From Pattern Matching to Simple Strategies
Program Transformation 2004–2005

Eelco Visser

Institute of Information & Computing Sciences
Utrecht University,
The Netherlands

February 15, 2005
Stratego is a language for program transformation based on term rewriting with programmable rewriting strategies.
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
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>Application</th>
<th>Void(), Call(t, t)</th>
</tr>
</thead>
<tbody>
<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

<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

<current term>

stratego> <strategy expression>

<transformed term>

stratego> <strategy expression>

command failed

http://www.strategoxt.org
Part I

Building and Matching Terms
2 Building and Matching Terms
   - Building Terms
   - Pattern Matching

3 Combining Match and Build
   - Sequential Composition
   - Term Variable Scope
   - Strategy Applications in Patterns

http://www.strategoxt.org
Building and Matching Terms

Atomic actions of program transformation:

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

http://www.strategoxt.org
Build Term

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

```
stratego> !Int("10")
Int("10")
```
### Build Term

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

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

- Syntax: \( \omega p \)
- Replace current term by instantiation of pattern \( p \)
- A pattern is a term with *meta-variables*

\[
\text{stratego}\geq :\text{binding } e \\
e \text{ is bound to } \text{Var("b")}
\]

\[
\text{stratego}\geq : \omega \text{Plus(Var("a"),e)} \\
\text{Plus(Var("a"),Var("b"))}
\]
**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

Plus(Var("a"),Int("3"))
stratego> ?e
Match Variable

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

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

```plaintext
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
```

http://www.strategoxt.org

**Pattern Matching**
Match Pattern

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

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

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

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

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

http://www.strategoxt.org
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

```
Plus(Var("a"),Int("3"))
stratego> ?Plus(e,_)  
stratego> :binding e  
e is bound to Var("a")
```
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
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")
```
Combining Match and Build

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
**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: \(?p_1; !p_2\)

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

stratego> (Plus(e1, e2) \rightarrow Plus(e2, e1))
Plus(Int("3"), Var("a"))
```
Apply and Match (Sugar)

- Syntax: $s \Rightarrow p$
- 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> $<\text{addS}>("1","2") \Rightarrow x$
"3"
stratego> :binding $x$
$x$ is bound to "3"
Term Variable Scope

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

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

```
strategy> (Plus(e1, e2) -> Plus(e2, e1))
Plus(Int("3"), Var("a"))
```

```
strategy> :binding e1
e1 is bound to Var("a")
```

```
strategy> \{e3, e4 : (Plus(e3, e4) -> Plus(e4, e3))\}
Plus(Var("a"), Int("3"))
```

```
strategy> :binding e3
e3 is not bound to a term
```

http://www.strategoxt.org
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: \((?p_1; \text{ where}(s); !p_2)\)

\[
\text{Plus(Int("14"),Int("3"))} \rightarrow \text{Int(k)} \text{ where } <\text{addS}>(i,j) \Rightarrow k \\
\text{Int("17")}
\]
Lambda Rules (Sugar)

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

\[
\text{Plus(\text{Int}("14"),\text{Int}("3"))}
\]
>
\[
\text{\\ Plus(\text{Int}(i),\text{Int}(j)) \rightarrow \text{Int}(k) \text{ where } \langle \text{addS}\rangle(i,j) \Rightarrow k \} \text{ Int("17")}
\]
>
:binding \( i \)
\( i \) is not bound to a term
>
:binding \( k \)
\( k \) is bound to "17"
Apply

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

\[\text{Plus}(\text{Int}("14"),\text{Int}("3")) \]
\[> \ \text{\textbackslash Plus}(\text{Int}(i),\text{Int}(j)) \rightarrow \text{Int}(<\text{addS}>(i,j)) \ \text{\textbackslash Int}("17")\]
Term Wrap

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

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

"foobar"

```
stratego> !Call(<id>, [])
Call("foobar", [])
```
Term Project

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

[1,2,3]
stratego> \(?[_]<id>\)
[2,3]

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

Composing Strategies
4 Strategy Definitions
- Naming Strategy Expressions
- Named Rewrite Rules
- Reusing Rewrite Rules

5 Composition of Transformation Strategies
- Choice
- Identity, Failure, and Negation
- Parameterized and Recursive Definitions
Strategy Definitions

Reuse of transformation requires definitions

1. Naming strategy expressions
2. Named rewrite rules
3. Reusing rewrite rules through modules
Simple Strategy Definition and Call

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

```
Plus(Var("a"),Int("3"))
stratego>  SwapArgs = \{ \( e1, e2 : (Plus(e1,e2) \rightarrow Plus(e2,e1)) \} \}
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 \( p_1, p_2, \text{ and } s \))

```
Plus(Var("a"),Int("3"))
stratego> SwapArgs : Plus(e1,e2) -> Plus(e2,e1)
stratego> SwapArgs
Plus(Int("3"),Var("a"))
```
Modules with Reusable Transformation Rules

module Simplification-Rules

rules

EvalPlus :
   Plus(Int(i), Int(j)) -> Int(k) where \langle addS\rangle(i, j) \Rightarrow 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 \langle not(eq)\rangle (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 :
$Plus(Plus(e_1, e_2), e_3) \rightarrow Plus(e_1, Plus(e_2, e_3))$

EvalPlus :
$Plus(Int(i), Int(j)) \rightarrow Int(k)$ where $<addS>(i, j) \Rightarrow k$

$Plus(Int("14"), Int("3"))$

stratego> PlusAssoc
command failed

stratego> PlusAssoc $\leftarrow$ EvalPlus

Int("17")

http://www.strategoxt.org

Pattern Matching
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!

\[
\text{?Call(\_,\_)} < \text{complex-transformation-for-call} \\
+ \text{?BinOp(\_,\_,\_)} < \text{transformation-for-binop} \\
+ \text{other-cases}
\]
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: where($s_1$) < $s_2 + s_3$
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

http://www.strategoxt.org
## 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`

## Negation (Sugar)
- **Syntax:** `not(s)`
- **Fail if** `s` **succeeds, succeed if** `s` **fails**
- **Equivalent to:** `s < fail + id`

[Link](http://www.strategoxt.org)
Parameterized and Recursive Definitions

- Syntax: \( f(x_1, \ldots, x_n \mid y_1, \ldots, y_m) = s \)
- Strategy definition parameterized with strategies \((x_1, \ldots, x_n)\) and terms \((y_1, \ldots, y_m)\)
- Note: definitions may be recursive
Parameterized and Recursive Definitions

- **Syntax:** \( f(x_1, \ldots, x_n \mid y_1, \ldots, y_m) = s \)

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

- Note: definitions may be recursive

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

map(s) =
    ?[] <+ \ [x | xs] -> [<s> x | <map(s)> xs] \\

foldr(s1, s2, f) =
    []; s1 <+ \\
    \ [x|xs] -> <s2>(<f> x, <foldr(s1, s2, f)> xs) \\

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

- Data-type specific traversal with congruence operators
- Strategic programming
- Application: evaluation using control-flow strategy
- Transformation tool composition