1 Motivation

1.1 Review [4, 7]

• Quantification

\textbf{Defn. 1.1 (Quantified Statements)} have an effect on many places in the program

as opposed to “in the underlying code”, which is biased toward the base + aspects model

• Obliviousness

\textbf{Defn. 1.2 (Obliviousness)} the execution of cross-cutting code $A$ without any reference to $A$ from the client code that $A$ cross-cuts

– semantic interaction
– without syntactic coupling

• Modular Reasoning

Understanding a module $M$ based on:

– the code \textit{in} $M$,
– the code \textit{surrounding} $M$, and
– the \textit{signature} and \textit{specification} of any modules referred to by that code.

• Behavioral Subtyping Analogy
– Behavioral subtyping in OOP:
  an overriding method must **satisfy the specification of the overridden method**
– Behavioral subtyping is a **discipline**
  * It places constraints on **the subtype programmer**
  * It provides the benefit of modular reasoning for **clients**
– What about AOP?
**Q:** Can a language have quantification and obliviousness **and** allow modular reasoning?

*It isn’t clear.*

**Q:** Is there a discipline like behavioral subtyping that would allow modular reasoning in aspect-oriented programming languages? in AspectJ?

### 1.2 Spectators and Assistants [3]

- **Assistants**
  - can change the behavior of **advised code**
  - must be explicitly accepted by either
    * the module containing the advised join points, (all clients see the effects)
    * or a client of that module (only that client sees the effects)

- **Spectators**

**Defn. 1.3** A spectator is an aspect that "does not change the behavior of any other module."

**Q:** What might that mean? What is “spectator-ness”?

– Safety and Liveness [10]

**Defn. 1.4** A safety property says that **nothing bad happens**

**Defn. 1.5** A liveness property says that **eventually something good happens**

* Before-advice that immediately went into an infinite loop would be **safe but not live**
* Before-advice that deleted all the files on your hard drive and then proceeded to the original method would be **live but not safe**

– Spectators and Safety

Some possible interpretations:

* A spectator cannot **modify any state but its own**
A spectator cannot violate the specification of advised modules

Q: Is it that simple? Are there any problems with these notions?

What about I/O?
Can we modularly find all the advised modules? What about quantification?

– Spectators and Liveness
Goal: Spectators must always allow the advised method to execute with its original arguments and must return the result unchanged.

Q: Is this decidable?
No! by reduction from the halting problem.

What if we:

* Restrict control flow constructs in spectator advice to make the problem decidable?
  
  Q: What constructs could we allow? loops? method calls? mathematical expressions?

* Run spectators in a separate thread?
  
  Q: What if advice isn’t finished before advised method is called again?

* Approximate by prohibiting spectators from using around-advice or throwing checked exceptions?

• Do you buy it? (Direct discussion towards needing formal proof.)

  – Which of these notions of “spectator-ness” could be statically enforced? All but the specification safety property (and perhaps that could be if the specifications were sufficiently restricted).

  – Do spectators and assistants provide modular reasoning? How do we know?
  – Can we implement reasonable aspect-oriented programs under these restrictions?

1.3 Why formal semantics?

Defn. 1.6 A formal semantics is a mathematically complete description of a programming language

• Makes proofs about language properties tractable

• Lingua franca of programming language researchers
1.4 Why core calculi?

Defn. 1.7 A core calculus is a programming language stripped of all but its essential elements

Q: What is “essential”? Depends on the problem
A core calculus:

- Eliminates “noise”
- Makes construction of complete formal semantics tractable
- Can be used to define user-level languages
- Examples
  - \( \lambda \) calculus and Haskell
  - Object calculus and Smalltalk
  - Parameterized aspect calculus and AspectJ?

2 Introduction to Formal Semantics

2.1 Kinds of Formal Semantics

Example: the semantics of a while loop

- Denotational [9]
  - Strength: proving properties about the language
  - Map values in language to mathematical entities, like \( \{ T, F \} \) or the natural numbers
  - Model operations in language as mathematical operations, like \( \land, \neg, or + \)
  - Example:

\[
\begin{align*}
\text{\([\text{while } E \text{ do } C;]_s = w(s), \text{ where } w(s) = if (\llbracket E \rrbracket_s, w(\llbracket C \rrbracket_s), s)\]}\end{align*}
\]

\( s \) is the state, typically a mapping from variables to values
Read double brackets as “the meaning of foo in the state \( s \”).
\( w \) is recursive

\( \llbracket \) is overloaded:
- \( \llbracket E \rrbracket_s : boolean \)
- \( \llbracket C \rrbracket_s : state \)
- Q: what is the type of the if function?
  \( \text{A: if: Boolean \times State \times State \rightarrow State} \)

- Axiomatic [2]
- Strength: proving properties about actual programs
- Map values in language to mathematical entities
- Describe operations using logical assertions, for example pre- and post-conditions and loop invariants
- Uses Hoare triples: \( \{ P \} C \{ Q \} \)
  * \( P \) is a pre-condition
  * \( Q \) is a post-condition
  * For two states \( s \) and \( s' \) we write:

\[
(\{ P \} C \{ Q \}) \iff [P]_s \land ([C] = s') \land [Q]_{s'}
\]

We say "the Hoare triple \( \{ P \} C \{ Q \} \) is valid for the pair of states \( (s, s') \)."

- Example:

\[
\{ I \land E \} C \{ I \} \\
\{ I \} \text{while } E \text{ do } C; \{ I \land \neg E \}
\]

\( I \) is the loop invariant

Typically the rule used is actually:

\[
P \Rightarrow I \quad \{ I \land E \} C \{ I \} \quad (I \land \neg E) \Rightarrow Q \\
\{ P \} \text{while } E \text{ do } C; \{ Q \}
\]

- Operational

  - Strength: clarity, guides implementation, proving behavioral properties of the language
  - Values in language represent themselves (typically)
  - Operations are described by rewrite rules that reduce a term to a new term, given that a set of premises is satisfied.

General form:

\[
\begin{array}{c}
\text{premise}_1 \quad \ldots \quad \text{premise}_n \\
\hline
\text{Env} \vdash a \rightsquigarrow b
\end{array}
\]

\( \text{Env} \) is an environment
\( a \) and \( b \) might be terms, or might be sequences describing the state of some virtual machine (e.g., term + state)
Two sorts of operational semantics

* Small Step: a sub-term of \( a \) is replaced with a new sub-term to form \( b \) rules chain horizontally.

Example:
The semantics of the if statement is:

\[
\vdash \text{if true then } C_0 \text{ else } C_1 \cdot s \rightarrow C_0 \cdot s
\]

\[
\vdash \text{if false then } C_0 \text{ else } C_1 \cdot s \rightarrow C_1 \cdot s
\]

\[
\vdash E \cdot s \rightarrow E' \cdot s'
\]

\[
\vdash \text{if } E \text{ then } C_0 \text{ else } C_1 \cdot s \rightarrow \text{if } E' \text{ then } C_0 \text{ else } C_1 \cdot s'
\]

and the semantics of statement sequencing is:

\[
\vdash \text{skip; } C_1 \cdot s \rightarrow C_1 \cdot s
\]

\[
\vdash C_0 \cdot s \rightarrow C_0' \cdot s'
\]

\[
\vdash C_0; C_1 \cdot s \rightarrow C_0'; C_1 \cdot s'
\]

Using these, the semantics of the while statement is [8]:

\[
\vdash \text{while } E \text{ do } C; \cdot s \rightarrow \text{if } E \text{ then } C; \text{ while } E \text{ do } C; \text{ else skip } \cdot s
\]

Reduction terminates with \((\text{skip, s})\).

* Big Step (a.k.a. “natural”): \( a \) is reduced to a value in one (big) step rules stack vertically.

Sometimes when people (e.g., Abadi and Cardelli) say “operational semantics”, they mean big step.

Example:

\[
\vdash E \cdot s \leadsto \text{false } \cdot s'
\]

\[
\vdash \text{while } E \text{ do } C; \cdot s \leadsto s'
\]

\[
\vdash E \cdot s \leadsto \text{true } \cdot s_e \quad \vdash C \cdot s_e \leadsto s' \quad \vdash \text{while } E \text{ do } C; \cdot s' \leadsto s''
\]

\[
\vdash \text{while } E \text{ do } C; \cdot s \leadsto s''
\]

The result of reducing a statement is just the state. Reducing an expression just yields a value, assuming expressions cannot have side effects.

Other kinds of formal semantics

- Labelled transition systems (enhancement of small step op sem)
- Chemical semantics
2.2 Operational semantics for the $\lambda$ calculus

- Small step semantics (review, but in Abadi and Cardelli format)
  - Rules
    - Top-level, one-step reduction omitting alpha and eta rules
      \[ \beta \]
      \[ \vdash ((\lambda x.e) e') \rightarrow e \{ x \leftarrow e' \} \]
      A&C substitution style, and sometimes the $x$ is omitted
    - One-step reduction
      **Defn. 2.1** A context $C[-]$ is a term with a single hole. $C[e]$ represents the result of filling the hole with the term $e$ (possibly capturing free variables of $e$).
      \[ \vdash e \rightarrow e' \quad C[-] \text{ is any context} \]
      \[ \vdash C[e] \rightarrow C[e'] \]
    - Many-step reduction
      $\rightarrow$ is the reflexive transitive closure of $\rightarrow$
    - Example
      \[ \vdash ((\lambda z.2) 2) \rightarrow 2 \quad C[-] = ((\lambda y.3) -) \]
      \[ \vdash ((\lambda y.3) ((\lambda z.2)) \rightarrow ((\lambda y.3) 2) \quad \vdash ((\lambda y.3) 2) \rightarrow 3 \quad C[-] = - \]
      \[ \vdash ((\lambda y.3) ((\lambda x.x)) (\lambda x.x))) \]
      Rules chain horizontally
  - Non-deterministic:
    \[ ((\lambda y.3) ((\lambda x.x)) (\lambda x.x))) \]
    Can be made deterministic by restricting the shape of contexts.
    - Normal order: $C[-] ::= - \mid (C[-] e)$
    - Applicative order?
      Need a notion of values
      $C[-] ::= - \mid (v \ C[-]) \mid (C[-] e)$
      Need to restrict the $\beta$ rule to reduce only terms of the form $((\lambda x.e) v)$. 

• Big step semantics
  – Judgment: $\vdash e \leadsto v$
    The term $e$ reduces to the value $v$
  – Values
    * $\lambda$ terms, $(\lambda x.e)$
    * free variables
  – Rules

\[
\begin{align*}
\quad & \beta \\
\vdash e \{ x \leftarrow e' \} \leadsto v & \quad \vdash e \leadsto e' \quad \vdash (v \ e') \leadsto v & e \text{ is not a value} & \quad \vdash v \leadsto v \\
\vdash ((\lambda x.e) \ e') \leadsto v & \quad \vdash (e \ e') \leadsto v & \\
\quad & \text{RATOR} \quad \text{VAL} \quad \text{RATOR}
\end{align*}
\]

Q: Do these rules describe applicative order? normal order? some other order? **normal order**

**Homework:** Give the big step semantics for applicative order reduction. E.C.: implement interpreter based on big step semantics

– Examples

\[
\begin{align*}
\quad & \vdash 3 \leadsto 3 \\
\vdash ((\lambda y.3) \ ((\lambda z.2))) \leadsto 3 & \beta
\end{align*}
\]

Let them work out this one:

\[
\begin{align*}
\quad & \vdash (\lambda y.3) \leadsto (\lambda y.3) & \quad \vdash 3 \leadsto 3 \\
\vdash ((\lambda x. (\lambda y.3)) \ ((\lambda z.2))) \leadsto (\lambda y.3) & \beta & \quad \vdash ((\lambda y.3) \ ((\lambda z.2))) \leadsto 3 & \beta \\
\quad & \text{VALUE} & \quad \text{VALUE} & \quad \text{RATOR}
\end{align*}
\]

– Q: Is this semantics deterministic?
  Yes, because only one rule is applicable to any term.

• Abadi and Cardelli Proof Style [1, pp. 79–80]

\[
\begin{align*}
\vdash & \quad \text{(RULE 2)} \\
\vdash \quad \text{(RULE 3)} \\
\quad & \text{REASON} \quad \text{(RULE 5)}
\end{align*}
\]

Example:
2.3 Untyped Object Calculus, \( \varsigma \)

- **Syntax**

  \[
  \begin{align*}
  \text{variables} & \quad x \in Vars \\
  \text{labels} & \quad l \in Labels \\
  \text{terms} & \quad a, b, c ::= x \\
  & \quad | \quad \left[ l_i = \varsigma(x_i) \right] \ b_i \ \ i \in I \\
  & \quad | \quad a.l \\
  & \quad | \quad a.l \leftarrow \varsigma(x)
  \end{align*}
  \]

- **Big step semantics** (omitting small step semantics due to limited time)

**Homework:** Implement a stack object using the object calculus

- **Object:** a set of pairs of labels and methods

\[
\text{RED OBJECT} \\
\vdash [l_i = \varsigma(x_i) \ b_i \ i \in I] \leadsto [l_i = \varsigma(x_i) \ b_i \ i \in I]
\]

Example: \([\text{pos} = \varsigma(x).n, n = \varsigma(x)2]\), where 2 is shorthand for an object that represents the natural number 2.

- **Method Selection:** reduces the body of the named method, substituting object for the self parameter

\[
\text{RED SELECT} \\
\vdash a \leadsto [l_i = \varsigma(x_i) \ b_i \ i \in I] \quad \vdash b_j \left\{ x_j \leftarrow [l_i = \varsigma(x_i) \ b_i \ i \in I] \right\} \leadsto v \quad j \in I \\
\vdash a.l_j \leadsto v
\]

Example: \([\text{pos} = \varsigma(x).n, n = \varsigma(x)2], \text{pos}\)
– Method update: generates a new object, with the given method replacing the named method

\[
\text{Red Update} \quad \vdash a \leadsto [l_i = \varsigma(x)b_{i \in I}] \quad j \in I
\]
\[
\vdash a.l_j \leftarrow \varsigma(x)b \leadsto [l_j = \varsigma(x)b, l_i = \varsigma(x)b_{i \in I\setminus j}]
\]

Q: What’s the result of reducing this term: \([\text{pos}=\varsigma(x).n, n=\varsigma(x)2].n \leftarrow \varsigma(x)3\]
A: \([\text{pos}=\varsigma(x).n, n=\varsigma(x)3]\]

Q: What about this one: \([\text{pos}=\varsigma(x).n, n=\varsigma(x)2].\text{pos} \leftarrow \varsigma(x).n.\text{succ}\]
A: \([\text{pos}=\varsigma(x).n.\text{succ}, n=\varsigma(x)2]\]

Q: What happens if we select \text{pos} on the result?
A: 3, assuming \(2.\text{succ} \leadsto 3\)

• Syntactic sugar

– Fields: methods in which the self parameter does not appear free

\([\text{pos}=\varsigma(x).n, n=2]\) desugars to \([\text{pos}=\varsigma(x).n, n=\varsigma(y)2]\) where y is not free in 2

\([\text{pos}=\varsigma(x).n, n=2].n := 3\) desugars to \([\text{pos}=\varsigma(x).n, n=3]\)

– Lambda expressions

Can translate untyped λ calculus into the \(\varsigma\) calculus.

Let \(\{\}\) map λ calculus to \(\varsigma\) calculus as follows:

\[
\begin{align*}
\langle x \rangle &= x \\
\langle (e_1 e_2) \rangle &= (\langle e_1 \rangle).\text{arg}=\langle e_2 \rangle).\text{val} \\
\langle (\lambda x.e) \rangle &= [\text{arg} = 0, \text{val} = \varsigma(s)\langle e \rangle \{x \leftarrow s.\text{arg}\}] 
\end{align*}
\]

**Homework:** Translate some lambda calculus expressions and reduce them in the object calculus

3 Parameterized Aspect Calculus, \(\varsigma_{asp}\) [5, 6]

3.1 Changes vs. the object calculus

Object calculus plus aspects plus constants

• Join point abstraction

– Each reduction step triggers a search for advice

– Search uses a four-part abstraction of the reduction step

  * Reduction kind, \(\rho\), one of \{VAL, IVK, UPD\}
  * Evaluation context, \(K\), represents the call stack
  * Target signature, represents the “shape” of the target of the operation
    • either the set of labels in the target object, or
    • the name of a constant
  * Invocation or update message
either a label, or
· a functional constant

– The search semantics is specified by a point cut description language, or PCDL

· PCDL is a parameter to the calculus, various PCDL may be used

Q: How might this be useful?

A: can easily experiment with different PCDL
A: can restrict the set of join points that might be matched

Q: What problems might this cause?

A: might make the semantics more complex
A: possible that complexity is hidden in the PCDL, making the core calculus “less core”

· PCDL consists of two parts:
  · Point cut description syntax, \( \mathcal{C} \)
  · Advice matching function, \( \text{match} \)

• Syntax of \( \varsigma_{\text{asp}} \)

– All object calculus terms
– Constants

\[
d \in \text{Consts} \quad f \in \text{FConsts} \quad \text{terms} \quad a, b, c \quad ::= \quad \ldots \quad | \quad d \quad | \quad a.f
\]

Constants are things like natural numbers
Functional constants are operations like successor
The primary reason for introducing constants is to simplify examples, going forward they may be eliminated–discuss this if time allows

– Advice

\[
p\text{cd} \in \mathcal{C} \quad \text{programs} \quad \mathcal{P} \quad ::= \quad a \otimes \overrightarrow{\mathcal{A}}
\]

\[
\text{advice} \quad \mathcal{A} \quad ::= \quad p\text{cd}\otimes (\overrightarrow{\gamma})b
\]
A program consists of a base term (think “main”) and a sequence of advice. Advice maps a point cut description to a “naked method”. Define naked method

\[
\text{terms } a, b, c ::= \ldots \\
\quad | \text{proceed}_{\text{VAL}}() \\
\quad | \text{proceed}_{\text{IVK}}(a) \\
\quad | \text{proceed}_{\text{UPD}}(a, \varsigma(x)b) \\
\quad | \pi
\]

\[
\text{proceed closures } \pi ::= \Pi_{\text{VAL}}\{B, v\}(a) \\
\quad | \Pi_{\text{IVK}}\{B, S, k\}(a) \\
\quad | \Pi_{\text{UPD}}\{B, k\}(a, \varsigma(x)b)
\]

Advice can contain proceed terms. Proceed terms are converted to proceed closures during advice lookup. User programs cannot contain proceed closures.

- Semantics
  - Changes
    * Object calculus reduction rules are changed to add advice lookup
    * Rules are added for:
      - Constants
      - Object calculus terms to which advice applies
      - Proceeding
  - Helper functions
    * Advice lookup
      \[
      \text{advFor}_M(jp, \bullet) = \bullet \\
      \text{advFor}_M(jp, (pcd\triangleright\varsigma((y\overleftarrow{y}))b) + \overleftarrow{A}) = \\
      \quad \text{match}(pcd\triangleright\varsigma((y\overleftarrow{y}))b, jp) + \text{advFor}_M(jp, \overleftarrow{A})
      \]

Returns a list of naked methods
Invokes PCDL’s \text{match} function for each piece of advice
* Proceed closure

\[\text{close}_{\text{VAL}}(\text{proceed}_{\text{VAL}}(\emptyset B, v)) = \Pi_{\text{VAL}}\{B, v\}(\emptyset)\]

\[\text{close}_{\text{IVK}}(\text{proceed}_{\text{IVK}}(a), \{B, S, k\}) = \Pi_{\text{IVK}}\{B, S, k\}(\text{close}_{\text{IVK}}(a, \{B, S, k\}))\]

\[\text{close}_{\text{UPD}}(\text{proceed}_{\text{UPD}}(a, \varsigma(x)b), \{B, k\}) = \Pi_{\text{UPD}}\{B, k\}(\text{close}_{\text{UPD}}(a, \{B, k\}), \varsigma(x)\text{close}_{\text{UPD}}(b, \{B, k\}))\]

Takes proceed terms in advice and converts them to proceed closures, squirreling away any information needed for proceeding. These are the most interesting definitions, the others just recurse to sub-terms.

– Objects and Basic Constants

values \( v \ ::= d \mid [ \{ l_i = \varsigma(x_i)b_i \mid i \in I \} ] \)

\[
\begin{align*}
\text{RED VAL 0} & \quad \dfrac{\mathcal{K} \vdash_{\mathcal{M}, \overline{A}} \overset{\tau}{\text{advFor}}_M((\text{VAL}, \mathcal{K}, \text{sig}(v), \epsilon), \overline{A}) = \bullet}{\mathcal{K} \vdash_{\mathcal{M}, \overline{A}} v \sim v}
\end{align*}
\]

\[
\begin{align*}
\text{RED VAL 1} & \quad \dfrac{\mathcal{K} \vdash_{\mathcal{M}, \overline{A}} \overset{\tau}{\text{advFor}}_M((\text{VAL}, \mathcal{K}, \text{sig}(v), \epsilon), \overline{A}) = \varsigma()b + B
\quad \text{close}_{\text{VAL}}(b, \{B, v\}) = b' \quad \text{va} \cdot \mathcal{K} \vdash_{\mathcal{M}, \overline{A}} b' \sim v'}
}{\mathcal{K} \vdash_{\mathcal{M}, \overline{A}} v \sim v'}
\end{align*}
\]

Q: What, in plain English, is the meaning of these two rules?

Things to note:

* subscripts on the turnstile
* wellformedness premise
* RED VAL 0 correspondence to RED OBJECT
* advice lookup
  * join point abstraction
Required shape of result in RED VAL 1

* proceed closure, and information stored
* evaluation context in last premise of RED VAL 1

- Method Selection

**RED SEL 0** (where \( o \equiv [l_i = \varsigma(x_i)_{\mathcal{I}}] \))

\[
\begin{array}{c}
\mathcal{K}_{M, \mathcal{A}} a \leadsto o \quad l_j \in \mathcal{I}_{\mathcal{I}} \ni \mathcal{I} \\
adv_{M, \mathcal{A}}(\langle \mathcal{I}V, \mathcal{K}, \mathcal{I}_{\mathcal{I}} \ni \mathcal{I}, l_j \rangle, \mathcal{A}) = \bullet \quad \mathcal{I}b(\mathcal{I}_{\mathcal{I}} \ni \mathcal{I}, l_j) \cdot \mathcal{K}_{M, \mathcal{A}} b \langle x_j \leftarrow o \rangle \leadsto v \\
\mathcal{K}_{M, \mathcal{A}} a.l_j \leadsto v
\end{array}
\]

**RED SEL 1** (where \( o \equiv [l_i = \varsigma(x_i)_{\mathcal{I}}] \))

\[
\begin{array}{c}
\mathcal{K}_{M, \mathcal{A}} a \leadsto o \quad l_j \in \mathcal{I}_{\mathcal{I}} \ni \mathcal{I} \\
adv_{M, \mathcal{A}}(\langle \mathcal{I}V, \mathcal{K}, \mathcal{I}_{\mathcal{I}} \ni \mathcal{I}, l_j \rangle, \mathcal{A}) = \varsigma(y) b + B \\
close_{\mathcal{I}V}(b, \langle y \leftarrow \varsigma(x_j) b \rangle, \mathcal{I}_{\mathcal{I}} \ni \mathcal{I}, l_j \rangle) = b' \quad \mathcal{I}a \cdot \mathcal{K}_{M, \mathcal{A}} b' \langle y \leftarrow o \rangle \leadsto v \\
\mathcal{K}_{M, \mathcal{A}} a.l_j \leadsto v
\end{array}
\]

Q: What, in plain English, is the meaning of these two rules?
Q: Where does the final value come from?

- Functional Constant Application

\( \delta(f, v') \) means “apply the functional constant \( f \) to the value \( v' \). \( \delta \) is intentionally underspecified, since we don’t say what the basic and functional constants are. Suppose \( F\text{Consts} = \{\text{succ}\} \) and \( \text{Consts} \) is the natural numbers: \( \delta(\text{succ}, 3) = 4. \)
RED FConst 0
\[ K_{\mathcal{M}, \mathcal{A}} a \leadsto v' \]
\[ \text{advFor}_M(\langle \text{IVK}, \mathcal{K}, \text{sig}(v'), f \rangle, \overline{A}) = \bullet \quad \text{ib}(\text{sig}(v'), f) : K_{\mathcal{M}, \mathcal{A}} \delta(f, v') \leadsto v \]
\[ K_{\mathcal{M}, \mathcal{A}} a.f \leadsto v \]

RED FConst 1
\[ K_{\mathcal{M}, \mathcal{A}} a \leadsto v' \]
\[ \text{advFor}_M(\langle \text{IVK}, \mathcal{K}, \text{sig}(v'), f \rangle, \overline{A}) = \zeta(y)b + B \]
\[ \text{close}_{\text{IVK}}(b, \{ B, \text{sig}(v'), f \}) = b' \quad \text{ia} : K_{\mathcal{M}, \mathcal{A}} b' \{ y \leftarrow v' \} \leadsto v \]
\[ K_{\mathcal{M}, \mathcal{A}} a.f \leadsto v \]

Q: What is the meaning of these two rules?

Things to note:

* Q: Aren’t these rules non-deterministic given the selection rules? Not if $FConsts \cup Labels = \emptyset$
* Q: How do these rules differ from the selection rules?

No label presence test
Join point abstraction uses sig function
The 0 rule uses δ function

Method Update

RED UPD 0 (where $o \triangleq \{ l_i = \zeta(x_i)b_i \mid i \in I \}$)
\[ K_{\mathcal{M}, \mathcal{A}} a \leadsto o \]
\[ l_j \in \mathcal{T}_i \\text{advFor}_M(\langle \text{UPD}, \mathcal{K}, l_i \mid i \in I, l_j \rangle, \overline{A}) = \bullet \]
\[ K_{\mathcal{M}, \mathcal{A}} a.l_j \leftarrow \zeta(x)b \leadsto [ l_i = \zeta(x_i)b_i \mid i \in I \setminus \{ j \}, l_j = \zeta(x)b ] \]

RED UPD 1 (where $o \triangleq \{ l_i = \zeta(x_i)b_i \mid i \in I \}$)
\[ K_{\mathcal{M}, \mathcal{A}} a \leadsto o \]
\[ \text{advFor}_M(\langle \text{UPD}, \mathcal{K}, l_i \mid i \in I, l_j \rangle, \overline{A}) = \zeta(\text{targ, rval})b' + B \]
\[ \text{close}_{\text{UPD}}(b', \{ B, l_j \}) = b'' \quad \text{ua} : K_{\mathcal{M}, \mathcal{A}} b'' \{ \text{rval} \leftarrow b \{ x \leftarrow \text{targ} \} \} \text{targ} \{ \text{targ} \leftarrow o \} \leadsto v \]
\[ K_{\mathcal{M}, \mathcal{A}} a.l_j \leftarrow \zeta(x)b \leadsto v \]

Things to note:
Correspondence of RED UPD 0 and RED UPDATE

Evaluation context in RED UPD 1

Data used for proceed closure

Shape of returned advice: two parameters

- \( \text{targ} \), corresponds to the target object, \( o \), of the update operation.
- \( \text{rval} \), corresponds to the body, \( b \), of the update’s r-value.

Two kinds of substitution

- \( b\{x ← c\} \) is normal capture-avoiding substitution

  Key rules: the rest just recurse over the grammar

  \[
  (ς(y)b)\{x ← c\} ≡ ς(y')(b\{y ← y'\}\{x ← c\})
  \]
  where \( y' \notin \text{FV}(ς(y)b) \cup \text{FV}(c) \cup \{x\} \)

  \[
  x\{x ← c\} ≡ c
  \]

  \[
  y\{x ← c\} ≡ y \quad \text{if } x \neq y
  \]

- \( b''\{x ← c\}_z \) says: in \( b'' \) replace all free occurrences of \( x \) with \( c \), capturing any free occurrences of \( z \) in \( c \)

  Key rules: varref is same as above, the rest just recurse over the grammar

  \[
  (ς(z)b)\{x ← c\}_z ≡ ς(z)(\{x ← c\}_z) \quad \text{no renaming}
  \]

  \[
  (ς(y)b)\{x ← c\}_z ≡ ς(y')(b\{y ← y'\}\{x ← c\}_z) \quad \text{renaming}
  \]

  \[
  \text{if } y \neq z, \text{ where } y' \notin \text{FV}(ς(y)b) \cup \text{FV}(c) \cup \{x\}
  \]

Q: Which of these rules does the capturing?
A: the first

Why two kinds of substitution? solicit ideas

- \( b\{x ← \text{targ}\} \): renames the self parameter in the body, \( b \), of the original r-value
- \( \text{targ}-\text{capturing substitution for } \text{rval} \text{ in the advice body, } b'' \text{, lets advice author:} \)
  capture occurrences of the self-parameter, by placing \( \text{rval} \) under a \( ς(\text{targ}) \)
  binder
  or
  not capture occurrences of the self-parameter, by not placing \( \text{rval} \) under a binder or by placing it under a non-\( \text{targ} \) binder

Examples:

\[
[n=ς(y)0, \text{pos}=ς(p).\text{p.n}].\text{pos} ⇐ ς(x).\text{n.succ}
\]

- In the absence of advice, this would reduce to:

\[
[n=ς(y)0, \text{pos}=ς(x).\text{n.succ}]
\]

Q: What happens if we update \( n \) to 2 in this object and then select \( \text{pos} \)?
A: We get back 3.
· Advice designed to avoid capture: \texttt{targ does not appear bound in } b''
\begin{equation}
\varsigma(targ,rval)\text{proceed}_{UPD}(targ, \varsigma(z) rval)
\end{equation}

fixes the value of the \texttt{pos} method to the result of evaluating the new method body, \texttt{x.n.succ}, substituting the original target object for \texttt{x}:

Assuming no other advice:
\begin{equation}
b'' = \Pi_{UPD}\{ \bullet, \text{pos}\}(targ, \varsigma(z) rval)
\end{equation}

Underbars indicate target of next substitution

\begin{align*}
\Pi_{UPD}\{ \bullet, \text{pos}\}(targ, \varsigma(z) rval) & \{rval \leftarrow x.n.succ\{x \leftarrow \text{targ}\}\}_\text{targ} \\
& \{\text{targ} \leftarrow [n=\varsigma(y)0, \text{pos}=\varsigma(p).p.n]\}
\end{align*}

\begin{align*}
&= \Pi_{UPD}\{ \bullet, \text{pos}\}(targ, \varsigma(z) rval) \{rval \leftarrow \text{targ.n.succ}\}_\text{targ} \\
& \{\text{targ} \leftarrow [n=\varsigma(y)0, \text{pos}=\varsigma(p).p.n]\}
\end{align*}

\begin{align*}
&= \Pi_{UPD}\{ \bullet, \text{pos}\}(targ, \varsigma(z) target.n.succ) \{\text{targ} \leftarrow [n=\varsigma(y)0, \text{pos}=\varsigma(p).p.n]\}
\end{align*}

\begin{align*}
&= \Pi_{UPD}\{ \bullet, \text{pos}\}([n=\varsigma(y)0, \text{pos}=\varsigma(p).p.n], \varsigma(z)[n=\varsigma(y)0, \text{pos}=\varsigma(p).p.n].n.succ)
\end{align*}

The last term will reduce to:

\begin{equation}
[n=\varsigma(y)0, \text{pos}=\varsigma(z)[n=\varsigma(y)0, \text{pos}=\varsigma(p).p.n].n.succ]
\end{equation}

Q: What happens if we update \texttt{n} to 2 in this object and then select \texttt{pos}?
A: We get back 1!

· Advice designed to capture: because \texttt{rval} appears under a \texttt{targ} binder
\begin{equation}
\varsigma(targ,rval)\text{proceed}_{UPD}(targ,\varsigma(targ) rval.succ)
\end{equation}

uses the body of the update’s r-value without causing it to be reduced

Assuming no other advice was found in the advice lookup, then after closing the \texttt{proceed}_{UPD} sub-term, the substitutions for this advice are:

\begin{align*}
\Pi_{UPD}\{ \bullet, \text{pos}\}(targ,\varsigma(targ) rval.succ) & \{rval \leftarrow x.n.succ\{x \leftarrow \text{targ}\}\}_\text{targ} \\
& \{\text{targ} \leftarrow [n=\varsigma(y)0, \text{pos}=\varsigma(p).p.n]\}
\end{align*}

\begin{align*}
&= \Pi_{UPD}\{ \bullet, \text{pos}\}(targ,\varsigma(targ) rval.succ) \{rval \leftarrow \text{targ.n.succ}\}_\text{targ} \\
& \{\text{targ} \leftarrow [n=\varsigma(y)0, \text{pos}=\varsigma(p).p.n]\}
\end{align*}

\begin{align*}
&= \Pi_{UPD}\{ \bullet, \text{pos}\}(targ,\varsigma(targ)target.n.succ\text{.succ}) \text{ capture!} \\
& \{\text{targ} \leftarrow [n=\varsigma(y)0, \text{pos}=\varsigma(p).p.n]\}
\end{align*}

\begin{align*}
&= \Pi_{UPD}\{ \bullet, \text{pos}\}([n=\varsigma(y)0, \text{pos}=\varsigma(p).p.n], \varsigma(targ) \text{ targ.n.succ.succ})
\end{align*}
The last targ is not free and so isn’t replaced.
(Those last two targ’s should really be renamed, but this is alpha equivalent.)

This term will reduce to:

\[ [n=\varsigma(y)0, \text{pos}=\varsigma(targ)\text{targ.n.succ.succ}] \]

**Q:** What happens if we update \( n \) to 2 in this object and then select \( \text{pos} \)?

**A:** We get back 4!

- **Proceeding**
  
  
  **General ideas:**
  
  · Two rules for each kind of advice **one for proceeding to lower precedence advice, one for proceeding to original operation**
  · Rules are very similar to the regular operations, **except** . . .
  · No additional advice lookup subsequent advice and original operation are taken from the proceed closure
  · Proceed closure formed **lazily**

  **Proceeding from Value Advice**

  \[
  \text{RED VPRCD 0}
  \]

  \[
  \begin{align*}
  \text{K} & \vdash_M m, \ cancelled \\
  \text{K} & \vdash_M \Pi_{\text{VAL}}([\bullet, v]) \rightsquigarrow v
  \end{align*}
  \]

  \[
  \text{RED VPRCD 1}
  \]

  \[
  \begin{align*}
  \text{K} & \vdash_M a \rightsquigarrow o \\
  \text{close}_{\text{VAL}}(b, \{B, v\}) & = b' \\
  \text{va} \cdot \text{K} & \vdash_M b' \rightsquigarrow v' \\
  \text{K} & \vdash_M \Pi_{\text{VAL}}(\varsigma(b + B), v') \rightsquigarrow v'
  \end{align*}
  \]

  **Proceeding from Selection Advice**

  \[
  \text{RED SPRCD 0}
  \]

  \[
  \begin{align*}
  \text{K} & \vdash_M a \rightsquigarrow o \\
  \text{ib}(l, l) \cdot \text{K} & \vdash_M b\{y \leftarrow o\} \rightsquigarrow v \\
  \text{K} & \vdash_M \Pi_{\text{IVK}}(\varsigma(b), l) \rightsquigarrow v
  \end{align*}
  \]

  \[
  \text{RED SPRCD 1}
  \]

  \[
  \begin{align*}
  \text{K} & \vdash_M a \rightsquigarrow o \\
  B & \neq \bullet \\
  \text{close}_{\text{IVK}}(b, \{B, l\}) & = b' \\
  \text{ia} \cdot \text{K} & \vdash_M b'\{y \leftarrow o\} \rightsquigarrow v \\
  \text{K} & \vdash_M \Pi_{\text{IVK}}(\varsigma(b + B), l) \rightsquigarrow v
  \end{align*}
  \]

**Q:** Where does the target object in the 0 rule come from?

**A:** the proceed closure’s argument

**Q:** Where does the method body evaluated in the 0 rule come from?

**A:** the proceed closure’s thunk **not** the target object
* Proceeding from Application Advice

\[
\text{RED FPrCD 0} \quad \begin{align*}
\mathcal{K} \vdash_{M, \mathcal{A}} a & \sim \nu' \\
\mathcal{K} \vdash_{M, \mathcal{A}} \delta(f, \nu') & \sim \nu
\end{align*} \\
\mathcal{K} \vdash_{M, \mathcal{A}} \Pi_{\text{IVK}} \{ \bullet, S, f \} (a) & \sim \nu
\]

\[
\text{RED FPrCD 1} \quad \begin{align*}
\mathcal{K} \vdash_{M, \mathcal{A}} a & \sim \nu' \\
\text{close}_{\text{IVK}} (b, \{ B, S, f \}) & = \nu'
\end{align*} \\
\mathcal{K} \vdash_{M, \mathcal{A}} \Pi_{\text{IVK}} \{(y) b + B), S, f \} (a) & \sim \nu
\]

* Proceeding from Update Advice

\[
\text{RED UPrCD 0} \quad \begin{align*}
\mathcal{K} \vdash_{M, \mathcal{A}} a & \sim [l_i = \varsigma(x_i) b_i \mid i \in I] \\
l_j & \in l_i \mid j \in I
\end{align*} \\
\mathcal{K} \vdash_{M, \mathcal{A}} \Pi_{\text{UPD}} \{ \bullet, l_j \} (a, \varsigma(x) b) & \sim \begin{align*}
[l_i = \varsigma(x_i) b_i \mid i \in I], l_j = \varsigma(x) b
\end{align*}
\]

\[
\text{RED UPrCD 1} \quad \begin{align*}
\mathcal{K} \vdash_{M, \mathcal{A}} a & \sim o \\
\text{close}_{\text{UPD}} (b', \{ B, l_j \}) & = \nu''
\end{align*} \\
\text{ua} \cdot \mathcal{K} \vdash_{M, \mathcal{A}} b'' \{ \text{rval} \leftarrow b \{ x \leftarrow \text{targ} \} \} \{ \text{targ} \leftarrow o \} & \sim \nu
\]

\[
\mathcal{K} \vdash_{M, \mathcal{A}} \Pi_{\text{UPD}} \{(\varsigma(\text{targ}, \text{rval}) b' + B), l_j \} (a, \varsigma(x) b) & \sim \nu
\]
4 Sample Point Cut Description Languages

4.1 Natural Selection, $M_s$

Let $M_s = \langle C_s, match_s \rangle$, where $C_s ::= [\llbracket . \rrbracket].l$ and:

$$
match_s([\llbracket . \rrbracket].l \triangleright \varsigma(y)b, \langle \rho, K, S, k \rangle) =
\begin{cases}
\langle \varsigma(y)b \rangle & \text{if } (\rho = \text{IVK}) \land (S = I) \land (k = l) \\
\text{• otherwise}
\end{cases}
$$

Example:

- Without advice:
  \[ \text{[pos} = \varsigma(p).n, n = \varsigma(y)2].\text{pos} \sim 2 \]
- With before advice \([\text{pos, n}].\text{pos} \triangleright \varsigma(x)\text{proceed}_{\text{IVK}}((x.n \Leftarrow \varsigma(y)0)):\]
  \[ \text{[pos} = \varsigma(p).n, n = \varsigma(y)2].\text{pos} \sim 0 \]
- With after advice \([\text{pos, n}].\text{pos} \triangleright \varsigma(x)\text{proceed}_{\text{IVK}}(x).\text{succ}:\]
  \[ \text{[pos} = \varsigma(p).n, n = \varsigma(y)2].\text{pos} \sim 3 \]

4.2 General Matching, $M_G$

- Allows queries over all portions of the join point abstraction.
  
  - Reduction Kind: \text{IVK}
    $$
    C_G ::= \text{VAL} \mid \text{IVK} \mid \text{UPD} \mid \ldots
    $$
  
  - Message: \text{IVK} \land k = \text{pos}
    $$
    C_G ::= \ldots \mid k = k \mid \ldots
    $$
  
  - Target signature: \text{IVK} \land k = \text{pos} \land S = \{\text{pos,n}\}
    $$
    C_G ::= \ldots \mid S = k \mid \ldots
    $$
  
  - Evaluation Context: \(K \in .*/.\text{ib}(\{\text{pos,n}\},\text{pos})\)
    $$
    C_G ::= \ldots \mid K \in r \mid \ldots
    $$

  context expr. \( r ::= \epsilon \mid \text{ib}(M, m) \mid \text{va} \mid \text{ia} \mid \text{ua} \mid \ldots \mid r + r \mid rr \mid r^* \)
  
  signatures \( M ::= d \mid I \mid . \)
  
  messages \( m ::= f \mid l \mid . \)
Boolean Combinations: \( \text{IVK} \land k = \text{pos} \land S = \{\text{pos},n\} \land \neg(K \in .^* \text{ib}({\{\text{pos},n}\},\text{pos})) \)

\[ C_G ::= \ldots | \neg pcd | pcd \land pcd | pcd \lor pcd | \]

Q: To what AspectJ point cut description does this example correspond?
A: \(\text{call}(\text{Point.pos}) \land \neg \text{cflowbelow}(\text{call}(\text{Point.pos}))\)

\(- M_G \) is sufficient to model AspectJ

- Join points

<table>
<thead>
<tr>
<th>AspectJ Point Cut</th>
<th>Modeled In ( \varsigma_{asp}(M_G) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{call}(\text{void Point.pos})())</td>
<td>(\text{IVK} \land S = {n,\text{pos}} \land k = \text{pos} )</td>
</tr>
<tr>
<td>(\text{call}(\text{Point.new})())</td>
<td>(\text{VAL} \land S = {n,\text{pos}} )</td>
</tr>
<tr>
<td>(\text{execution}(\text{void Point.pos})())</td>
<td>(\text{VAL} \land K \in \text{ib}({{n,\text{pos}}},\text{pos}).^* )</td>
</tr>
<tr>
<td>(\text{get}(\text{int Point.n}))</td>
<td>(\text{IVK} \land S = {n,\text{pos}} \land k = \text{n} )</td>
</tr>
<tr>
<td>(\text{set}(\text{int Point.n}))</td>
<td>(\text{UPD} \land S = {n,\text{pos}} \land k = \text{n} )</td>
</tr>
<tr>
<td>(\text{adviceexecution}())</td>
<td>(K \in .^* (\text{va} + \text{ia} + \text{ua}).^* )</td>
</tr>
<tr>
<td>(\text{within}(\text{Point}))</td>
<td>(K \in \text{ib}({{n,\text{pos}}},.).^* )</td>
</tr>
<tr>
<td>(\text{withincode}(\text{Point.pos}))</td>
<td>(K \in \text{ib}({{n,\text{pos}}},\text{pos}).^* )</td>
</tr>
<tr>
<td>(\text{cflow}(\text{Point.pos}))</td>
<td>(K \in .^* \text{ib}({{n,\text{pos}}},\text{pos}).^* )</td>
</tr>
<tr>
<td>(\text{cflowbelow}(\text{Point.pos}))</td>
<td>(K \in .^* . \text{ib}({{n,\text{pos}}},\text{pos}).^* )</td>
</tr>
<tr>
<td>(\text{this}(\text{Point}))</td>
<td>(K \in \text{ib}({{n,\text{pos}}},.).^* )</td>
</tr>
<tr>
<td>(\text{target}(\text{Point}))</td>
<td>(S = {n,\text{pos}} )</td>
</tr>
</tbody>
</table>

Q: Does cflowbelow consider advice execution to be “below” a cflow?
Q: Does our model?
A: Yes
Q: What if it should not be?
A: \(K \in .^* \text{ib}(.), \text{ib}({\{n,\text{pos}\}},\text{pos}).^* \)
Q: What about the variable binding form of this?
A: Would need to change core calculus to track this in evaluation context.
Q: What else is missing?
A: Non-sensicle: Constructor advice, initialization advice, handler advice, args point cut
A: Omitted: if (but could be handled in PCDL without changing core calculus)

**Homework:** Are there interesting things that can be said in $M_G$ that would give insight into join points "missing" from AspectJ?
- Open Classes (a.k.a. intertype declarations)
  
  ```java
  int Point.color = 0;
  ```

  A model of this in $M_G$ uses two pieces of advice:

  \[
  (\text{VAL} \land S = \{ n, \text{pos} \}) \triangleright \varsigma() \begin{cases} 
  \text{orig}=\varsigma(s)\text{proceed}_{\text{VAL}}(), \\
  n=\varsigma(s)s.\text{orig}.n, \\
  \text{pos}=\varsigma(s)s.\text{orig}.\text{pos}, \color{gray}=\varsigma(s)0
  \end{cases}
  \]

  \[
  (\text{UPD} \land S = \{ \text{orig}, \text{pos}, \color{gray} \} \land (k = n \lor k = \text{pos})) \triangleright \\
  \varsigma(t,r) \begin{cases} 
  \text{orig}=\varsigma(s)\text{proceed}_{\text{UPD}}(t.\text{orig}, \varsigma(t)r), \\
  n=\varsigma(s)s.\text{orig}.n, \\
  \text{pos}=\varsigma(s)s.\text{orig}.\text{pos}, \color{gray}=\varsigma(s)t.\color{gray}
  \end{cases}
  \]

  Q: Why is the second piece of advice needed?
  A: Consider an invocation after an update if we just have the first piece of advice.

### 4.3 Other Models

- Modeling HyperJ
  - Can use $M_G$
  - Like Open Classes, but two key differences:
    - Special basic constants represent module names
    - A model for abstract methods allows composed modules to call each other while remaining oblivious to the other modules implementation

- Modeling Adaptive Methods
  - Basic Idea
    Adaptive methods allow a **declarative** specification of a **traversal strategy** over an **object graph**.
    Specify:
    - **links to be traversed in object graph**
• actions to performed at each node in the graph

Example: for each Employee e of each CostCenter c: total += e.salary()

– Is \( M_G \) sufficient?
  Need mechanism to find fields of an object that are not known when the advice is written. Reflection

– Keys to model in \( \varsigma_{asp} \)
  * Use distinguished names to indicate fields of objects
  * Extend \( M_G \) with reflection
    \[ \forall f \in \text{fieldsOf}(S) \]
    Matching advice returns \( n \) copies of the advice body, for a target object with \( n \) fields.
  * Use the two parameters of update advice in a unique way
    · Target object is used for dispatching to the appropriate code for the node
    · R-value is used to pass a visitor (accumulator) object

4.4 Insights

• Spectators and Assistants
  Q: Can we study them using \( \varsigma_{asp} \)?
  Q: How might we add imperative features?
  Q: Can we eliminate any features from \( \varsigma_{asp} \)? Should we?

• Interaction of PCDL and base language
  Q: How does the design of the PCDL effect reasoning in the base language?

• Comparisons
  Q: What do we learn about similarities between the modeled langauges?
  Q: Differences?

4.5 Decisions in the design of \( \varsigma_{asp} \)

• Big step or little step?

• Functional or imperative?

• Include constants?

• Advice declarations or terms? Advice scoping
References


