IA010: Principles of Programming Languages

Types

Achim Blumensath
blumens@fi.muni.cz

Faculty of Informatics, Masaryk University, Brno

Static and dynamic typing

Static typing: types of expressions are computed at **compile-time**

Dynamic typing: **runtime** values are tagged with type information

Static and dynamic typing

Static typing: types of expressions are computed at **compile-time**

Dynamic typing: **runtime** values are tagged with type information

Dynamic typing

- is slow
- only catches type errors in executed code
- more permissive and (sometimes) convenient

$\Rightarrow$ mostly useful (if at all) in scripting languages

Static and dynamic typing

Static typing

- **stricter,** catches more errors
- can **prove** that the program is free of type errors
- no **runtime overhead**
- can be **inconvenient:** might need additional code/annotations
- not all properties can be checked statically (array bounds)
- error messages from the type checker can be **hard to understand**
- type annotations help **document** the code
- types can be used to **control** the behaviour of code (overloading)
- indispensable for **serious** software development:
  - proves the absence of certain errors
  - helps with interface design
  - helps with refactoring
- advantages apply mostly to **symbolic** computations, less so to **numeric** ones

Type annotations

New syntax

$$\langle expr \rangle ::= \ldots \mid \texttt{let}\ \langle id \rangle : \langle type \rangle = \langle expr \rangle\ ;\ \langle expr \rangle$$
$$\mid \texttt{let}\ \langle id \rangle\ (\ \langle id \rangle : \langle type \rangle\ )\ : \langle type \rangle\ \{\ \langle expr \rangle\ \};\ \langle expr \rangle$$
$$\mid \texttt{fun(}\ \langle id \rangle : \langle type \rangle\ )\ : \langle type \rangle\ \{\ \langle expr \rangle\ \}$$

Basic types

```
int
a -> b
type foo = | A(a,b,...) | ... | Z(c,d,...);

let fac(n: int) : int {
  if n == 0 then 1 else n * fac(n-1)
};

let compose(f: int -> int, g: int -> int): int -> int {
  fun (x: int) { f(g(x)) }
};
```

Common types

Basic types

- integers (signed/unsigned, various precisions, including arbitrary precision)
- floating point numbers, decimal numbers (`0000.00`), arbitrary precision rational numbers
- integer ranges (`1..100`)
- enumerations (`enum colours { Red, Green, Blue, Yellow }`)
- booleans
- characters
- strings
- the empty type, the unit type

Common types

Composite types

- arrays
- pointers, references
- functions, procedures
- records, tuples
- unions, variants
- lists, maps, dictionaries

Arrays

Definition

homogeneous collection indexed by an ordinal type

Possible variations

- **index type:** integers, ranges, enumeration types
- **dimension:** 1-dimensional, many-dimensional

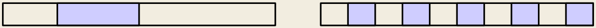
Remarks

- FORTRAN is famous for its extensive array support
- **bounds checking** must be done dynamically

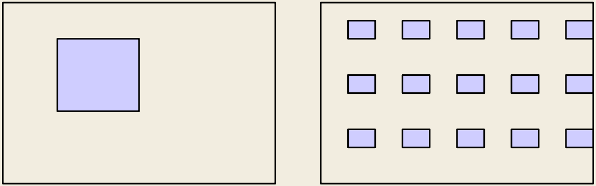
Array slices

(not necessarily contiguous) subsets of an array

One-dimensional



Two-dimensional



Product and sum types

Product types

inhomogeneous collection of elements of a fixed size: tuples, records

```
type triple = int * int * int;
type vector = [ x : float, y : float, z : float ];
```

Product and sum types

Product types

inhomogeneous collection of elements of a fixed size: tuples, records

```
type triple = int * int * int;
type vector = [ x : float, y : float, z : float ];
```

Sum types

alternative between several types: tagged unions, variant types, algebraic types

```
type int_list = | Nil | Cons(int, int_list);
type expr = | Num(int) | Plus(expr, expr) | Times(expr, expr);
type nat = | Zero | Suc(nat)
```

Unit and void

Unit type

type with a single value

```
type unit = | Nothing;
```

- Can be used for functions that do not take arguments or do not return a value.

Unit and void

Unit type

type with a single value

```
type unit = | Nothing;
```

- Can be used for functions that do not take arguments or do not return a value.

Void type

type with no value

```
type void = ;
```

- When used as argument type of a function, you cannot call it.
- When used as return type, the function does not terminate.

Recursive types

types who are used in their own definition

```
type expr = | Num(int) | Plus(expr, expr) | Times(expr, expr);
```

- usually the recursion is via a pointer
- some languages allow arbitrary recursive definitions

```
type t = t -> t
```

Example: recursion operator

```
type b = b -> a;
let rec(f : a -> a) : a =
  (fun (x : b) : a { f(x(x)) })
  (fun (x : b) : a { f(x(x)) });
```

Type equivalence

Name equivalence

Two types are equivalent if they have the same name.

Structural equivalence

Two types are equivalent if they have the same definition.

```
type vector = [ x : int, y : int ];
type pair   = [ x : int, y : int ];
type pair2  = [ y : int, x : int ];
```

Type conversions

Cast explicit conversion

Coercion implicit conversion

- convenient
- can make code hard to understand

Variations

- If the memory representation is the same, we can just change the type.
- Otherwise, we have to convert the value.
- We need a runtime check, if not every value can be converted to the new type.
- Some languages support non-converting type casts.

Polymorphism

Some code works without changes for several types.

Polymorphism

Some code works without changes for several types.

What is the type of

```
fun (x) { x }
```

Polymorphism

Some code works without changes for several types.

What is the type of

```
fun (x) { x }
```

```
int -> int
```

Polymorphism

Some code works without changes for several types.

What is the type of

```
fun (x) { x }
```

```
int -> int
float -> float
```

Polymorphism

Some code works without changes for several types.

What is the type of

```
fun (x) { x }

int -> int
float -> float
string -> string
(int -> unit) -> (int -> unit)
...
```

Forms of polymorphism

- **Ad-hoc polymorphism** (also called **overloading**)
- **Parametric polymorphism** (in ML-like languages)
- **Subtyping polymorphism** (in object-oriented languages)

Forms of polymorphism

- **Ad-hoc polymorphism** (also called **overloading**)
- **Parametric polymorphism** (in ML-like languages)
- **Subtyping polymorphism** (in object-oriented languages)

Ad-hoc polymorphism

- several versions of a function
- selection depending on argument types

```
+ : int -> int -> int
+ : float -> float -> float
+ : string -> string -> string
```

- **Advantages:**
  - flexible
- **Disadvantages:**
  - one has to write a separate function for every type
  - program can become harder to understand

Forms of polymorphism

- **Ad-hoc polymorphism** (also called **overloading**)
- **Parametric polymorphism** (in ML-like languages)
- **Subtyping polymorphism** (in object-oriented languages)

Parametric polymorphism

- types can contain **type variables**

```
map : (a -> b) -> list(a) -> list(b)
```

- **Advantages:**
  - simple, clean, easy to understand, few drawbacks
- **Disadvantages:**
  - less flexible than ad-hoc polymorphism

Forms of polymorphism

- **Ad-hoc polymorphism** (also called **overloading**)
- **Parametric polymorphism** (in ML-like languages)
- **Subtyping polymorphism** (in object-oriented languages)

Subtyping polymorphism

- a is a **subtype** of b if every value of type a can be used where type b is expected. (a is a **specialisation** of b, b is more **general**.)
- basis of object-oriented programming
- makes the type system much more complicated

Type inspection

- branching on values of type variables
- at compile-time or runtime
- add power of overloading to parametric polymorphism

```
let serialise(value) {
  case type_of(value)
  | int    => int_to_string(value)
  | bool   => bool_to_string(value)
  | string => sanitise_string(value)
  | cons   => "cons(" ++ serialise(fst(value)) ++ ","
                      ++ serialise(snd(value)) ++ ")"
  | ...
};
```

Type inference

Problem

Writing type annotations is tedious, especially if the types are complex and long.

Solution

The compiler derives the types automatically without annotation.

- Developed for ML by Damas, Hindley, and Milner.
- For languages with more complex type systems, this is only partially possible.

Idea

Given an expression, look at all subexpressions and create a system of type equations.

```
let twice(x) { 2 * x };
```

Type inference

Problem

Writing type annotations is tedious, especially if the types are complex and long.

Solution

The compiler derives the types automatically without annotation.

- Developed for ML by Damas, Hindley, and Milner.
- For languages with more complex type systems, this is only partially possible.

Idea

Given an expression, look at all subexpressions and create a system of type equations.

```
let twice(x) { 2 * x };

let compose(f, g) { fun (x) { f(g(x)) } };
```

Type inference

Problem

Writing type annotations is tedious, especially if the types are complex and long.

Solution

The compiler derives the types automatically without annotation.

- Developed for ML by Damas, Hindley, and Milner.
- For languages with more complex type systems, this is only partially possible.

Idea

Given an expression, look at all subexpressions and create a system of type equations.

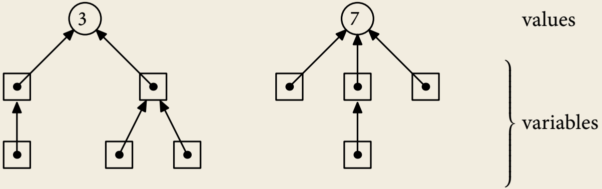
```
let twice(x) { 2 * x };

let compose(f, g) { fun (x) { f(g(x)) } };

let sum(lst) = fold(fun (acc,x) { acc + x }, 0, lst);
```

Unification

solving a type equation $s = t$

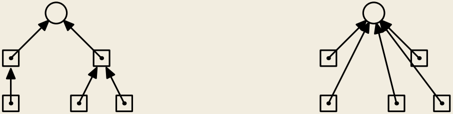
$$
\begin{array}{rcl}
x = t & \rightsquigarrow & x := t \\
s = x & \rightsquigarrow & x := s \\
s \to s' = t \to t' & \rightsquigarrow & s = t \wedge s' = t' \\
c(s_1, \ldots, s_n) = c(t_1, \ldots, t_n) & \rightsquigarrow & s_1 = t_1 \wedge \cdots \wedge s_n = t_n \\
s = t & \rightsquigarrow & \text{failure}
\end{array}
$$

Union-Find-Algorithm



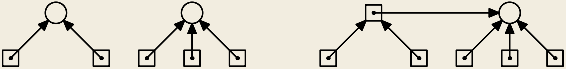
find : *variable* → *value*

- follows pointers to the root and creates shortcuts



union : (*variable* × *variable*) → *unit*

- links roots by a pointer



Advantages of type inference

- convenient, less friction
- finds the **most general type**
- automatically introduces polymorphism

Disadvantages

- Type annotations serve as documentation.
- Error messages from the type checker are more complicated.
  (It hides **where** the type error occurred.)

Advanced topics

- **linear types:** types for resource management
- **dependent types:** types with arguments
- **gradual typing:** mixing dynamic and static typing
- types for **software verification**