Type systems
We have seen types in the course already

- types as a description of a *data structure*
  - to generate code to allocate and construct variables
  - every identifier comes with a type
    variable declarations char c

- *type checking* (C, Pascal, ...)
  - detecting runtime errors: bad usage of variables
  - checking function calls \( f(t_1, \ldots, t_n) \)
    - functions have types of the form \( (\tau_1 \times \cdots \times \tau^n) \rightarrow \tau' \)
    - the \( \tau_i, \tau' \) must be provided by the programmer
  - some flexibility: subtyping char \( \leq \) int

- types can also be used for *program analysis*
  - Hoare triples as types?
    \{A\} p \{B\} can be written \( p : A \rightarrow B \) *(assigning a type to a whole program)*

- what *(inert)* data structures *are* vs what programs *do* move to *richer types*
The language for types

- a lot of research in programming languages focuses on *type systems*
- analyse the behaviour of programs
  - absence of runtime errors
  - provide guarantees (termination, non-interference, complexity, protocol compliance, ..)
- two languages, for *programs* and for *types*
  - the notion of function is central
  - types for functions: $\tau_1 \rightarrow \tau_2$

programs $\text{FUN}$
types $\tau ::= \text{int} \mid \tau_1 \rightarrow \tau_2$
Typing: definition

on the board
Exercise:

typing the CPS transform
Types in functional languages

- **typing guarantees** *absence of runtime errors*
  
  **Theorem:** if $\Gamma \vdash e : \tau$, then running $e$ will not generate a bad application of a function to an argument.

- language design: functions, and function types, are *primitive* in functional languages
  
  - less constructs in the language of types, (no `struct`, `typedef`)
  
  - but the language is somehow *richer*
  
  - promoting the use of functions: applications everywhere
  
  - more typing, “hence” less bugs

- **ML** also has *polymorphic types*: `’a -> (’a -> ’b) -> ’b`
  
  - not only `:=` and `=` (as seen before)
  
  - the programmer can define polymorphic functions
  
  - `int -> (bool->int) -> bool` and `int -> (int->int) -> int` are instances of the type above

- types for *functional programming languages* have their origins in logic/proof theory
  
  - $\to$ stands for $\Rightarrow$
  
  - but $\forall$ (as in `fun z -> z : ’a -> ’a`) does not really stand for $\forall$
  
  - $\forall$ is rather *dependent types*, as in Coq’s type system
Type inference
Type inference as in ML / Haskell

- the core of ML/Haskell  
  (basically, Fun)
  - not modules/functors
- no need for any annotation
  - input: a bare program
  - output: a type, or an error message
    the type, actually  (there are “principal types”)

- how does it work?
  1. constraint generation
  2. constraint solving

  Theorem: the generated constraint problem has a solution iff
  the program has a principal type.

- this approach, known as the Hindley-Milner approach, is global
Partial type inference

- issues in type inference
  - decidability
  - to a lesser extent, complexity
  - being intuitive / predictable
    readability of error messages

- some languages adopt *partial type inference*
  - pragmatical reasons
    - writing type annotations can be a good habit
    - but we don’t want to write annotations which are
      - silly *nothing informative*
      - common *ok for rare situations*
  - theoretical reasons
    the type system is so rich (objects, subtyping, modules, polymorphism, ..) that we cannot decide inference
    Scala, ML, Coq

- an example: type inference in Scala
  - builds on Java: Java users praise type inference
  - is close to a functional language:
    functional programmers blame partiality
Bidirectional type inference

on the board
Programming languages zoology
Things left to say

- exam
  - all of the course (C+AP)
  - written documents (notes, books) are allowed

- évaluation