Combining Aspect-Oriented and Strategic Programming

Stratego User Days 2005

3. May
Utrecht, The Netherlands

Karl Trygve Kalleberg$^{1,2}$
Eelco Visser$^2$

$^1$University of Bergen
Norway

$^2$Utrecht University
The Netherlands
Outline

- Introduction
- Aspects in Stratego
- Case Studies
- Discussion
- Conclusion
Aspect-Oriented Programming

• Goal of AOP is the modularization of separate and cross-cutting concerns.
  - concern – here: a non-functional property;
  - cross-cutting – spread across and repeatedly expressed in many program units;
  - separate – (most often) independent of each other.

• One concern maps to one aspect, examples
  - security, persistence, exception handling, traceability.

• Modularization – one concept = one module – not particular to any programming paradigm.
Stratego

• A term rewriting language where the application of the term rewriting *rules* are controlled by *user-definable strategies*.

• Language primitives:
Stratego

- A term rewriting language where the application of the term rewriting *rules* are controlled by user-defined *strategies*.

- Language primitives:
  - constructors
    - `Var : Id -> Var`
    - `Var : Var -> Exp`
    - `Int : String -> Exp`
    - `Plus : Exp * Exp -> Exp`
    - `If : Exp * Exp * Exp -> Exp`
    - `While : Exp * Exp -> Exp`
    - `Assign : Var * Exp -> Exp`
Stratego

- A term rewriting language where the application of the term rewriting *rules* are controlled by user-definable *strategies*.

- Language primitives:
  - constructors
  - rules

```plaintext
rules
EvalBinOp:
Plus(Int(i), Int(j)) -> Int(k)
where <addS>(i,j) => k

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

• A term rewriting language where the application of the term rewriting rules are controlled by user-definable strategies.

• Language primitives:
  – constructors
  – rules
  – strategies

```prolog
strategies
prop-const = PropConst
  <+ prop-const-assign
  <+ prop-const-if
  <+ prop-const-while
  <+ (all(prop-const)
       ; try(EvalBinOp)
  )
```
Stratego

- A term rewriting language where the application of the term rewriting rules are controlled by user-definable strategies.
- Language primitives:
  - constructors
  - rules
  - strategies
- Library provides many common strategies, such as topdown, bottomup and innermost.
Stratego/XT

• A distribution of Stratego, providing
  – a *component model* (XTC);
  – a *modular syntax definition formalism* (SDF) with *parser* (SGLR);
  – a *library* of generic strategies for program transformation, including
    • various term traversals;
    • data and control flow analysis;
    • command-line option parsing.
Example: Constant Propagation

- Problem: We want to (a) replace variables with their constant values when known, and (b) simplify expressions involving constants.
  - (strictly speaking, only (a) is constant propagation)
Example: Constant Propagation (2)

```
module prop-const
signature
  constructors
    Var  : Id -> Var
    : Var -> Exp
    Int  : String -> Exp
    Plus : Exp * Exp -> Exp
    If   : Exp * Exp * Exp -> Exp
    While : Exp * Exp -> Exp
    Assign : Var * Exp -> Exp
rules
  EvalBinOp : Plus(Int(i), Int(j)) -> Int(k) where <addS>(i,j) => k
  EvalIf    : If(Int("0"), e1, e2) -> e2
strategies
  prop-const = PropConst <+ prop-const-assign <+ prop-const-if
               <+ prop-const-while <+ (all(prop-const) ; try(EvalBinOp))

  prop-const-assign = Assign(?Var(x), prop-const => e)
                      ; if <is-value> e then rules( PropConst : Var(x) => e )
                      else rules( PropConst :~ Var(x) ) end

  prop-const-if = If(prop-const, id, id)
                  ; (EvalIf ; prop-const <+ (If(id,prop-const,id) /PropConst\ If(id,id,prop-const)))

  prop-const-while = ?While(e1, e2) ; (While(prop-const,id)
                                  ; EvalWhile <+ (/PropConst\* While(prop-const, prop-const)))
```
AspectStratego

• An extension to Stratego, providing **pointcuts** and **advice** which together offer a type of **code composition**.

• Properties
  
  – declarative;
  
  – a strict superset of Stratego – every valid Stratego program is a valid AspectStratego program;
  
  – allows modularization of some cross-cutting concerns.

  • but not timing constraints, synchronization, ...
Pointcuts

- **Pointcut** $\equiv$ an expression on logical *combinators* and *joinpoint (context) predicates* which identifies a joinpoint.

- **Joinpoint** $\equiv$ a *point in the program execution* where control-flow passes twice: into and out of the subcomputation at that point.

- Pointcuts are used to *identify locations* in the program, based on *static* (structural) and *dynamic* (runtime) properties.
Pointcuts (2)

```plaintext
pointcut log-rule-or-strat(n, y) =
  (rules(P* => n) + strategies(p* => n)) ; args(y)
```
Combining Aspect-Oriented and Strategic Programming

Pointcuts (2)

pointcut log-rule-or-strat(n, y) =

(rules(P* => n) + strategies(p* => n)) ; args(y)

joinpoint predicates combiners joinpoint context predicate
Pointcuts (3)

• **Joinpoint predicates**
  – Static (structure) predicates
  – `calls(name-expr => n)` – a strategy or rule invocation
  – `strategies(...)` – a strategy definition
  – `rules(...)` – a rule definition
  – `matches(pattern => t)` – a pattern match
  – `builds(...)` – a term construction
  – `fails` – explicit invocation of `fail`
Pointcuts (4)

• *Joinpoint context predicates*
  
  - (potentially) dynamic (runtime) predicates

• *Modifiers to joinpoint predicates*
  
  - `withincode(name-expr => n)` – joinpoint within a rule or strategy
  
  - `matchflow(flow-regex => t)` – joinpoint with a given execution history
  
  - `args(n_0, n_1, ..., n_n)` – joinpoint with n arguments
  
  - `rhs / lhs(pattern => t)` – joinpoint with given left-hand side
Pointcuts (5)

- **Combinators**
  - ; (and)
  - + (or)
  - `not ()`

```plaintext
pointcut
log-rule-or-strat(n, y) =
  (rules(P* => n) + strategies(p* => n)) ; args(y)
```
Pointcuts (6)

pointcut
log-rule-or-strat(n, y) =

(rules(P* => n) + strategies(p* => n)) ; args(y)
Advice

- *Advice* $\triangleq$ a *body of code* $c$ (a strategy expression) associated with a pointcut $p$.
- Advice is used to insert code into the joinpoint identified by the pointcut $p$.
- Three main kinds of advice:
  - *before* – $c$ is inserted before the joinpoint of $p$
  - *after* – $c$ is inserted after the joinpoint of $p$
  - *around* – $c$ is inserted around the joinpoint of $p$
  - use *proceed* to identify the “hole” for $p$
Advice (2)

• Three kinds of after advice
  – after succeed – run iff the joinpoint code succeeded
  – after fail – run iff the joinpoint code failed
  – after – run after joinpoint code, regardless of success

• after succeed and after fail may change success to failure or vice versa. After may not.
strategies

invoked(|s) = ![ "Invoked '"', s, "'" ]
failed(|s) = ![ "Failed '"', s, "'" ]
finished(|s) = ![ "Finished '"', s, "'" ]

prop-const = PropConst <+ prop-const-assign
PropConst = id

aspects

pointcut rule-or-strat(n, y) =
  (rules(P* => n) + strategies(p* => n)) ; args(y)

aspect p-logger =
before : rule-or-strat(s, t) = log(|Debug, <invoked(|s)>)
after fail : rule-or-strat(s, t) = log(|Debug, <failed(|s)>)
after : rule-or-strat(s, t) = log(|Debug, <finished(|s)>)
Weaving

- **Weaving** $\overset{\text{def}}{=} \text{the act of composing advice code into the joinpoints picked out by their pointcuts.}

- Conceptually, the procedure is:

```plaintext
foreach aspect as 
foreach advice ad 
    traverse program { 
        if pointcut of ad matches 
            insert code of ad 
    }
```
Weaving (2)

pointcut
  \( \text{rules}(P^* \Rightarrow n) + \text{strategies}(p^* \Rightarrow n) \) ; \( \text{args}(y) \)

\[ + \]

before :
  \( \text{rule-or-strat}(s, t) = \log(\text{Debug}, <\text{invoked}(|s|)>) \)

on

\( \text{prop-const} = \text{PropConst} \leftrightarrow \text{prop-const-assign} \)
Weaving (3)

- Decomposing pointcuts:

\[
\text{rules}(P^* \Rightarrow n) + \text{strategies}(p^* \Rightarrow n) ; \text{args}(y)
\]

\[
\text{rules}(P^* \Rightarrow n) ; \text{args}(y)
\]

\[
\text{strategies}(p^* \Rightarrow n) ; \text{args}(y)
\]
Weaving (4)

```
strategies(p* => n) ; args(y)

rule-or-strat(s, t) = log(|Debug, <invoked(|s|)>)

prop-const = PropConst <+ prop-const-assign

↓

prop-const =
  log(|Debug, <invoked(|"prop-const"|)>)
; PropConst <+ prop-const-assign
```
Weaving (5)

• All weaving is done on the template:

```plaintext
before ;
if pointcut-code then
  if after-succeed then try(after) else after ; fail end
else
  if after-fail then try(after) else after ; fail end
end
```

• Idea: *after succeed* and *after fail* can change success/failure, *after* cannot.
Example: Algorithm Skeletons

- Problem: *We have a generic strategy for forward propagating transformations, and we want to instantiate it.*

- Library implementor
  - *Generalize constant propagator into forward propagator*
  - *Provide instantiation aspect.*

- Library user
  - *Implements concrete instantiation aspect.*

- Weaver produces *constant propagator.*
Example: Algorithm Skeletons (2)

```plaintext
module forward-prop
strategies

forward-prop =
    fail <+ prop-assign <+ prop-if <+ prop-while <+ all(forward-prop)

prop-assign =
    Assign(?Var(x), forward-prop => e) ; id

prop-if =
    If(forward-prop, id, id)
    ; (If(id,forward-prop,id) \ If(id,id,forward-prop))

prop-while =
    ?While(e1, e2)
    ; (/* While(forward-prop, forward-prop))
```

- Skeleton follows *subject language*; common to all *forward propagating transformations*. 
These pointcuts identify the variation points of the skeleton. By filling in the variation points using advice, the skeleton is instantiated.
Example: Type Checking of Terms

- Problem: We want to check the validity of a term with respect to a signature by inserting format checking code.
  - In Stratego, terms are “free form”, only numerical arity must be respected.
  - The Stratego/XT suite contains format checker generators, but they are intended to be run only after a transformation is complete.
  - We can use aspects to insert typechecking code for us, and have detailed control of its insertion.
Example: Type Checking of Terms

module typecheck-example
aspects

pointcut typecheck-rules(n, t) = rules(n) ; rhs(t)

aspect typechecker =
  around : typecheck-rules(n, t) =
  proceed
  ; (typecheck(|t)
  <+ (log(|incorrect-term(n)) ; fail)
  )
Discussion

- General *meta-programming, extensible interpreters* and *reflection* are strictly more powerful than aspects, but also more complicated.
  - We are questing for a linguistic “*sweet spot*” where we can still do useful code composition.

- Even with its apparent limitations, AspectStratego provides a mechanism for *extension* and *adaptation* in addition to *modularization*. 
Discussion (2)

• Experience from implementation and use tells us:
  – Our aspects are most likely *too limited*
    • We wanted to use aspects for *origin tracking*, but we have not (yet) come up with a good aspect-based solution.
    • *Introspection* offered in the current version is insufficient – more *reflective capabilities* of Stratego needed.
      – E.g. querying a strategy for its arity, module, etc.
    – A Stratego *interpreter* or *staged programming facilities* would be useful as a mechanism for implementing the aspect language compiler.
Conclusion

• Presented a *declarative extension* to Stratego and its *concrete application* to logging, typechecking of terms and *algorithm adaptation*.

• Compared our language extension to other language techniques for *modularization, extension* and *adaptation*.