OCaml core

Syntax

EBNF grammar:

```
Exp ::= ID | NUM | Exp Exp | 'fun' Pat+ '->' Exp | UOP Exp |
    Exp BOP Exp | '(' Exp ')'
Pat ::= ID // simplified pattern
```

Functions and application

- examples of anonymous functions

```
fun x -> x+1 (* the increment function *)
fun x y -> x+y (* the addition function *)
```

- function application

```
(fun x -> x+1) 3 (* evaluation returns 4 *)
```

OCaml core

Syntax

EBNF grammar:

```
Exp ::= ID | NUM | Exp Exp | 'fun' Pat+ '->' Exp | UOP Exp |
    Exp BOP Exp | '(' Exp ')'
Pat ::= ID // simplified pattern
```

More on application

`exp1 exp2`

- evaluation of `exp1` is expected to return a function *f*
- evaluation of `exp2` is expected to return a valid argument *a*
- evaluation of `exp1 exp2` returns $f(a)$ (*f* applied to *a*)

OCaml core

Precedence and associativity rules

- standard rules for arithmetic expressions
- application is left-associative

```
(fun x y -> x+y) 3 4   (* is ((fun x y -> x+y) 3) 4 *)
```

- application has higher precedence then binary operators

```
(fun x->x*2) 1+2 (* is ((fun x->x*2) 1)+2 *)
1+(fun x->x*2) 2 (* is 1+((fun x->x*2) 2) *)
```

- anonymous functions have lower precedence then application and binary operators

```
fun x->x*2    (* is fun x->(x*2) *)
fun f a->f a  (* is fun f a->(f a) *)
```

- more critical cases: application and unary operators

```
f + 3 (* addition *)        f (+3) (* application *)
f - 3 (* subtraction *)     f (-3) (* application *)
+ f 3 (* is +(f 3) *)    - f 3     (* is -(f 3) *)
```

OCaml type inference

A simple interpreter session (Read Eval Print Loop)

Types can be inferred by the interpreter!

```
# 42
- : int = 42
# fun x->x*2
- : int -> int = <fun>
# (fun x->x+1) 2
- : int = 3
```

Syntax of OCaml core type expressions

BNF Grammar

```
Type ::= 'int' | Type '->' Type
```

OCaml core types

Terminology

- `int` is a *primitive type*: the type of integers
- `int -> int` is a *composite type*
- `->` is a type *constructor*: it is used for building composite types from simpler types
- types built with the `->` (arrow) constructor are called *arrow types* or *function types*

Meaning of arrow types

t_1 `->` t_2 is the type of functions from t_1 to t_2 that

- can only be applied to a single argument of type t_1
- always returns values of type t_2

OCaml core types

Remarks

- the arrow type constructor is right-associative

  `int->int->int = int->(int->int)`

- a type constructor always builds a type different from its type components

  $t_1 \texttt{->} t_2 \neq t_1$ and $t_1 \texttt{->} t_2 \neq t_2$

- two arrow types are equal if they are built with the same type components

  $t_1 \texttt{->} t_2 = t_3 \texttt{->} t_4$ if and only if $t_1 = t_3$ and $t_2 = t_4$

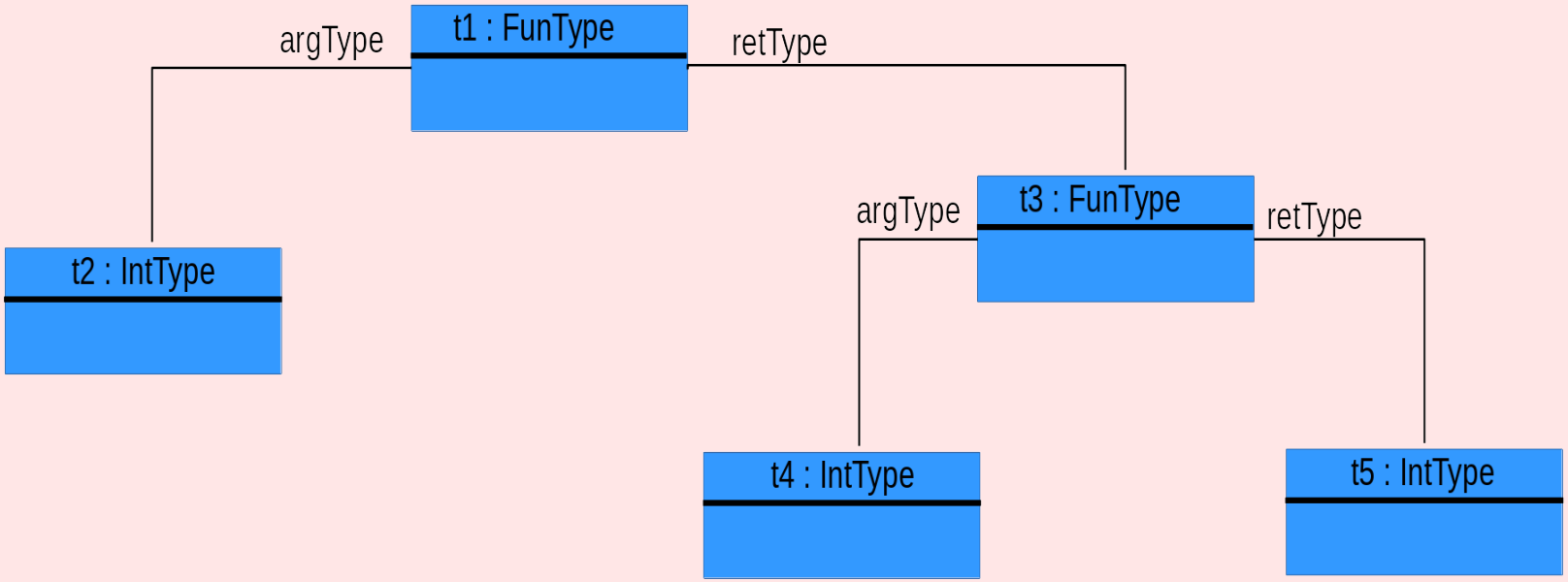
- example: from the items above we deduce

  `int->int->int` $\neq$ `(int->int)->int`

Types and type expressions

Remarks

- `int->int->int` is a type expressions, but is also called a type, because it represents a specific type
- `int->int->int` and `int->(int->int)` are *different* type expressions which represent the *same* type
- the type represented by a type expression *e* corresponds to the Abstract Syntax Tree (AST) of *e*



Higher order functions in OCaml

A useful syntactic abbreviation

fun $pat_1\ pat_2\ \ldots\ pat_n$ -> exp

is an abbreviation for

fun pat_1 -> **fun** pat_2 -> ... **fun** pat_n -> exp

Examples

```
# fun x y->x+y
- : int -> int -> int = <fun>
# fun x->fun y->x+y
- : int -> int -> int = <fun>
# fun x y z->x*y*z
- : int -> int -> int -> int = <fun>
# fun x->fun y->fun z->x*y*z
- : int -> int -> int -> int = <fun>
```

Higher order functions in OCaml

Examples

- `(int->int)->int`: functions that take a function (from `int` to `int`) as argument;
- `int->int->int`: functions that return a function (from `int` to `int`) as result;
- `(int->int)->int->int`: functions that
  - take a function (from `int` to `int`) as argument;
  - return a function (from `int` to `int`) as result.

```
# fun f -> 1+f 0
- : (int -> int) -> int = <fun>
# fun x y -> x+y
- : int -> int -> int = <fun>
# fun f x -> 1+f (1+x)
- : (int -> int) -> int -> int = <fun>
```

Tuples in OCaml

Syntax

New productions for `Exp` and `Pat`

```
Exp ::= '(' ')' | Exp ',' Exp
Pat ::= '(' ')' | '(' Pat (',' Pat)* ')'
```

New production for `Type`

```
Type ::= 'unit' | Type '*' Type
```

Precedence and associativity rules

- the tuple operator `,` has lower precedence than the other operators
- the tuple operator `,` is neither left- nor right-associative
- the `*` constructor has higher precedence than the `->` constructor
- the `*` constructor is neither left- nor right-associative

Tuples in OCaml

Examples

```
# ()
- : unit = ()
# 1,2,3
- : int * int * int = (1, 2, 3)
# (1,2),3
- : (int * int) * int = ((1, 2), 3)
# 1,(2,3)
- : int * (int * int) = (1, (2, 3))
# fun()->3
- : unit -> int = <fun>
# fun (x,y,z)->x*y*z
- : int * int * int -> int = <fun>
# fun ((x,y),z)->x*y*z
- : (int * int) * int -> int = <fun>
# fun (x,(y,z))->x*y*z
- : int * (int * int) -> int = <fun>
```