Selected Topics

Discussed Topics 1

Model-View-Controller vs. Model-View-Presenter
(similar to Model-Vew-ViewModel)

Blocking vs. nonblocking IO (NIO, NIO2)

Parallel class hierarchies (as a symptom)

Language bindings

Design patterns vs. language features

Functional Object-oriented Programming

possible point of view: object-orientation = organisational paradigm,
model of computation orthogonal (imperative, functional, etc.)

contradiction: object has state, behaviour, identity;
referential transparency = no state, no identity

in practice: no referential transparency for input/output

→ state and identity also in functional programming,
but most inner program parts are referentially transparent

state of the art: functional part separated from the object-oriented part

pattern matching also usable in object-oriented programming,
genericity related to referential transparency

Type Inference

rather simple type inference algorithm:

use fresh **type variable** where no type is known,
unify types that should be equal (thereby binding type variables)

algorithm usually efficient, but can become exponential

restriction: inappropriate for polymorphic invocations (especially subtyping)
because unification causes types to become equal

problem undecidable when using semi-unification instead of unification

→ state-of-the-art where unification is appropriate
(functional programming, lambda expressions, genericity)

→ not supported where semi-unification would be needed (subtyping)

Checker Framework (Java)

program code contains annotations on types used as types

a tool statically verifies these type annotations (on source code or byte code)

many checks predefined, programmers can define their own type systems

typical example: **Nullness Checker** gives warnings on improper uses of `null`

```
@Nullable Object o;  // can be null
@NonNull  Object n;  // never null (Default)
...
o.toString() // warning: can cause null pointer exception
n = o;       // warning: null can be assigned to n
if (n==null) // warning: redundant test
```

Demand of Type Information

many annotations necessary to keep track of information, e.g. for nullness:

- `@RequiresNonNull` on method: precondition on given (field) variables,
- `@EnsuresNonNull` on method: postcondition on given (field) variables,
- `@EnsuresNonNullIf`: same, but holds only under a given condition,
- `@Initialized` on type: ensures that a variable is fully initialized,
- ...

still not enough information for complete checking → "hacks" necessary:

```
@NonNull X[] nxa = new @NonNull X[10];  // error
@MonotonicNonNull X[] tmp = new @MonotonicNonNull X[10];
for (int i=0; i<tmp.length; i++) tmp[i] = new X();
@SuppressWarnings("nullness")   // checker does not know
@NonNull X[] xa = tmp;
```