a"),Int("3"))
stratego> :binding e3
   e3 is not bound to a term
```
Combining Match and Build

**Where (Sugar)**

- Syntax: `where(s)`
- Test and compute variable bindings
- Equivalent to: `{x: ?x; s; !x}
 for some fresh variable x

```plaintext
Plus(Int("14"),Int("3"))
stratego> where(?Plus(Int(i),Int(j)); <addS>(i,j) => k)
Plus(Int("14"),Int("3"))
stratego> :binding i
i is bound to "14"
stratego> :binding k
k is bound to "17"
```
Conditional Rewrite Rules (Sugar)

- Syntax: \( (p_1 \rightarrow p_2 \text{ where } s) \)
- Rewrite rule with condition \( s \)
- Equivalent to: \( (?p_1; \text{ where}(s); !p_2) \)

\[
\text{Plus}(	ext{Int}("14"),\text{Int}("3")) > \text{Plus}(	ext{Int}(i),\text{Int}(j)) \rightarrow \text{Int}(k) \text{ where } \text{<addS>}(i,j) \Rightarrow k \\
\text{Int}("17")
\]
**Combining Match and Build**

**Lambda Rules (Sugar)**

- Syntax: \( p_1 \to p_2 \) where \( s \)
- Rewrite rule with condition \( s \)
- Equivalent to: \( \{x_1, \ldots, x_n: (p_1 \to p_2 \text{ where } s)\} \)
  with \( x_1, \ldots, x_n \) the free variables in \( p_1 \)

```latex
\text{Plus}(\text{Int("14")}, \text{Int("3")})
\to \text{Int}(\text{Int("17")})
\) where \( \langle \text{addS}\rangle(i, j) \Rightarrow k \) \)
\text{:binding } i
\) i is not bound to a term
\) :binding k
\) k is bound to "17"
```
Combining Match and Build

Apply

- Syntax: \( !p[<s>p'] \)
- Strategy application in pattern
- Equivalent to: \( \{ x : \text{where}(<s>p' \Rightarrow x) ; !p[x] \} \)

Plus(Int("14"),Int("3"))
\( > \) Plus(Int(i),Int(j)) \( \Rightarrow \) Int(<addS>(i,j)) \( \setminus \)
Int("17")
Combining Match and Build

Term Wrap

- Syntax: \(!p[<s>]\)
- Strategy application in pattern to current subterm
- Equivalent to: \{x: where(s => x); !p[x]\}
  for some fresh variable x

3
stratego> !(<id>,<id>)
(3,3)
stratego> !(<Fst; inc>,<Snd>)
(4,3)

"foobar"
stratego> !Call(<id>, [])
Call("foobar", [\])
Combining Match and Build

Term Project

- Syntax: ?p[<s>]
- Strategy application in pattern match
- Equivalent to: \{x: ?p[x]; <s>x\}
  for some fresh variable x

```
[1,2,3]
stratego> ?[_]<id>]
[2,3]

Call("foobar", [])
stratego> ?Call(<id>, [])
"foobar"
```
Part II

Naming and Composing Strategies

<table>
<thead>
<tr>
<th>Strategy Definitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reuse of transformation requires definitions</td>
</tr>
<tr>
<td>1. Naming strategy expressions</td>
</tr>
<tr>
<td>2. Named rewrite rules</td>
</tr>
<tr>
<td>3. Reusing rewrite rules through modules</td>
</tr>
</tbody>
</table>
### Simple Strategy Definition and Call

- **Syntax:** $f = s$
- **Name strategy expression** $s$
- **Syntax:** $f$
- **Invoke (call) named strategy** $f$

```plaintext
Plus(Var("a"), Int("3"))

stratego> SwapArgs = \{e_1, e_2: (Plus(e_1, e_2) -> Plus(e_2, e_1))\}

stratego> SwapArgs
Plus(Int("3"), Var("a"))
```
Named Rewrite Rules (Sugar)

- Syntax: \( f : p_1 \rightarrow p_2 \text{ where } s \)
- Name rewrite rule \( p_1 \rightarrow p_2 \text{ where } s \)
- Equivalent to: \( f = \{ x_1, \ldots, x_n : (p_1 \rightarrow p_2 \text{ where } s) \} \)
  (with \( x_1, \ldots, x_n \) the variables in \( p_1, p_2, \) and \( s \))

\[
\text{Plus(Var("a"),Int("3"))}
\]

\text{stratego}\>
\text{SwapArgs : Plus(e1,e2) \rightarrow Plus(e2,e1)}

\text{stratego}\>
\text{SwapArgs}
\text{Plus(Int("3"),Var("a"))}
module Simplification-Rules
rules
EvalPlus :
    Plus(Int(i),Int(j)) -> Int(k) where <addS>(i,j) => k

PlusAssoc :
    Plus(Plus(e1, e2), e3) -> Plus(e1, Plus(e2, e3))

EvalIf :
    If(Int("0"), e1, e2) -> e2

EvalIf :
    If(Int(i), e1, e2) -> e1 where <not(eq)>(i, "0")

stratego> import Simplification-Rules
Composition of Transformation Strategies

Rules define one-step transformations

Program transformations require many one-step transformations and selection of rules

1. Choice
2. Identity, Failure, and Negation
3. Parameterized and Recursive Definitions
Deterministic Choice (Left Choice)

- Syntax: \( s_1 \leftarrow s_2 \)
- First apply \( s_1 \), if that fails apply \( s_2 \)
- Note: local backtracking

PlusAssoc :
\[
\text{Plus(Plus}(e_1, \ e_2), \ e_3) \rightarrow \text{Plus}(e_1, \ \text{Plus}(e_2, \ e_3))
\]
EvalPlus :
\[
\text{Plus(Int}(i),\text{Int}(j)) \rightarrow \text{Int}(k) \text{ where } <\text{addS}>(i, j) \Rightarrow k
\]

\[
\text{Plus(Int("14"),Int("3"))}
\]

stratego> PlusAssoc
command failed

stratego> PlusAssoc \leftarrow EvalPlus

Int("17")
### Conditional Choice

- **Syntax:** \( s_1 < s_2 + s_3 \)
- First apply \( s_1 \) if that succeeds apply \( s_2 \) to the result else apply \( s_3 \) to the original term
- Do not backtrack to \( s_3 \) if \( s_2 \) fails!

- \( ?\text{Call}(\_, \_) \) < complex-transformation-for-call
- + \( ?\text{BinOp}(\_, \_, \_) \) < transformation-for-binop
- + other-cases
Composing Strategies

Conditional Choice

- Syntax: \( s_1 < s_2 + s_3 \)
- First apply \( s_1 \) if that succeeds apply \( s_2 \) to the result else apply \( s_3 \) to the original term
- Do not backtrack to \( s_3 \) if \( s_2 \) fails!

?Call(\_,\_\_) < complex-transformation-for-call
+ ?BinOp(\_,\_,\_) < transformation-for-binop
+ other-cases

If Then Else (Sugar)

- Syntax: if \( s_1 \) then \( s_2 \) else \( s_3 \) end
- Equivalent to: \( \text{where}(s_1) < s_2 + s_3 \)
## Composing Strategies

### Identity
- **Syntax:** `id`
- **Always succeed**
- **Some laws**
  - `id ; s ≡ s`
  - `s ; id ≡ s`
  - `id <+ s ≡ id`
  - `s <+ id ≢ s`
  - `s₁ < id + s₂ ≡ s₁ <+ s₂`

### Failure
- **Syntax:** `fail`
- **Always fail**
- **Some laws**
  - `fail <+ s ≡ s`
  - `s <+ fail ≡ s`
  - `fail ; s ≡ fail`
  - `s ; fail ≢ fail`
**Composing Strategies**

<table>
<thead>
<tr>
<th>Identity</th>
<th>Failure</th>
</tr>
</thead>
<tbody>
<tr>
<td>Syntax: <code>id</code></td>
<td>Syntax: <code>fail</code></td>
</tr>
<tr>
<td>Always succeed</td>
<td>Always fail</td>
</tr>
<tr>
<td>Some laws</td>
<td>Some laws</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td><code>id ; s ≡ s</code></td>
<td><code>fail &lt;+ s ≡ s</code></td>
</tr>
<tr>
<td><code>s ; id ≡ s</code></td>
<td><code>s &lt;+ fail ≡ s</code></td>
</tr>
<tr>
<td><code>id &lt;+ s ≡ id</code></td>
<td><code>fail ; s ≡ fail</code></td>
</tr>
<tr>
<td><code>s &lt;+ id ≢ s</code></td>
<td><code>s ; fail ≢ fail</code></td>
</tr>
<tr>
<td><code>s₁ &lt; id + s₂ ≡ s₁ &lt;+ s₂</code></td>
<td></td>
</tr>
</tbody>
</table>

**Negation (Sugar)**

- Syntax: `not(s)`
- Fail if `s` succeeds, succeed if `s` fails
- Equivalent to: `s < fail + id`
Parameterized and Recursive Definitions

- Syntax: $f(x_1, \ldots, x_n | y_1, \ldots, y_m) = s$
- Strategy definition parameterized with strategies $(x_1, \ldots, x_n)$ and terms $(y_1, \ldots, y_m)$
- Note: definitions may be recursive
Parameterizing Strategies

Parameterized and Recursive Definitions

• Syntax: \( f(x_1, \ldots, x_n | y_1, \ldots, y_m) = s \)

• Strategy definition parameterized with strategies \((x_1, \ldots, x_n)\) and terms \((y_1, \ldots, y_m)\)

• Note: definitions may be recursive

\[
\begin{align*}
\text{try}(s) &= s \leftarrow \text{id} \\
\text{repeat}(s) &= \text{try}(s; \text{repeat}(s)) \\
\text{while}(c, s) &= \text{if } c \text{ then } s; \text{while}(c, s) \text{ end} \\
\text{do-while}(s, c) &= s; \text{if } c \text{ then } \text{do-while}(s, c) \text{ end}
\end{align*}
\]
Parameterizing Strategies

List transformations strategies

map(s) : [] -> []
map(s) : [x | xs] -> [<s> x | <map(s)> xs]

foldr(s1, s2, f) :
    [] -> <s1> []
foldr(s1, s2, f) :
    [x|xs] -> <s2>(<f> x, <foldr(s1, s2, f)> xs)

length = foldr(!0, add, !1)

reverse = reverse(|[]|)

reverse(|ys) : [] -> ys
reverse(|ys) : [x | xs] -> <reverse(|[x | ys]|)> xs
Summary

Basic transformations can be defined using a few basic operations:

- Term construction
- Pattern matching
- Sequential composition
- Choice
- Definitions
### Summary

Basic transformations can be defined using a few basic operations:

- Term construction
- Pattern matching
- Sequential composition
- Choice
- Definitions

### Next

- Term traversal
- Rewriting strategies
- Collecting strategies