Aspects and Modular Reasoning in Nonmonotonic Logic

Klaus Ostermann
Darmstadt University of Technology
Many people have noted that programs should “look like” our thought process about the problem.

- direct mapping principle (Meyer)
- low representational gap (Larman)
- logical vs. physical hierarchies (Wegner)
- ...

However, research from the AI community on how humans think has so far had little impact on PL research.
Overview

- Fundamental insight in AI research: Humans reason in a non-monotonic way. Humans reason frequently with incomplete or changing information.
  - New knowledge may invalidate previous conclusions
- Example: Birds usually fly and Tweety is a bird $\Rightarrow$ Tweety flies.
- Later we learn that Tweety is a penguin...
- In classical logic, if $\Gamma \vdash X$ and $\Gamma \subseteq \Gamma'$, then $\Gamma' \vdash X$.
  - Not possible to express “rules of thumb” or defaults as above in classical logic.
- Nonmonotonic logic has been developed to deal with nonmonotonicity in a rigorous and controlled way.
Hypothesis of this work

- Aspects can be interpreted as a form of nonmonotonicity
  - We can give a “default meaning” to a computational entity
  - Later (when we learn about a different concern) we can refine the meaning of this entity.

- To validate the hypothesis we perform three experiments:
  - Modeling the semantics of an AO language using nonmonotonic logic.
  - Modeling advice precedence rules with prioritized default logic.
  - Revisit the question of modular reasoning and modular verification on the basis of a semantics in default logic.
Default Logic

► Default logic is the best-known variant of nonmonotonic logics.

► Our rule about birds can be expressed as follows:

\[
\begin{align*}
\text{bird}(X) & : \text{flies}(X) \\
\implies \text{flies}(X)
\end{align*}
\]

► A default \( \varphi;\psi_1,\ldots,\psi_n \) is applicable to a deductively closed set of formulae \( E \), if \( \varphi \in E \) and \( \neg\psi_1 \notin E, \ldots, \neg\psi_n \notin E \).

► Set of conclusions from a knowledge base is in general not unique.

► Possible consistent world views from a knowledge base \( T = (W, D) \) are called extensions.

► Normal defaults...
Algorithm to compute extensions

\[ E := Th(W); A := \emptyset; \]
while there is a default \( \delta \notin A \) that is applicable to \( E \) {
\[ E := Th(E \cup \{\text{consequent}(\delta)\}); A := A \cup \{\delta\}; \]
}
if \( \forall \delta \in A. E \) is consistent with all justifications of \( \delta \)
then return \( E \) else failure
AO semantics in the style of Jagadeesan et al

\[ \tilde{a} = \text{ApplicableAdvice}(o, m) \]
\[ \ldots o.m(\vec{v}) \leftarrow \ldots o.m[\tilde{a}](\vec{v}) \]

(\textit{WEAVE})

\[ \text{AdviceLookup}(a) = (\vec{x}, e) \]
\[ \ldots o.m[a, \tilde{a}](\vec{v}) \leftarrow \ldots e \left[ o/\textit{this}, \vec{v}/\vec{x}, o.m[\tilde{a}](\vec{v}) \right] \]

(\textit{ADVEXEC})

\[ \text{MethodLookup}(o, m) = (\vec{x}, e) \]
\[ \ldots o.m[\emptyset](\vec{v}) \leftarrow \ldots e \left[ o/\textit{this}, \vec{v}/\vec{x} \right] \]

(\textit{METHEXEC})
AO semantics in the style of Jagadeesan et al

- Semantics requires global operation that requires knowledge of the full program to compute the list of all advice that applies: *ApplicableAdvice*

- There is no direct specification of the semantics of an aspect, but just a specification of what its effect on the program is.

- Hence, the set of rule instances does not grow monotonically with the program.

- Next up: AO semantics using defaults

- To get rid of the global advice list, we re-interpret the advice list in a method call to mean the set of *already executed* advice.
AO semantics using defaults

\[\text{MethodLookup}(o, m) = (x, e)\]

\[\text{unadvised}(o, m, a)\]

\[...o.m[a](\vec{v}) \leftrightarrow ...e \overset{o}{\text{/this}}, \overset{\vec{v}}{\text{/x}}\]

\[\text{NextAdvice}(o, m, a) = a\]

\[\text{AdviceLookup}(a) = (x, e)\]

\[...o.m[a](\vec{v}) \leftrightarrow ...e \overset{o}{\text{/this}}, \overset{\vec{v}}{\text{/x}}, o.m[a,a](\vec{v}) / \text{proceed}\]

\[\text{true} : \text{unadvised}(o, m, a)\]

\[\text{unadvised}(o, m, a)\]

\[a \in \text{ApplicableAdvice}(o, m) \land a \notin \vec{a} : \text{NextAdvice}(o, m, a) = a\]

\[\text{NextAdvice}(o, m, a) = a\]

\[\text{true} : \text{unadvised}(o, m, a)\]

\[a \in \text{ApplicableAdvice}(o, m) \land a \notin \vec{a} : \text{NextAdvice}(o, m, a) = a\]

\[\text{true} : \text{unadvised}(o, m, a)\]

\[\text{true} : \text{unadvised}(o, m, a)\]

\[a \in \text{ApplicableAdvice}(o, m) \land a \notin \vec{a} : \text{NextAdvice}(o, m, a) = a\]

\[\text{true} : \text{unadvised}(o, m, a)\]

\[a \in \text{ApplicableAdvice}(o, m) \land a \notin \vec{a} : \text{NextAdvice}(o, m, a) = a\]

\[\text{true} : \text{unadvised}(o, m, a)\]

\[a \in \text{ApplicableAdvice}(o, m) \land a \notin \vec{a} : \text{NextAdvice}(o, m, a) = a\]

\[\text{true} : \text{unadvised}(o, m, a)\]

\[a \in \text{ApplicableAdvice}(o, m) \land a \notin \vec{a} : \text{NextAdvice}(o, m, a) = a\]

\[\text{true} : \text{unadvised}(o, m, a)\]

\[a \in \text{ApplicableAdvice}(o, m) \land a \notin \vec{a} : \text{NextAdvice}(o, m, a) = a\]

\[\text{true} : \text{unadvised}(o, m, a)\]

\[a \in \text{ApplicableAdvice}(o, m) \land a \notin \vec{a} : \text{NextAdvice}(o, m, a) = a\]

\[\text{true} : \text{unadvised}(o, m, a)\]

\[a \in \text{ApplicableAdvice}(o, m) \land a \notin \vec{a} : \text{NextAdvice}(o, m, a) = a\]

\[\text{true} : \text{unadvised}(o, m, a)\]

\[a \in \text{ApplicableAdvice}(o, m) \land a \notin \vec{a} : \text{NextAdvice}(o, m, a) = a\]

\[\text{true} : \text{unadvised}(o, m, a)\]

\[a \in \text{ApplicableAdvice}(o, m) \land a \notin \vec{a} : \text{NextAdvice}(o, m, a) = a\]

\[\text{true} : \text{unadvised}(o, m, a)\]

\[a \in \text{ApplicableAdvice}(o, m) \land a \notin \vec{a} : \text{NextAdvice}(o, m, a) = a\]

\[\text{true} : \text{unadvised}(o, m, a)\]

\[a \in \text{ApplicableAdvice}(o, m) \land a \notin \vec{a} : \text{NextAdvice}(o, m, a) = a\]

\[\text{true} : \text{unadvised}(o, m, a)\]

\[a \in \text{ApplicableAdvice}(o, m) \land a \notin \vec{a} : \text{NextAdvice}(o, m, a) = a\]

\[\text{true} : \text{unadvised}(o, m, a)\]

\[a \in \text{ApplicableAdvice}(o, m) \land a \notin \vec{a} : \text{NextAdvice}(o, m, a) = a\]

\[\text{true} : \text{unadvised}(o, m, a)\]

\[a \in \text{ApplicableAdvice}(o, m) \land a \notin \vec{a} : \text{NextAdvice}(o, m, a) = a\]

\[\text{true} : \text{unadvised}(o, m, a)\]

\[a \in \text{ApplicableAdvice}(o, m) \land a \notin \vec{a} : \text{NextAdvice}(o, m, a) = a\]

\[\text{true} : \text{unadvised}(o, m, a)\]

\[a \in \text{ApplicableAdvice}(o, m) \land a \notin \vec{a} : \text{NextAdvice}(o, m, a) = a\]

\[\text{true} : \text{unadvised}(o, m, a)\]

\[a \in \text{ApplicableAdvice}(o, m) \land a \notin \vec{a} : \text{NextAdvice}(o, m, a) = a\]

\[\text{true} : \text{unadvised}(o, m, a)\]

\[a \in \text{ApplicableAdvice}(o, m) \land a \notin \vec{a} : \text{NextAdvice}(o, m, a) = a\]

\[\text{true} : \text{unadvised}(o, m, a)\]

\[a \in \text{ApplicableAdvice}(o, m) \land a \notin \vec{a} : \text{NextAdvice}(o, m, a) = a\]

\[\text{true} : \text{unadvised}(o, m, a)\]

\[a \in \text{ApplicableAdvice}(o, m) \land a \notin \vec{a} : \text{NextAdvice}(o, m, a) = a\]

\[\text{true} : \text{unadvised}(o, m, a)\]

\[a \in \text{ApplicableAdvice}(o, m) \land a \notin \vec{a} : \text{NextAdvice}(o, m, a) = a\]

\[\text{true} : \text{unadvised}(o, m, a)\]

\[a \in \text{ApplicableAdvice}(o, m) \land a \notin \vec{a} : \text{NextAdvice}(o, m, a) = a\]
AO semantics using defaults

- A global list of all advice that apply at some point is never required.
- Rule instances are preserved by program expansion.
- An aspect is given a (logical) meaning independent of the program to which it applies.
- If at most one pointcut applies at any joinpoint, the two semantics agree because:
  - There is only one unique extension in the default theory, which is the same theory that is generated by the conventional operational semantics
- The semantics differ in how they treat shared joinpoints.
  - Order returned by $\text{ApplicableAdvice}$ vs. one extension for every possible execution order
- Next up: prioritized default logic to model AspectJ-like global orders and ordering hints (such as $\text{declare precedence}$ in AspectJ) on advice.
Prioritized Default Logic (PRDL)

- In PRDL, every default $\delta_i$ has a name $d_i$.
- ... and has a special symbol $\prec$ operating on default names.
- $d_i \prec d_j$ means $d_i$ has priority over $d_j$.
- Formulae containing $\prec$ can be used both in the background theory and in default rules.
Algorithm to compute priority extensions

\[ E := \text{Th}(W); \ A := \emptyset; \ Prio := \emptyset \]

while there is a default \( \delta \notin A \) that is applicable to \( E \) {
\[
\begin{align*}
C &:= \{ \text{nameof}(\delta') \mid \delta' \in D, \delta' \neq \delta, \delta' \text{ is applicable to } E \} \\
Prio &:= Prio \cup \{ \text{nameof}(\delta) \prec d \mid d \in C \} \\
E &:= \text{Th}(E \cup \{ \text{consequent}(\delta) \}); \ A := A \cup \{ \delta \};
\end{align*}
\]
}

if \( E \) is consistent with \( Prio \)
then return \( E \) else failure
Modeling AspectJ-like priorities in PRDL

\[
\text{true} : \text{defaultOrder}\left(\{a_1, a_2\}\right) \\
\text{defaultOrder}\left(\{a_1, a_2\}\right)
\]

\[
\text{defaultOrder}\left(\{a_1, a_2\}\right) \land (a_1 <_{\text{default}} a_2) \\
\text{NEXTAdv}_{o,m,\bar{a},a_1} < \text{NEXTAdv}_{o,m,\bar{a},a_2}
\]

(DEFAULT)

(DECLDEFLT)
Modeling AspectJ-like priorities in PRDL

\[
\begin{align*}
\text{true} : \ & \text{defaultOrder}(\{a_1, a_2\}) \\
\Rightarrow \ & \text{defaultOrder}(\{a_1, a_2\}) \\
\text{(DEFAULT)}
\end{align*}
\]

\[
\begin{align*}
\text{defaultOrder}(\{a_1, a_2\}) \land (a_1 <_{\text{default}} a_2) \\
\Rightarrow \ & \text{NEXTAdv}_{o,m,\bar{a},a_1} \prec \text{NEXTAdv}_{o,m,\bar{a},a_2} \\
\text{(DECLDEFLT)}
\end{align*}
\]

\[
\begin{align*}
\text{declare precedence } a_1, a_2 \in P \\
\Rightarrow \ & \neg \text{defaultOrder}(\{a_1, a_2\}) \\
\text{(DECLPREC1)}
\end{align*}
\]

\[
\begin{align*}
\text{declare precedence } a_1, a_2 \in P : (\text{NEXTAdv}_{o,m,\bar{a},a_1} \prec \text{NEXTAdv}_{o,m,\bar{a},a_2}) \\
\Rightarrow \ & \text{NEXTAdv}_{o,m,\bar{a},a_1} \prec \text{NEXTAdv}_{o,m,\bar{a},a_2} \\
\text{(DECLPREC2)}
\end{align*}
\]
Modeling AspectJ-like priorities in PRDL

- Again, the precedence declarations are given a compositional semantics, independent of the rest of the program.

- Semantics agrees with “classical” semantics in that there is only one unique extension that is equal to the theory of the classical semantics.

- ...except if there are contradicting precedence declarations

  - Purpose of the justification in (DeclPrec2)...  

- Higher-order (and dynamic) priority declarations can easily be modelled in PRDL.
We believe that the absence of any global operations in the formal semantics can make a difference w.r.t. modular reasoning.

But... what exactly is modular reasoning?

From the perspective of logic, reasoning means the application of a proof calculus of a logic on a knowledge base.

To reason about a program, we hence need a way to generate a knowledge base from a program and a proof calculus.
Program $P'$ is an expansion of $P$ if $P$ is a part of $P'$.

Definition: A language admits modular reasoning with respect to a $\text{prog2kb}$ function, if, for all programs $P$ and $P'$ such that $P'$ is an expansion of $P$, we have $\text{prog2kb}(P) \subseteq \text{prog2kb}(P')$.

The set of rule instances of an operational semantics for some program is such a knowledge base.

Observation: The default logic version of the semantics admits modular reasoning, the conventional semantics does not.
One may argue that modular reasoning is not worth much in a nonmonotonic logic.

- Rather than preservation of the knowledge base one would rather have preservation of the set of conclusions.

We believe there is still value in our approach because we can now deal with the nonmonotonicity in a reasoning framework that has been specifically developed for this purpose.

To illustrate this claim we discuss how properties of a program can be verified in a modular way.
Example

bool f(int n) {
    if n<=0 then return g(n)
    else return isPrime(n);
}
bool g(int n) { return isPrime(-n); }

bool isPrime(int n) {
    if n<=1 then return false;
    for (int i=2; i<n; i++) {
        if n modulo i = 0 then return false;
    }
    return true;
}
Proof of a property in default logic

\[ \forall n. \text{body}_{\text{isPrime}} \rightarrow^\ast \text{true} \iff \text{prime}(n) \]

Assumption: \text{unadvised}(\text{isPrime}(n))

\[ \forall n. \text{isPrime}(n) \rightarrow^\ast \text{true} \iff \text{prime}(n) \]

\[ \forall n. \text{isPrime}(-n) \rightarrow^\ast \text{true} \iff \text{prime}(-n) \]

Assumption: \text{unadvised}(\text{g}(n))

\[ \forall n. \text{g}(n) \rightarrow^\ast \text{true} \iff \text{prime}(-n) \]

\[ \forall n. \text{body}_{\text{f}} \rightarrow^\ast \text{true} \iff (n > 0 \text{ and } \text{prime}(n)) \text{ or } (n \leq 0 \text{ and } \text{prime}(-n)) \]

Assumption: \text{unadvised}(\text{f}(n))

\[ \forall n. \text{f}(n) \rightarrow^\ast \text{true} \iff (n > 0 \text{ and } \text{prime}(n)) \text{ or } (n \leq 0 \text{ and } \text{prime}(-n)) \]
Now consider an expansion of the program with additional advice. Is the proof $s$ (and hence property) still valid?

Quick check: Compare whether the justification set $J(s)$ is consistent with our expansion.

If an assumptions in $J(s)$ has been violated by the extension, however, the property may no longer hold.

We can still try to “repair” the proof without revisiting the program.

Example: Expansion with the following advice:

```cpp
advice(int n) returns bool:
    around call(isPrime(n)) {
        if n % 2 = 0 then return false;
        return proceed;
    }"
Repairing the proof

∀n. body_{isPrime} →* true ↔ prime(n)

Assumption: unadvised(isPrime[primopt](n))

∀n. n mod 2 = 0 ⇒ not prime(n)  ∀n. isPrime[primopt](n) →* true ↔ prime(n)

∀n. body_{advice} →* true ↔ prime(n)

Assumption: NextAdvice(isPrime(n)) = primopt

∀n. isPrime(n) →* true ↔ prime(n)

∀n. isPrime(-n) →* true ↔ prime(-n)

Assumption: unadvised(g(n))

∀n. g(n) →* true ↔ prime(-n)

∀n. body_{f} →* true ↔ (n > 0 AND prime(n)) OR
(n ≤ 0 AND prime(-n))

Assumption: unadvised(f(n))

∀n. f(n) →* true ↔ (n > 0 AND prime(n)) OR
(n ≤ 0 AND prime(-n))
Conclusions

- Nonmonotonic logic is a good (mental and formal) model to explain AOP.
- I hope that many results from nonmonotonic logic can be used to improve AOP
  - Semantics for AO languages
  - Advanced priority mechanisms
  - Proof theory / modular verification
- Future Work: More direct incorporation of defaults into AO languages