Traversing Strategies

Eelco Visser

Institute of Information & Computing Sciences
Utrecht University,
The Netherlands

February 16, 2005
Reading Assignments

Week 1 (infrastructure)
- Stratego/XT Tutorial: Part II

Week 2 (rules, strategies, traversals)
- Stratego/XT Tutorial: Chapters 16, 10, 11, 12, 13, 14, 15, 17
- FI’05 (dynamic rewrite rules): Sections 1, 2, 3

Week 3, lecture 5 (collecting strategies)
- Stratego/XT Tutorial: Chapter 18

Week 3, lecture 6 (binding and substitution)
- Article: Building Interpreters with Rewriting Strategies
- FI’05 (dynamic rewrite rules): Sections 4 and 5
Previously

From pattern matching to simple strategies

- First-order (prefix) terms to represent programs
- Basic operations for constructing and analyzing terms
- Strategy combinators for making complex transformations
- Strategy definitions for creating reusable transformations
- Rewrite rules and other syntactic abstractions
### Is Stratego typed?

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>The Stratego compiler does not guarantee well-formedness (according to a signature) of transformed terms</td>
</tr>
<tr>
<td>Yes</td>
<td>Programs can be considered dynamically typed</td>
</tr>
<tr>
<td></td>
<td>Stratego programs don’t crash because of type errors (pattern match failure is normal behaviour)</td>
</tr>
</tbody>
</table>

### Why Not?

- Historical reasons
- Generic traversals don’t fit in existing type systems (this lecture)
- Focus of research is on expressivity and applications
Stratego and Types

What are signatures good for?
- The compiler checks that constructors are declared with correct arity
- Documentation

Would a typesystem be useful?
- A typesystem would detect certain problems earlier
- Typesystems are only a partial solution to checking the safety of (meta) programs

Challenge
- Interested in type systems?
- Design a type system for Stratego that covers as many existing programs as possible
- (But don’t get too distracted from the course)
First class pattern matching is strong stuff; Use with caution

**Bad style**

Large strategy definitions with nested conditionals and lots of match and builds

\[
distr = ((\text{?And(Or(x, y), z) <+ ?And(z, Or(x, y))}); !\text{Or(<distr>And(x, z), <distr>And(y, z))) <+ id})
\]

**Good style**

- Named rewrite rules (\(L : p1 \rightarrow p2\)) that do one thing
- Simple strategies that combine named rules

\[
\text{DAOL : And(Or(x, y), z) -> Or(And(x, z), And(y, z))}
\]
\[
\text{DAOR : And(z, Or(x, y)) -> Or(And(z, x), And(z, y))}
\]
\[
distr = \text{topdown(try(DAOL <+ DAOR))}
\]
1. In control of rewriting
   motivation for separation of rules and strategies
2. Programmable rewriting strategies
   some typical idioms for using traversal strategies
3. Realizing term traversal
   how traversal strategies are constructed
Part I

In Control of Rewriting
Term Rewriting

- apply set of rewrite rules exhaustively

Advantages

- First-order terms describe abstract syntax
- Rewrite rules express basic transformation rules (operationalizations of the algebraic laws of the language.)
- Rules specified separately from strategy

Limitations

- Rewrite systems for programming languages often non-terminating and/or non-confluent
- In general: do not apply all rules at the same time or apply all rules under all circumstances
A Non-terminating Rewrite System

signature
  sorts Prop

constructors
  False : Prop
  True : Prop
  Atom : String -> Prop
  Not  : Prop -> Prop
  And  : Prop * Prop -> Prop
  Or   : Prop * Prop -> Prop

rules
  DAOL : And(Or(x, y), z) -> Or(And(x, z), And(y, z))
  DAOR : And(z, Or(x, y)) -> Or(And(z, x), And(z, y))
  DOAL : Or(And(x, y), z) -> And(Or(x, z), Or(y, z))
  DOAR : Or(z, And(x, y)) -> And(Or(z, x), Or(z, y))
  DN   : Not(Not(x))        -> x
  DMA  : Not(And(x, y))     -> Or(Not(x), Not(y))
  DMO  : Not(Or(x, y))      -> And(Not(x), Not(y))
Encoding Control with Recursive Rules

Common solution

- Introduce additional constructors that achieve normalization under a restricted set of rules
- Replace a ‘pure’ rewrite rule
  \[ p_1 \rightarrow p_2 \]
  with a functionalized rewrite rule:
  \[ f : p_1 \rightarrow p'_2 \]
  applying \( f \) recursively in the right-hand side
- Normalize terms \( f(t) \) with respect to these rules
- The function now controls where rules are applied
Recursive Rules: Map

Map

\[
\text{map}(s) : [] \rightarrow [] \\
\text{map}(s) : [x \mid xs] \rightarrow [<s> \mid <\text{map}(s)> \hspace{1mm} xs]
\]
Recursive Rewrite Rules: Constant Folding

Constant folding rules

Eval : Plus(Int(i), Int(j)) -> Int(<addS>(i,j))
Eval : Times(Int(i), Int(j)) -> Int(<mulS>(i,j))

Constant folding entire tree

fold : Int(i) -> Int(i)
fold : Var(x) -> Var(x)
fold : Plus(e1, e2) -> <try(Eval)>Plus(<fold>e1, <fold>e2)
fold : Times(e1, e2) -> <try(Eval)>Times(<fold>e1, <fold>e2)

Traversal and application of rules are tangled
Recursive Rewrite Rules: Disjunctive Normal Form

\[
\begin{align*}
dnf &: True &\rightarrow& True \\
dnf &: False &\rightarrow& False \\
dnf &: Atom(x) &\rightarrow& Atom(x) \\
dnf &: Not(x) &\rightarrow& \langle\text{not}\rangle(<\text{dnf}\rangle x) \\
dnf &: And(x,y) &\rightarrow& \langle\text{and}\rangle(<\text{dnf}\rangle x,<\text{dnf}\rangle y) \\
dnf &: Or(x,y) &\rightarrow& Or(<\text{dnf}\rangle x,<\text{dnf}\rangle y) \\
\text{and1} &: (\text{Or}(x,y),z) &\rightarrow& \text{Or}(\langle\text{and}\rangle(x,z),\langle\text{and}\rangle(y,z)) \\
\text{and2} &: (z,\text{Or}(x,y)) &\rightarrow& \text{Or}(\langle\text{and}\rangle(z,x),\langle\text{and}\rangle(z,y)) \\
\text{and3} &: (x,y) &\rightarrow& \text{And}(x,y) \\
\text{and} &= \text{and1} <+ \text{and2} <+ \text{and3} \\
\text{not1} &: \text{Not}(x) &\rightarrow& x \\
\text{not2} &: \text{And}(x,y) &\rightarrow& \text{Or}(\langle\text{not}\rangle(x),\langle\text{not}\rangle(y)) \\
\text{not3} &: \text{Or}(x,y) &\rightarrow& \langle\text{and}\rangle(\langle\text{not}\rangle(x),\langle\text{not}\rangle(y)) \\
\text{not4} &: x &\rightarrow& \text{Not}(x) \\
\text{not} &= \text{not1} <+ \text{not2} <+ \text{not3} <+ \text{not4}
\end{align*}
\]
Analysis

Functional encoding has two main problems

*Overhead* due to explicit specification of *traversal*

- A traversal rule needs to be defined for each constructor in the signature and for each transformation.

*Separation of rules and strategy is lost*

- Rules and strategy are completely *intertwined*
- Intertwining makes it more difficult to *understand* the transformation
- Intertwining makes it impossible to *reuse* the rules in a different transformation.
Analysis

Language Complexity

Traversal overhead and reuse of rules is important, considering the complexity of real programming languages:

<table>
<thead>
<tr>
<th>language</th>
<th># constructors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tiger</td>
<td>65</td>
</tr>
<tr>
<td>C</td>
<td>140</td>
</tr>
<tr>
<td>Java</td>
<td>140</td>
</tr>
<tr>
<td>COBOL</td>
<td>300–1200</td>
</tr>
</tbody>
</table>

Requirements

- Control over application of rules
- No traversal overhead
- Separation of rules and strategies
Part II

Programmable Rewriting Strategies
### Programmable Rewriting Strategies

- Select rules to be applied in specific transformation
- Select strategy to control their application
- Define your own strategy if necessary
- Combine strategies

### Idioms

- Cascading transformations
- One-pass traversal
- Staged transformation
- Local transformation
<table>
<thead>
<tr>
<th>Strategic Idioms</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rules for rewriting proposition formulae</td>
</tr>
</tbody>
</table>

**signature**

- sorts Prop
- constructors
  - False : Prop
  - True : Prop
  - Atom : String -> Prop
  - Not : Prop -> Prop
  - And : Prop * Prop -> Prop
  - Or : Prop * Prop -> Prop

**rules**

- DAOL : And(Or(x, y), z) -> Or(And(x, z), And(y, z))
- DAOR : And(z, Or(x, y)) -> Or(And(z, x), And(z, y))
- DOAL : Or(And(x, y), z) -> And(Or(x, z), Or(y, z))
- DOAR : Or(z, And(x, y)) -> And(Or(z, x), Or(z, y))
- DN : Not(Not(x)) -> x
- DMA : Not(And(x, y)) -> Or(Not(x), Not(y))
- DMO : Not(Or(x, y)) -> And(Not(x), Not(y))
## Strategic Idioms: Cascading Transformation

### Cascading Transformations
- Apply small, independent transformations in combination
- Accumulative effect of small rewrites

### Realization in Stratego

\[
simplify = \text{innermost}(R_1 \leftarrow \ldots \leftarrow R_n)
\]

### Example

Disjunctive normal form

\[
dnf = \text{innermost}(DAOL \leftarrow DAOR \leftarrow DN \leftarrow DMA \leftarrow DMO)
\]

And conjunctive normal form

\[
cnf = \text{innermost}(DOAL \leftarrow DOAR \leftarrow DN \leftarrow DMA \leftarrow DMO)
\]
One-pass Traversal

Apply rules in a single traversal over a program tree

Realization in Stratego

simplify1 = downup(repeat(R1 <+ ... <+ Rn))
simplify2 = bottomup(repeat(R1 <+ ... <+ Rn))

Example: Constant Folding

Eval : And(True, e) -> e
Eval : And(False, e) -> False
Eval : ...

eval = bottomup(try(Eval))
### Example: Desugarings

<table>
<thead>
<tr>
<th>DefN</th>
<th>Not(x)</th>
<th>Impl(x, False)</th>
</tr>
</thead>
<tbody>
<tr>
<td>DefI</td>
<td>Impl(x, y)</td>
<td>Or(Not(x), y)</td>
</tr>
<tr>
<td>DefE</td>
<td>Eq(x, y)</td>
<td>And(Impl(x, y), Impl(y, x))</td>
</tr>
<tr>
<td>DefO1</td>
<td>Or(x, y)</td>
<td>Impl(Not(x), y)</td>
</tr>
<tr>
<td>DefO2</td>
<td>Or(x, y)</td>
<td>Not(And(Not(x), Not(y)))</td>
</tr>
<tr>
<td>DefA1</td>
<td>And(x, y)</td>
<td>Not(Or(Not(x), Not(y)))</td>
</tr>
<tr>
<td>DefA2</td>
<td>And(x, y)</td>
<td>Not(Impl(x, Not(y)))</td>
</tr>
<tr>
<td>IDefI</td>
<td>Or(Not(x), y)</td>
<td>Impl(x, y)</td>
</tr>
<tr>
<td>IDefE</td>
<td>And(Impl(x, y), Impl(y, x))</td>
<td>Eq(x, y)</td>
</tr>
</tbody>
</table>

\[
desugar = \text{topdown}(\text{try}(\text{DefI} <+ \text{DefE}))
\]

\[
\text{impl-nf} = \text{topdown}(\text{repeat}(\text{DefN} <+ \text{DefA2} <+ \text{DefO1} <+ \text{DefE}))
\]
Staged Transformation

- Transformations are not applied to a subject term all at once, but rather in stages.
- In each stage, only rules from some particular subset of the entire set of available rules are applied.

Realization in Stratego

```plaintext
simplify =
    innermost(A1 <+ ... <+ Ak)
    ; innermost(B1 <+ ... <+ Bl)
    ; ...
    ; innermost(C1 <+ ... <+ Cm)
```
Local transformation

Apply rules only to selected parts of the subject program

Realization in Stratego

```
transformation =
  alltd(
    trigger-transformation
    ; innermost(A1 <+ ... <+ An)
  )
```
Part III

Realizing Term Traversal
### Realizing Term Traversal

#### Requirements

- Control over application of rules
- No traversal overhead
- Separation of rules and strategies

#### Many ways to traverse a tree

- Bottom-up
- Top-down
- Innermost
- ...

What are the primitives of traversal?
## Traversal Primitives

### One-Level Traversal Operators
Apply a strategy to one or more direct subterms

### Congruence: Data-Type Specific Traversal
Apply a different strategy to each argument of a specific constructor

### Generic Traversal

<table>
<thead>
<tr>
<th>All</th>
<th>Apply to all direct subterms</th>
</tr>
</thead>
<tbody>
<tr>
<td>One</td>
<td>Apply to one direct subterm</td>
</tr>
<tr>
<td>Some</td>
<td>Apply to as many direct subterms as possible, and at least one</td>
</tr>
</tbody>
</table>
Congruence Operators

Congruence Operator: Data-type Specific Traversal

- Syntax: \( c(s_1, \ldots, s_n) \)
  for each \( n \)-ary constructor \( c \)
- Apply strategies to direct sub-terms of a \( c \) term

```
Plus(Int("14"),Int("3"))
stratego> Plus(!Var("a"), id)
Plus(Var("a"), Int("3"))
```
Congruence Operators

**Congruence Operator: Data-type Specific Traversal**

- Syntax: \( c(s_1, \ldots, s_n) \)
  
  for each \( n \)-ary constructor \( c \)

- Apply strategies to direct sub-terms of a \( c \) term

```
Plus(Int("14"), Int("3"))
stratego> Plus(!Var("a"), id)
Plus(Var("a"), Int("3"))
```

**Example: List Operations**

```
map(s) = Nil + Cons(s, map(s)))

fetch(s) = Cons(s, id) <+ Cons(id, fetch(s))

filter(s) =
  Nil + (Cons(s, filter(s)) <+ ?Cons(_, <id>); filter(s))
```
Congruence Operators

**Congruence Operator: Data-type Specific Traversal**

- Syntax: \( c(s_1, \ldots, s_n) \)
  for each \( n \)-ary constructor \( c \)
- Apply strategies to direct sub-terms of a \( c \) term

**Example: List Operations**

\[
\text{map}(s) = [\] + [s \mid \text{map}(s)]
\]

\[
\text{fetch}(s) = [s \mid \text{id}] \leftarrow [\text{id} \mid \text{fetch}(s)]
\]

\[
\text{filter}(s) = \\
[\] + ([s \mid \text{filter}(s)] \leftarrow ?[\_\mid<\text{id}>]; \text{filter}(s))
\]
Example: Constant Folding

BinOp(PLUS(), Int("14"), Int("3"))

stratego> EvalBinOp
Int("17")
Example: Constant Folding

\[
\text{BinOp}(	ext{PLUS}(), \text{Int}("14"), \text{Int}("3"))
\]

\[
\text{stratego}\>\>\text{EvalBinOp}\>\>\text{Int}("17")
\]

\[
\text{const-fold} =
\begin{align*}
&\text{BinOp}() (= \text{id}, \text{const-fold}, \text{const-fold}); \text{try}() (\text{EvalBinOp}) \\
&\hspace{1em}<> \text{Call}() (= \text{id}, \text{map}() (\text{const-fold})); \text{try}() (\text{EvalCall}) \\
&\hspace{2em}<> \text{If}() (= \text{const-fold}, \text{const-fold}, \text{const-fold}, \text{const-fold}); \text{try}() (\text{EvalIf})
\end{align*}
\]
### Example: Constant Folding

```stratego
BinOp(PLUS(), Int("14"), Int("3"));
```
```
stratego> EvalBinOp
Int("17")
```

```stratego
cost-fold =
  BinOp(id, cost-fold, cost-fold); try(EvalBinOp)
  <+ Call(id, map(const-fold)); try(EvalCall)
  <+ If(const-fold, const-fold, const-fold, const-fold); try(EvalIf)
```

```stratego
BinOp(TIMES(),
  If(BinOp(PLUS(), Int("14"), Int("3")), Int("2"), Int("23")), Int("4"))
)
```
```
stratego> const-fold
Int("8")
```
Example: Format Checking

Format checking

- Describe a subset of a term language using a recursive pattern

Applications

- Verify output (testing)
- Verify input (pre-condition)
- Documentation
## Recursive Patterns

- Describe format of terms in normal form
- $\text{conj}(s) = \text{rec } x(\text{And}(x, x) <+ s)$
  - Conjunct
- $\text{disj}(s) = \text{rec } x(\text{Or}(x, x) <+ s)$
  - Disjunct
- $\text{conj-nf} = \text{conj}(\text{disj}(\text{Not}(\text{Var}(x)) + \text{Var}(x)))$
  - Conjunctive normal form
- $\text{disj-nf} = \text{disj}(\text{conj}(\text{Not}(\text{Var}(x)) + \text{Var}(x)))$
  - Disjunctive normal form
<table>
<thead>
<tr>
<th>Generic Traversal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data-type specific traversal requires tedious enumeration of cases</td>
</tr>
<tr>
<td>Even if traversal behaviour is uniform</td>
</tr>
<tr>
<td>Generic traversal allows concise specification of default traversals</td>
</tr>
</tbody>
</table>
### Visiting All Subterms

- **Syntax:** `all(s)`
- **Apply strategy** `s` to all direct sub-terms

```plaintext
Plus(Int("14"),Int("3"))
stratego> all(!Var("a"))
Plus(Var("a"),Var("a"))
```
Generic Traversal

Visiting All Subterms

- Syntax: `all(s)`
- Apply strategy `s` to all direct sub-terms

```latex
\text{Plus(Int("14"),Int("3"))}
\text{stratego> all(!Var("a"))}
\text{Plus(Var("a"),Var("a"))}
```

```latex
\text{bottomup(s) = all(bottomup(s)); s}
\text{topdown(s) = s; all(topdown(s))}
\text{downup(s) = s; all(downup(s)); s}
\text{alltd(s) = s <+ all(alltd(s))}
```
Visiting All Subterms

- Syntax: `all(s)`
- Apply strategy `s` to all direct sub-terms

```
Plus(Int("14"),Int("3"))
stratego> all(!Var("a"))
Plus(Var("a"),Var("a"))
```

```
bottomup(s) = all(bottomup(s)); s
topdown(s) = s; all(topdown(s))
downup(s) = s; all(downup(s)); s
alltd(s) = s <+ all(alltd(s))
```

```
const-fold =
    bottomup(try(EvalBinOp <+ EvalCall <+ EvalIf)))
```
Example: Desugaring Expressions

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

DefPlus : Plus(e1, e2) -> BinOp(PLUS(), e1, e2)

DesugarExp = DefAnd <+ DefPlus <+ ...

desugar = topdown(try(DesugarExp)

IfThen(
    And(Var("a"), Var("b")),
    Plus(Var("c"), Int("3")))

stratego> desugar
IfThen(
    If(Var("a"), Var("b"), Int("0")),
    BinOp(PLUS, Var("c"), Int("3")))
Fixed-point Traversal

\[ \text{innermost}(s) = \text{bottomup} (\text{try}(s; \text{innermost}(s))) \]

Normalization

\[ \text{dnf} = \text{innermost}(\text{DAOL} \leftrightarrow \text{DAOR} \leftrightarrow \text{DN} \leftrightarrow \text{DMA} \leftrightarrow \text{DMO}) \]

\[ \text{cnf} = \text{innermost}(\text{DOAL} \leftrightarrow \text{DOAR} \leftrightarrow \text{DN} \leftrightarrow \text{DMA} \leftrightarrow \text{DMO}) \]
## Generic Traversal: One

### Visiting One Subterms

- **Syntax:** `one(s)`
- **Apply strategy s to exactly one direct sub-terms**

<table>
<thead>
<tr>
<th>Plus(Int(&quot;14&quot;),Int(&quot;3&quot;))</th>
<th>stratego&gt; one(!Var(&quot;a&quot;))</th>
<th>Plus(Var(&quot;a&quot;),Var(&quot;a&quot;))</th>
</tr>
</thead>
</table>

```plaintext
stratego> one(!Var("a"))
```
Generic Traversal: One

Visiting One Subterms

- Syntax: `one(s)
- Apply strategy `s` to exactly one direct sub-terms

```
Plus(Int("14"),Int("3"))
stratego> one(!Var("a"))
Plus(Var("a"),Var("a"))
```

```
oncetd(s) = s <+ one(oncetd(s))
oncebu(s) = one(oncebu(s)) <+ s
spinetd(s) = s; try(one(spinetd(s)))
spinebu(s) = try(one(spinebu(s))); s
```
Generic Traversal: One

### Visiting One Subterms

- **Syntax:** `one(s)`
- **Apply strategy s to exactly one direct sub-terms**

```plaintext
Plus(Int("14"), Int("3"))
stratego> one(!Var("a"))
Plus(Var("a"), Var("a"))
```

```plaintext
oncetd(s) = s <+ one(oncetd(s))
oncebu(s) = one(oncebu(s)) <+ s
spinetd(s) = s; try(one(spinetd(s)))
spinebu(s) = try(one(spinebu(s))); s
```

```plaintext
contains(|t) = oncetd(?t)
```
## Generic Traversal: One

### Visiting One Subterms

- **Syntax:** `one(s)`
- **Apply strategy** `s` to exactly one direct sub-terms

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Plus(Int(&quot;14&quot;), Int(&quot;3&quot;))</code></td>
<td>Example of Plus operation</td>
</tr>
<tr>
<td><code>stratego&gt; one(!Var(&quot;a&quot;))</code></td>
<td>Example of applying strategy</td>
</tr>
<tr>
<td><code>Plus(Var(&quot;a&quot;), Var(&quot;a&quot;))</code></td>
<td>Example of Plus operation with variables</td>
</tr>
<tr>
<td><code>oncetd(s) = s &lt;+ one(oncetd(s))</code></td>
<td>Definition of <code>oncetd</code></td>
</tr>
<tr>
<td><code>oncebu(s) = one(oncebu(s)) &lt;+ s</code></td>
<td>Definition of <code>oncebu</code></td>
</tr>
<tr>
<td><code>spinetd(s) = s; try(one(spinetd(s)))</code></td>
<td>Definition of <code>spinetd</code></td>
</tr>
<tr>
<td><code>spinebu(s) = try(one(spinebu(s))); s</code></td>
<td>Definition of <code>spinebu</code></td>
</tr>
<tr>
<td>`contains(</td>
<td>t) = oncetd(?t)`</td>
</tr>
<tr>
<td><code>reduce(s) = repeat(rec x(one(x) + s))</code></td>
<td>Definition of <code>reduce</code></td>
</tr>
<tr>
<td><code>outermost(s) = repeat(oncetd(s))</code></td>
<td>Definition of <code>outermost</code></td>
</tr>
<tr>
<td><code>innermostI(s) = repeat(oncebu(s))</code></td>
<td>Definition of <code>innermost</code></td>
</tr>
</tbody>
</table>
Generic Traversal: Some

Visiting some subterms (but at least one)

- Syntax: `some(s)`
- Apply strategy `s` to as many direct subterms as possible, and at least one

```
Plus(Int("14"), Int("3"))
stratego> some(?Int(_); !Var("a"))
Plus(Var("a"), Var("a"))
```
Generic Traversal: Some

Visiting some subterms (but at least one)

- Syntax: some(s)
- Apply strategy $s$ to as many direct subterms as possible, and at least one

```
Plus(Int("14"),Int("3"))
stratego> some(?Int(_); !Var("a"))
Plus(Var("a"),Var("a"))
```

One-pass traversals

```
sometd(s) = s <+ some(sometd(s))
somebu(s) = some(somebu(s)) <+ s
```
## Generic Traversal: Some

### Visiting some subterms (but at least one)

- **Syntax:** `some(s)`
- **Apply strategy** `s` to as many direct subterms as possible, and at least one

```stratego
Plus(Int("14"),Int("3"))
stratego> some(?Int(_); !Var("a"))
Plus(Var("a"),Var("a"))
```

### One-pass traversals

- `sometd(s) = s <+ some(sometd(s))`
- `somebu(s) = some(somebu(s)) <+ s`

### Fixpoint traversal

- `reduce-par(s) = repeat(rec x(some(x) + s))`
Summary

- Tangling of rules and strategy (traversal) considered harmful
- Separate traversal from rules
- One-level traversal primitives allow wide range of traversals

Next

- Collecting strategies