∼ Topic VI ∼

Types in programming languages

References:

- **Chapter 6** of *Concepts in programming languages* by J. C. Mitchell. CUP, 2003.
- **Sections 4.9 and 8.6** of *Programming languages: Concepts & constructs* by R. Sethi (2ND EDITION). Addison-Wesley, 1996.

Types in programming

- A *type* is a collection of computational entities that share some common property.
- There are three main uses of types in programming languages:
  1. naming and organizing concepts,
  2. making sure that bit sequences in computer memory are interpreted consistently,
  3. providing information to the compiler about data manipulated by the program.

- Using types to organise a program makes it easier for someone to read, understand, and maintain the program. Types can serve an important purpose in documenting the design and intent of the program.
- Type information in programs can be used for many kinds of optimisations.

Type systems

A *type system* for a language is a set of rules for associating a type with phrases in the language.

Terms strong and weak refer to the effectiveness with which a type system prevents errors. A type system is *strong* if it accepts only *safe* phrases. In other words, phrases that are accepted by a strong type system are guaranteed to evaluate without type error. A type system is *weak* if it is not strong.

Type safety

A programming language is *type safe* if no program is allowed to violate its type distinctions.

Safety	**Example language**	**Explanation**
Not safe	C, C++	Type casts, pointer arithmetic
Almost safe	Pascal	Explicit deallocation; dangling pointers
Safe	LISP, SML, Smalltalk, Java	Type checking

Type checking

A *type error* occurs when a computational entity is used in a manner that is inconsistent with the concept it represents.

Type checking is used to prevent some or all type errors, ensuring that the operations in a program are applied properly.

Some questions to be asked about type checking in a language:

- Is the type system *strong* or *weak*?
- Is the checking done *statically* or *dynamically*?
- How *expressive* is the type system; that is, amongst safe programs, how many does it accept?

Static and dynamic type checking

Run-time type checking: The compiler generates code so that, when an operation is performed, the code checks to make sure that the operands have the correct types.

Examples: LISP, Smalltalk.

Compile-time type checking: The compiler checks the program text for potential type errors.

Example: SML.

NB: Most programming languages use some combination of compile-time and run-time type checking.

Static vs. dynamic type checking

Main trade-offs between compile-time and run-time checking:

Form of type checking	**Advantages**	**Disadvantages**
Run-time	Prevents type errors	Slows program execution
Compile-time	Prevents type errors Eliminates run-time tests Finds type errors before execution and run-time tests	May restrict programming because tests are *conservative*

Type checking in ML

Idea

Given a context Γ, an expression e, and a type τ, decide whether or not the expression e is of type τ in context Γ.

Examples:

- $$\frac{\Gamma \vdash e_1 : \texttt{bool} \qquad \Gamma \vdash e_2 : \texttt{bool}}{\Gamma \vdash e_1 \ \texttt{orelse} \ e_2 : \texttt{bool}}$$

$$\mathsf{TC}(\Gamma, e_1 \ \texttt{orelse} \ e_2, \tau) = \begin{cases} \mathsf{TC}(\Gamma, e_1, \texttt{bool}) \wedge \mathsf{TC}(\Gamma, e_2, \texttt{bool}) & \text{, if } \tau = \texttt{bool} \\ \text{false} & \text{, otherwise} \end{cases}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 * \tau_2}$$

$$\mathsf{TC}(\Gamma, (e_1, e_2), \tau)$$

$$= \begin{cases} \mathsf{TC}(\Gamma, e_1, \tau_1) \wedge \mathsf{TC}(\Gamma, e_2, \tau_2) & \text{, if } \tau = \tau_1 * \tau_2 \\ \text{false} & \text{, otherwise} \end{cases}$$

Type equality

The question of *type equality* arises during type checking.

? What does it mean for two types to be equal!?

Structural equality. Two type expressions are *structurally equal* if and only if they are equivalent under the following three rules.

SE1. A type name is structurally equal to itself.

SE2. Two types are structurally equal if they are formed by applying the same type constructor to structurally equal types.

SE3. After a type declaration, say `type n = T`, the type name `n` is structurally equal to `T`.

Name equality.

Pure name equality. A type name is equal to itself, but no constructed type is equal to any other constructed type.

Transitive name equality. A type name is equal to itself and can be declared equal to other type names.

Type-expression equality. A type name is equal only to itself. Two type expressions are equal if they are formed by applying the same constructor to equal expressions. In other words, the expressions have to be identical.

Examples:

- **Type equality in Pascal/Modula-2.** Type equality was left ambiguous in Pascal. Its successor, Modula-2, avoided ambiguity by defining two types to be *compatible* if
  1. they are the same name, or
  2. they are `s` and `t`, and `s = t` is a type declaration, or
  3. one is a subrange of the other, or
  4. both are subranges of the same basic type.
- **Type equality in C/C++.** C uses structural equivalence for all types except for records (`struct`s). `struct` types are named in C and C++ and the name is treated as a type, equal only to itself. This constraint saves C from having to deal with recursive types.

Type declarations

There are two basic forms of type declarations:

Transparent. An alternative name is given to a type that can also be expressed without this name.

Opaque. A new type is introduced into the program that is not equal to any other type.

Type inference

- *Type inference* is the process of determining the types of phrases based on the constructs that appear in them.
- An important language innovation.
- A cool algorithm.
- Gives some idea of how other static analysis algorithms work.

Type inference in ML
Idea

Typing rule:

$$\frac{}{\Gamma \vdash x : \tau} \quad \text{if } x : \tau \text{ in } \Gamma$$

Inference rule:

$$\frac{}{\Gamma \vdash x : \gamma} \quad \boxed{\gamma \approx \alpha} \text{ if } x : \alpha \text{ in } \Gamma$$

Typing rule:

$$\frac{\Gamma \vdash f : \sigma \texttt{->} \tau \qquad \Gamma \vdash e : \sigma}{\Gamma \vdash f(e) : \tau}$$

Inference rule:

$$\frac{\Gamma \vdash f : \alpha \qquad \Gamma \vdash e : \beta}{\Gamma \vdash f(e) : \gamma} \quad \boxed{\alpha \approx \beta \texttt{->} \gamma}$$

Typing rule:

$$\frac{\Gamma, x : \sigma \vdash e : \tau}{\Gamma \vdash (\texttt{fn}\ x \texttt{ => } e) : \sigma \texttt{ -> } \tau}$$

Inference rule:

$$\frac{\Gamma, x : \alpha \vdash e : \beta}{\Gamma \vdash (\texttt{fn}\ x \texttt{ => } e) : \gamma} \quad \boxed{\gamma \approx \alpha \texttt{ -> } \beta}$$

Example:

$$\dfrac{\dfrac{\dfrac{\dfrac{\surd}{f : \alpha_1, x : \alpha_3 \vdash f : \alpha_5} \qquad \dfrac{\dfrac{\surd}{f : \alpha_1, x : \alpha_3 \vdash f : \alpha_7} \qquad \dfrac{\surd}{f : \alpha_1, x : \alpha_3 \vdash x : \alpha_8}}{f : \alpha_1, x : \alpha_3 \vdash f(x) : \alpha_6}}{f : \alpha_1, x : \alpha_3 \vdash f(f(x)) : \alpha_4}}{f : \alpha_1 \vdash \texttt{fn}\ x \Rightarrow f(f(x)) : \alpha_2}}{\vdash \texttt{fn}\ f \Rightarrow \texttt{fn}\ x \Rightarrow f(f(x)) : \alpha_0}$$

$$\boxed{\begin{array}{c} \alpha_0 \approx \alpha_1 \rightarrow \alpha_2\,, \quad \alpha_2 \approx \alpha_3 \rightarrow \alpha_4\,, \quad \alpha_5 \approx \alpha_6 \rightarrow \alpha_4\,, \quad \alpha_5 \approx \alpha_1 \\ \alpha_7 \approx \alpha_8 \rightarrow \alpha_6\,, \quad \alpha_7 \approx \alpha_1\,, \quad \alpha_8 \approx \alpha_3 \end{array}}$$

Solution: $\alpha_0 = (\alpha_3 \rightarrow \alpha_3) \rightarrow \alpha_3 \rightarrow \alpha_3$

Polymorphism

Polymorphism, which literally means “having multiple forms”, refers to constructs that can take on different types as needed.

Forms of polymorphism in contemporary programming languages:

Parametric polymorphism. A function may be applied to any arguments whose types match a type expression involving type variables.

Parametric polymorphism may be:

Implicit. Programs do not need to contain types; types and instantiations of type variables are computed.
Example: SML.

Explicit. The program text contains type variables that determine the way that a construct may be treated polymorphically.

Explicit polymorphism often involves explicit instantiation or type application to indicate how type variables are replaced with specific types in the use of a polymorphic construct.

Example: C++ templates.

Ad hoc polymorphism or overloading. Two or more implementations with different types are referred to by the same name.

Subtype polymorphism. The subtype relation between types allows an expression to have many possible types.

let-polymorphism

- The standard sugaring

$$\texttt{let val } x = v \texttt{ in } e \texttt{ end} \quad \mapsto \quad \big(\texttt{fn } x \texttt{ => } e\big)(v)$$

  does not respect ML type checking.

  For instance

$$\texttt{let val f = fn x => x in f(f) end}$$

  type checks, whilst

$$\big(\texttt{fn f => f(f)}\big)(\texttt{fn x => x})$$

  does not.

- Type inference for let-expressions is involved, requiring *type schemes*.

Polymorphic exceptions

Example: Depth-first search for finitely-branching trees.

```
datatype
  'a FBtree = node of 'a * 'a FBtree list ;
fun dfs P (t: 'a FBtree)
  = let
      exception Ok of 'a;
      fun auxdfs( node(n,F) )
        = if P n then raise Ok n
          else foldl (fn(t,_) => auxdfs t) NONE F ;
    in
      auxdfs t handle Ok n => SOME n
    end ;
val dfs = fn : ('a -> bool) -> 'a FBtree -> 'a option
```

When a *polymorphic exception* is declared, SML ensures that it is used with only one type. The type of a top level exception must be monomorphic and the type variables of a local exception are frozen.

Consider the following nonsense:

```
exception Poly of 'a ;    (*** ILLEGAL!!! ***)
(raise Poly true) handle Poly x => x+1 ;
```