Type errors, variables and functions

Programmazione Funzionale
2024/2025
Università di Trento
Chiara Di Francescomarino

Today

- Rules on type correctness
- Type errors and conversion in ML
- Names, denotable objects and variables
- Variables in ML
- Complex types in ML
- Functions in ML

Type systems

- The type system of a language:
    1. Predefined types
    2. Mechanisms to define new types
    3. Control mechanisms
        - Equivalence
        - Compatibility
        - Inference
    4. specification of whether types are statically or dynamically checked

Rules on type correctness

Type equivalence

Type equivalence

- Two types `T` and `S` are equivalent if every object of type `T` is also of type `S`, and vice versa
- Two rules for type equivalence
  - Equivalence by name: the definition of a type is opaque
  - Structural equivalence: the definition is transparent

Equivalence by name

- Two types are (strongly) equivalent by name if they have the same name (Java)

Which of the following are strongly equivalent by name?

```
type T1 = 1..10;
type T2 = 1..10;
type T3 = int;
type T4 = int;
```

- None of the four
- Too restrictive

- Loose or weak equivalence by name (Pascal)
  - A declaration of an alias of a type generates a new name, not a new type
  - T3 and T4 are names of the same type
- Defined with reference to a specific program, not in general

Structural Equivalence

- Two types are structurally equivalent if they have the same structure: substituting names for the relevant definitions, identical types are obtained.
- Structural equivalence between types is the minimal equivalence relation that satisfies:
  - A type name is equivalent to itself
  - If `T` is defined as `type T = expression`, then `T` is equivalent to `expression`
  - Two types constructed using the same type constructor applied to equivalent types, are equivalent

Structural Equivalence Examples

Which of the following are structurally equivalent?

```
type T1 = int;
type T2 = char;
type T3 = struct{
    T1 a;
    T2 b;
}
type T4 = struct{
    int a;
    char b;
}
```

```
type S = struct{
    int a;
    int b;
}
type T = struct{
    int n;
    int m;
}
type U = struct{
    int m;
    int n;
}
```

- Some aspects are clear
  - T3 and T4 are structurally equivalent
- Other aspects are less clear
  - S, T and U have field names or order that are different: are they equivalent?
  - Usually no, yes T and U for ML.
- Defined in general – not specifically for a program

Type equivalence in languages

- Combination or variant of the two equivalence rules
  - Pascal → weak equivalence by name
  - Java → equivalence by name – except for arrays with structural equivalence
  - C → structural equivalence for arrays and types defined with typedef but equivalence by name for records and unions
  - ML → structural equivalence except for types defined with `datatype`

Compatibility and conversion

Compatibility

- `T` is compatible with `S` if objects of type `T` can be used in contexts where objects of type `S` are expected
- Example: `int n; float r; r=r+n` in some languages
- In many languages compatibility is used for checking the correctness of:
  - Assignments (right-hand type compatible with left-hand),
  - parameter passing (actual parameter type compatible with formal one), …
- Compatibility is reflexive and transitive but it is not symmetric
  - E.g., compatibility between `int` and `float` but not viceversa in some languages

Compatibility

- The definition depends on the language.
- `T` can be compatible with `S` if
  - `T` and `S` are equivalent
  - The values of `T` are a subset of the values of `S` (interval)
  - All the operations on values of `S` can be performed on values of `T` (extension of record) – sort of subtype
  - There is a natural correspondence between values of `T` and values of `S` (`int` to `float`)
  - The values of `T` can be made to correspond to some values of `S` (`float` to `int` with truncating, rounding)

Type conversion

- If `T` is compatible with `S`, there is some type conversion mechanism.
- The main ones are:
  - Implicit conversion, also called coercion. The language implementation does the conversion, with no mention at the language level
  - Explicit conversion, or cast, when the conversion is mentioned in the program

Coercion

- Coercion indicates, in a case of compatibility, how the conversion should be done
- Three possibilities for realizing the type conversion. The types are different, but
  - Same values and same storage representation for values of $T \subseteq S$. E.g., types that are structurally the same, but have different names
    - Conversion only at compile time → no code to be generated
  - Different values, but the common values have the same representation. E.g., integer interval and integer
    - Code for dynamic control when there is an intersection
  - Different representations for the values. E.g., reals and integers
    - Code for the conversion

Cast

- In certain cases, the programmer must insert explicit type conversion (C, Java: cast)

  `S s = (S) T`

  For example `r = (float) n` and `n=(int) r`
- Cases similar to coercion
- Not every explicit conversion is allowed
  - Only when the language knows how to do the conversion
  - Can always be done when types are compatible (useful for documentation)
- Modern languages prefer cast to coercion.

Type errors and conversion in ML

Type errors in ML

Type errors

```
> 1 + 2;
val it = 3: int

> 1.0 + 2.0;
val it = 3.0: real

> 1 + 2.0;
poly: : error: Type error in function application.
Function: + : int * int -> int
Argument: (1, 2.0) : int * real
Reason:
Can't unify int (*In Basis*) with real (*In Basis*)
(Different type constructors)
Found near 1 + 2.0
Static Errors
```

What does the error message mean?

Type errors

- Type of `+` is `int * int -> int`
- Actually, + can have different types, but, based on the first argument, ML decides on the integer version
- What if we have a real as “first argument”?

```
> 1.0 + 2;
poly: : error: Type error in function application.
Function: + : real * real -> real
Argument: (1.0, 2) : real * int
Reason:
Can't unify int (*In Basis*) with real (*In Basis*)
(Different type constructors)
Found near 1.0 + 2
Static Errors
```

- In this case, ML decides that we want real addition: `real * real -> real`
- In both cases, the second argument does not match the first

Other type errors

```
> #"a" ^ "bc";
poly: : error: Type error in function application.
Function: ^ : string * string -> string
Argument: (#"a", "bc") : char * string

> 1/2;
poly: : error: Type error in function application.
Function: / : real * real -> real
Argument: (1, 2) : int * int

> if 1<2 then #"a" else "bc";
poly: : error: Type mismatch between then-part and else-part.
Then: #"a" : char
Else: "bc" : string
```

Type conversion in ML

How to deal with these issues?

Type conversion

Conversion between integers and reals

- From integers to reals
  - `real`: convert from integer to real

- From reals to integers → four possibilities:
  - `floor`: round down
  - `ceil`: round up
  - `round`: nearest integer
  - `trunc`: truncate

Values halfway between two integers go towards the closest even value.
For example:
> round(3.5);
val it = 4: int
> round(4.5);
val it = 4: int

Examples

```
> real(4);
val it = 4.0: real
> real 4;
val it = 4.0: real

> floor(3.5);
val it = 3: int
> ceil(3.5);
val it = 4: int
> round(3.5);
val it = 4: int
> trunc(3.5);
val it = 3: int

> floor(~3.5);
val it = ~4: int
> trunc(~3.5);
val it = ~3: int
```

`trunc` and `ceil` are different on positive values, while `trunc` and `floor` on negative values.

Conversion between characters and integers

- `ord`: from character to integer
- `chr`: from integer to character

```
> ord #"a";
val it = 97: int
> ord #"a" - ord #"A";
val it = 32: int

> chr 97;
val it = #"a": char
```

Integers in the range 0 to 255. Outside the interval Exception – Chr raised

Conversion from characters to strings

- `str`: from character to string

```
> str #"a";
val it = "a": string
```

In other words …

- No automatic conversion of types
  - 5+7 and 5.0+7.0 are correct, resulting in `int` and `real`
  - 5+7.0 is wrong
- Conversion between types
  - `real`: integer to real
  - `ceil, floor, round` and `trunc`: real to integer
  - `ord`: character to integer
  - `chr`: reverse direction, i.e., integer to character
  - `str`: character to string

Some more questions

How to participate?

Copy participation link

1. Go to wooclap.com
2. Enter the event code in the top banner

Event code

AKESQF

Names, Denotable Objects and Variables

What is a name?

- Sequence of characters used to *denote* something else
  - `const p = 3.14` (object denoted: the constant 3.14)
  - `x` (object denoted: a variable)
  - `void f()(...)` (object denoted: the definition of `f`)
- In programming languages, the names are often identifiers (alphanumerical tokens)
- The use of a name serves to indicate the object to denote
  - Symbolic identifiers, that are easier to remember
  - Abstractions (data or control)

Names and denotable objects

- A name and the object it denotes are not the same thing.
  - A name is just a character string
  - Denotation can be a complex object (variable, function, type, etc.)
  - A single object can have more than one name ("aliasing")
  - A single name can denote different objects at different times
- We use "the variable `fie`" or "the function `foo`" as abbreviations for "the variable with the name `fie`" and "the function with the name `foo`"

What is a denotable object?

- An object that can be associated with a name
- Names defined by the user: variables, formal parameters, procedures, types defined by the user, labels, modules, constants defined by the user, exceptions
- Names defined by the language: primitive types, primitive operations, predefined constants
- Binding: association between name and denotable object

Environment

- Not all the associations between names and denotable objects are fixed once and for all.
- Environment: the collection of associations between names and denotable objects that exist at runtime at a specific point in a program and at a specific moment in the execution
- Declaration: a mechanism (implicit or explicit) that creates an association in an environment.

```
int x;
int f(){
    return 0;
}
type T = int;
```

Variable declaration

Function declaration

Type declaration

Names and denoted objects

- The same name can denote different objects at different positions in the program

```
{int fie;
 fie = 2;
     {char fie;
          fie = a;
     }
}
```

Integer

Char

- A single object is denoted by more than one name in different environments
  - A variable passed by reference to a procedure

```
procedure P (var &X:integer);
begin
...
end;

var A:integer;
P(A);
```

X

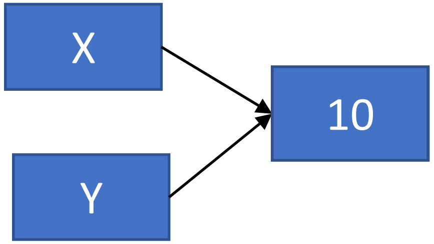
A

Names and denoted objects

- Aliasing: different names that denote the same object in the same environment
  - Pointers (integer pointers)

```
int *X, *Y; // X,Y pointers to integers
X = (int *) malloc (sizeof (int)); // allocate heap
memory
*X = 5; // * dereference
Y = X; // Y points to the same object as X
*Y = 10;
write(*X);
```

10



Variables

- Variable: in mathematics, an unknown that can take values from a predefined domain (that cannot be modified anymore)
- In computer science it depends on the paradigm
- In classical imperative languages (Pascal, C, Ada, etc.,):
  - modifiable variable: the value can be modified
  - A container of values that have a name

`x` $\boxed{3}$

The variable with name `x`
The variable `x`

Different models

- In some imperative languages (object-oriented ones)
  - a variable is a reference to a value, which has a name and is stored in the heap
  - similar to pointers but without the possibility to directly accessing the location
- In logical languages
  - a variable can be modified only under certain conditions (instantiation)
- In pure functional languages (Lisp, ML, Haskell, SmallTalk)
  - a variable is an identifier that stands for a value (not modifiable), as in mathematics

Variables in ML

Variables

- Environment: Set of pairs of identifiers and values
- It is possible to add an identifier to the environment and bind it to a value
- Environment is modified by val-declarations (a sort of assignment)

```
val <name> = <value>;
val <name>:<type> = <value>;
val <name> = <expression>;
```

- Example

```
> val pi = 3.14159;
val pi = 3.14159: real
> val v = 10.0/2.0;
val v = 5.0: real
```

- Note that the response has the name of the variable, rather than it.

Variables

- We do not need to specify the type
- Variables cannot be modified!
- `val` creates an association between a name and a variable
- A new declaration creates a new variable, and does not change the value of the existing one

```
> val pi = 3;
val pi = 3: int
> val pi = 3.14159;
val pi = 3.14159: real
```

pi	3.14159
pi	3
…	…

  create two variables with name `pi`, where the second hides the first
- Note that even the type can change
- Old definition is still there, but (at least at the top-level environment) it is no longer accessible

Variable identifiers

- Any “reasonable" sequence of alphanumeric characters
- We won't use things like this. . .

```
> val $$$ = "ab";
val $$$ = "ab": string
```

Examples

```
> val pi = 3.14159;
val pi = 3.14159: real
> val radius = 4.0;
val radius = 4.0: real
> pi * radius * radius;
val it = 50.26544: real
> val area = pi * radius * radius;
val area = 50.26544: real
```

What does the environment contain after these command?

```
> val a = 3;
val a = 3: int
> val b = 98.6;
val b = 98.6: real
> val a = "three";
val a = "three": string
> val c = a ^ str(chr(floor(b)));
al c = "threeb": string
```

$(a_1, a_2, \ldots, a_n)$

$[a_1, a_2, \ldots, a_n]$

Complex types in ML

$(a_1, a_2, \ldots, a_n)$

Tuples in ML

Tuples

- We have already seen something when looking at the operators
- Example

```
> val t = (1, 2, 3);
val t = (1, 2, 3): int * int * int
```

- We can mix types in tuple definitions

```
> val t = (4, 5.0, "six");
val t = (4, 5.0, "six"): int * real * string
```

- We can also use a complex type instead of a simple one

```
> val t = (1, (2, 3, 4));
val t = (1, (2, 3, 4)): int * (int * int * int)
```

Accessing tuple components:

- Let us consider

```
> val t = (4, 5.0, "six");
```

- Then

```
> #1 (t);
val it = 4: int
> #3 (t);
val it = "six": string
> #4 (t);
poly: : error: Type error in function application.
Function: #4 : 'a * 'b * 'c * 'd -> 'd
Argument: (t) : int * real * string
Reason: Can't unify {4: 'a} to int * real * string (Field 4
missing)
```

Generic type: the specific type is determined only when we use it.

$$[a_1, a_2, \ldots, a_n]$$

Lists in ML

Lists

- Represented with square brackets
- Example

```
> [1,2,3];
val it = [1, 2, 3]: int list
```

- Note that all elements of a list (unlike tuples) must be of the same type

More examples

```
> [1.0,2.0];
val it = [1.0, 2.0]: real list

> [1,2.0];
poly: : error: Elements in a list have different types.
Item 1: 1 : int
Item 2: 2.0 : real
Reason:
Can't unify int (*In Basis*) with real (*In Basis*)
(Different type constructors)

> [];
val it = []: 'a list
```

Generic type: the specific type is determined only when we use it.

Note that for an empty list, ML cannot determine the type of the elements

Head of a list: `hd`

- Head: first element of a list

```
> val L = [2,3,4];
val L = [2, 3, 4]: int list
> val M = [5];
val M = [5]: int list

> hd(L);
val it = 2: int
> hd(M);
val it = 5: int
```

Tail of a list: `tl`

- Tail: all the rest

```
> L;
val it = [2, 3, 4]: int list
> M;
val it = [5]: int list

> tl (L);
val it = [3, 4]: int list
> tl (M);
val it = []: int list
```

Note that ML can determine the type of this empty list

Concatenation of lists: @

- Example

```
> [1,2] @ [3,4];
val it = [1, 2, 3, 4]: int list
```

- Note that both lists must be of the same type

```
> [1,2] @ ["a","b"];
poly: : error: Type error in function application.
Function: @ : int list * int list -> int list
Argument: ([1, 2], ["a", "b"]) : int list * string list
Reason:
Can’t unify int (*In Basis*) with string (*In Basis*)
(Different type constructors)
```

- Two different types of concatenation
  - ^: Strings
  - @: List

Cons: ::

- An operator that takes an element of type 'a and a list of type 'a list and combines them

```
> 2 :: [3,4];
val it = [2, 3, 4]: int list
> 2 :: nil;
val it = [2]: int list
> 2 :: [];
val it = [2]: int list
```

nil is the same as the empty list []

- :: is right associative

```
> 1 :: 2 :: 3 :: nil;
val it = [1, 2, 3]: int list
> (1 :: (2:: (3::nil)));
val it = [1, 2, 3]: int list
```

- The other way would make no sense

Strings to lists: `explode`

```
> explode ("abcd");
val it = [#"a", #"b", #"c", #"d"]: char list
> explode ("");
val it = []: char list
```

Lists to strings: `implode`

```
> implode ([ #"a", #"b", #"c", #"d"]);
val it = "abcd": string
> implode (nil);
val it = "": string
> implode (explode ("xyz"));
val it = "xyz": string
```

The ML type system

- Basic types int, real, bool, char, string
- Complex types: For now, 2 constructors:
  - `T1 * T2 * ... * Tn (tuples)`
  - `T list`

Examples

```
> [1, 2, 3];
val it = [1, 2, 3]: int list

> ("ab", [1,2,3], 4);
val it = ("ab", [1, 2, 3], 4): string * int
list * int

> [[(1,2),(3,4)], [(5,6)], nil];
val it = [[(1, 2), (3, 4)], [(5, 6)], []]:
(int * int) list list
```

A list of a int*int list

What’s the type … without trying it ☺

How to participate?

Copy participation link

1. **Go to wooclap.com**
2. **Enter the event code in the top banner**

Event code

JLRDMB

Functions in ML

Functions

- In ML, just another type of value
- Represented by parametrized expressions
- Calculate a value based on parameters
  - No collateral effects
- Syntax `fn` (corresponds with λ in the λ-calculus, that we will see later)

```
fn <param> => <expression>;
```

- Example

```
fn n => n+1;
```

- We can directly apply the function to the parameter

```
(fn n => n+1) 5;
```

value 5 is associated to formal parameter n, and then the function is evaluated

Functions and names

- We can associate the functions to a name, just like values

```
> val increment = fn n => n+1;
val increment = fn: int -> int
```

- We also have a syntactic sugar notation for functions with names

```
> fun increment n = n+1;
val increment = fn: int -> int
```

- And then write

```
> increment 5;
val it = 6: int
```

Function types

- Example: function that converts character from lower to upper case

```
> fun upper(c) = chr (ord(c) - 32);
val upper = fn: char -> char
```

- Poly gives the type of the function. This is
  - The keyword `fn`
  - The type of the argument
  - The symbol `->`
  - The type of the result
- When using the function:

```
> upper (#"b");
val it = #"B": char
```

or

```
> upper #"b";
val it = #"B": char
```

The ML type system

- If `T1` and `T2` are types, so is

  `T1 -> T2`

- This is the type of functions that take an object of type `T1` and produce one of type `T2`
- Note that `T1` and `T2` can be any type, including function types

Specifying types

- ML deduces the types of functions automatically, as in our first example
- Another example

```
> fun square (x) = x * x;
val square = fn: int -> int
```

- But multiplication is also defined for reals. If we want to square real numbers, we can write

```
> fun square (x:real) = x * x;
val square = fn: real -> real
```

Examples

- Use of a function

```
> radius;
val it = 4.0: real
> pi;
val it = 3.14159: real
> pi * square (radius);
val it = 50.26544: real
```

- or

```
> pi * square radius;
val it = 50.26544: real
```

Multiple arguments

- All functions in ML have exactly one parameter but this parameter can be a tuple
- Example: Largest of three reals

```
> fun max3(a:real,b,c) = (* maximum of reals *)
if a>b then
if a>c then a else c
else
if b>c then b else c;
val max3 = fn: real * real * real -> real

> max3(5.0,4.0,7.0);
val it = 7.0: real
```

Note the syntax for comments

What would happen if we didn't specify that a was a real?

The input is a tuple of 3 int

```
>fun max3(a,b,c) =
  if a>b then
    if a>c then a else c
    else
      if b>c then b else c;
val max3 = fn: int * int * int -> int

> max3 (4,5,7);
val it = 7: int
```

Type inference in ML

Type inference in ML

- Types of operands and results of arithmetic expressions must agree, e.g., `(a+b)*2.0`
  - The right operand is a `real`
  - Therefore `a+b` must be a `real`
  - Therefore, so are `a` and `b`
- In a comparison (e.g., `a<=10`), both arguments have the same type, so `a` is an integer
- In a conditional, the types of the `then`, the `else` and the expression itself must all be the same

Type inference in ML

- If an expression used as an argument of a function is of a known type, the parameter must be of that type
- If the expression defining the result of a function is of a known type, the function returns that type
- If there is no way to determine the types of the arguments of an overloaded function (such as +), the type is the default (usually `integer`)
- If there is no way to determine the types of the arguments and operators are not used (so we do not have any type constraint), we can use the generic type ‘a
- If there is no way to determine the type of two arguments and there is no relation among them, we can use the generic types ‘a and ‘b

What can be inferred about the types of the arguments in the following function?

```
fun foo (a,b,c,d) =
   if a=b then c+1 else
      if a>b then c
      else b+d;
```

- In the second line we have the expression `c+1`
- Since 1 is an integer, `c` must also be an integer
- In the third line, the expressions following the `then` and `else` must be of the same type
- Since one of these is `c`, so the type of both is integer
- So `b+d` is of integer type
- Therefore, `b` and `d` must also be integers
- Since `a` and `b` are compared on lines 2 and 3, they must be of the same type
- Therefore, `a` is also an integer
- The function type is hence `val foo = fn: int * int * int * int -> int`

Summary

- Type errors and conversion in ML
- Names and denotable objects and variables
- Variables in ML
- Complex types in ML
- Functions in ML
- Type inference in ML

Readings

- Chapter 6 of the reference book
  - Maurizio Gabbrielli and Simone Martini "Linguaggi di Programmazione - Principi e Paradigmi", McGraw-Hill

Next time

- Visibility rules
- Abstraction of control
- Recursion