Type Systems

some terminology

based on the first pages of
"Type Systems", Luca Cardelli
CRC Handbook of Computer Science and Engineering

Execution errors

- Obvious symptoms
  - illegal instruction fault, illegal memory reference fault
- More subtle
  - data corruption (no immediate symptom)
- Not so easy
  - some software faults are not prevented by type systems
  - some languages without type systems where faults never occur

Typed and untyped languages

- Program variable can assume a range of values during execution
  - a **type** is an upper bound of such a range
- **typed language**: where variables can be given (non-trivial) types
- **untyped language** (*) range of variables is unrestricted
  - no type, or eq., a single universal type that contains all values)
  - application to inappropriate arguments can lead to *an arbitrary value, a fault, an exception, or an unspecified effect*
  - extreme case: λ-calculus, untyped, yet no faults

(*) also called "dynamically-typed", "latently-typed", "dynamically-checked"

Type system

- Component of a typed language that keeps track of the types of expressions in a program
- Used to determine whether programs are **well-behaved**
- Discard bad programs before they are run
- **Typed language**: ∃ type system for it
  - can be **explicitly typed** (types are part of the syntax)
  - can be **implicitly typed**
  - or mix (ML, Haskell)

Execution errors

- Two kinds of errors
- **Trapped errors**: cause computation to stop immediately
  - eg. division by zero
  - eg. accessing an illegal address
- **Untrapped errors**: can go unnoticed (for a while)
  - eg. improperly accessing a *legal* address
  - eg. jump to the wrong address

Safety

- A program fragment is **safe** if it does not cause untrapped errors to occur
- **Safe language**: language where all programs are safe
  - untyped languages may enforce **safety** by performing runtime checks
  - typed languages may enforce safety by statically rejecting programs that are *potentially* unsafe
  - typed languages may use a mixture of runtime and static checks
- Typed languages usually aim to rule out also large classes of trapped errors

Well-behaved programs

- We may designate a subset of the possible execution errors as **forbidden errors**
  - forbidden = untrapped + subset of trapped
- **Good behavior** = no forbidden error occurs
- **Strongly checked language**: all programs are well-behaved
  - no untrapped errors (safety)
  - no forbidden errors
  - other trapped errors can occur (programmer's responsability)

Well-behaved programs

- Typed languages can enforce good behavior by performing static checks
  - called **statically-checked** languages
  - checking process is called **typechecking**
  - algorithm is called **typechecker**
  - program that passes the typechecker is **well typed** (otherwise it is **ill typed**)
  - ill-typed does not necessarily mean ill-behaved

Well-behaved programs

- Untyped languages can enforce good behavior (including safety) by performing sufficiently detailed **runtime checks**
  - eg. check array bounds, division operations, generate recoverable exceptions when forbidden errors would happen
  - process: **dynamic checking**
  - such languages are strongly checked! (even though they have no static checking, no type system)
- Even statically-checked languages usually perform some tests at runtime (for safety)
  - static checking does not mean execution can proceed blindly

Dynamic type checks

- Java and others have constructs to discriminate based on the runtime type of an object
  - instanceof, cast
- Not fully statically checked, even though the dynamic tests are defined on the basis of the static type system
  - dynamic tests for type equality are compatible with statics

Lack of safety

- Well-behaved => safe
- Some statically-checked languages are not safe
  - ie. forbidden errors do not include all untrapped errors
  - sometimes called **weakly-checked** (weakly-typed)
  - C has many unsafe, widely used features (pointer arithmetic, casting)
- Most untyped languages are, by necessity, completely safe

Safety

	Typed	Untyped
Safe	ML, Java	Lisp, Smalltalk
Unsafe	C	Assembler

Why unsafe?

- Advantage:
  - execution time
- Problems:
  - development and maintenance time
  - security holes

Soundness

- Type systems define a notion of well typing, a static approximation of good behavior (including safety)
- How can we guarantee that:
  - *well typed programs are well behaved*
- Type soundness theorem
  - "type system is sound"
  - sometimes also called "type safety" (!= language safety)