Pluggable Types

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: n may become null
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;
```

Linear Checker for Preventing Aliasing

variable with linear type = currently no other reference to referenced object

```
@Linear Object l1 = new Object();
@Linear Object l2 = l1; // l1 no longer usable in any way
Object nl1 = l2;        // l2 no longer usable
Object nl2 = nl1;       // nl1 and nl2 refer to object

@Linear Object foo(@Linear p) {
    if (p.hashCode() <= 0)    // keeps linearity
        return new Object();  // linearity != identity
    return p;
}
```

very restrictive: only on parameters, local variables, return types
(everything else is future work)

Preventing Aliases vs. Controlled Use

preventing aliases is very restrictive $\longrightarrow$ rarely usable in practice

simple strategies to deal with aliasing:

referential transparency (functional programming),
factorisation to keep interconnections local (objects)

linearity possible without preventing aliases completely, e.g. by

locking to ensure transient exclusive access (not just synchronization)
exclusive access within specified time frames (TTP)
exclusive access under given conditions (like specific variable values)
exclusive use of specific methods (like one producer, one consumer)

Non-standard Models of Concurrency

Concurrent Constraint Logic Programming

example in Parlog (resembles Prolog, concurrency instead of backtracking):

```
prod1(x), prod2(y), merge(x,y,z).      % three concurrent goals
mode merge(list1?,list2?,merged^).     % direction of data flow
merge([u|x],y,[u|z]) <- merge(x,y,z). % select either rule ...
merge(x,[u|y],[u|z]) <- merge(x,y,z). % ... indeterministically
merge([],y,y).                         % finally close 'streams'
merge(x,[],x).
```

embeddings for functional programming as well as active objects

other languages in this AND-parallel tradition: Strand 88, Oz (Mozart)

OR-parallel Prolog = computing alternatives in parallel (e.g. Aurora)

Tuple Spaces

a tuple space is a sort of associative memory (alos known as **blackboard**)

origin: Linda tuple spaces (Gelernter and Carriero) with these operations:

- `in`: atomically read and consume a tuple from tuple space
- `rd`: read a tuple from tuple space (without consuming)
- `out`: write a new tuple into tuple space
- `eval`: create a process to compute a tuple, result written into tuple space

today there are many variants of (distributed) tuple spaces, e.g., MozartSpaces

Actors in Erlang

Erlang is a funktional language with an embedded actor model

```
ringing_a(A, B) ->
    receive
        {A, on_hook} ->
            A ! {stop_tone, ring},
            B ! terminate,
            idle(A);
        {B, answered} ->
            A ! {stop_tone, ring},
            switch ! {connect, A, B},
            conversation_a(A, B)
    end.
```

Final Remarks

History of Object-oriented Programming

Languages: Simula, Smalltalk, Objective-C, C++, Eiffel, Self, CLOS, Oberon, Java, C#, Python, Ruby, . . .

Concepts: structured programming, abstraction, inheritance, substitutability, interface specifications, parametrisation (genericity, annotations, aspects, . . .)

Methods: factorization, use cases, graphical representation (UML), design patterns, pair programming, . . .

Conflicts: functional programming, relational databases, collections and covariant problems, formal complexity, concurrency

Trends: object-based, object-oriented, (partially automated), typed, team+architecture-integrated, layers and frameworks, back to the roots

Future of Object-oriented Programming

OOP omnipresent $\longrightarrow$ no longer innovativ

splitting into many details and side issues

topics of the near future: concurrency, distributed programming, data integration and big data, cloud computing, complex behavioural interfaces, deeply layered architectures, security, ...

currently more open questions than answers

language support expected when most important questions answered
$\longrightarrow$ language support mainly for topics that are no longer up-to-date?