The UNIVERSITY of VERMONT

Benjamin Fenster
CS-103 (Skalka)
21 September 2003

Notes – Types (2)

I. Scope
  a. NB: OCaml uses static scope as do almost all modern PLs.
  b. Example
    i. `let x = 1;;`
    ii. `let addx = (fun (y : int) -> x + y);;`
    iii. `let x = 2;;`
    iv. `addx(1)` $\Downarrow$ ____
  c. Static
    i. x will *always* refer to the x that was in scope at the time of the function declaration
    ii. `addx(1)` $\Downarrow$ `2`
  d. Dynamic
    i. Uses whatever x is in scope at the time of the function call
    ii. `addx(1)` $\Downarrow$ `3`
  e. Static is more practical and more theoretically appealing.

II. Type Inference
  a. let x : τ = e_1 in e_2
  b. We've been assuming x has type τ when type-checking e_2
  c. (fun (x : τ) -> e)
  d. OCaml has a much smarter type analysis called "type inference" or "type reconstruction"
  e. We could write that same statement as (fun x-> e)
  f. Example
    i. (fun x -> x + 1);;
    ii. OCaml type inference analysis infers that x must be an int.
    iii. The function's type is int -> int
  g. Another Example: let x = 1 + 2 in 5 * x
  h. This is a really deep topic to be discussed in detail in the second half of the semester.
  i. This is an order of magnitude greater in complexity than regular type checking.
  j. In fact, it results in an algorithm of exponential complexity.
  k. Why isn't this a problem?
    i. The examples that have such ridiculous complexity are abhorrent (pathological examples)
    ii. In practice, time analysis will be polynomial (for real life programs)
    iii. So exponential complexity is not a deal-breaker.
  l. It's proven that if there is a type for an expression the analysis will find it.

III. Syntactic Sugar / Sugarings
  a. Nice to have, but don't really add anything new to the language.
  b. Define these in terms of other expression types.
  c. `let f = (fun x -> e)    let f x = e`
  d. `let rec f = (fun x _. e)    let rec f x = e`
  e. Example
    i. `let rec fact = (fun x -> if x = 0 then 1 else x * fact(x – 1))`
    ii. `let rec fact x = if x = 0 then 1 else x * fact(x – 1)`
  f. Certainly more convenient to write stuff the shorter way.
  g. Also don't have to define any new behavior. You can have a very limited language that's still easy to use by adding syntactic sugar.

IV. Commenting Conventions
  a. (* Comemnt *)
  b. These go right before definitions
  c. NB: These conventions will be enforced on homework assignment.

- d. For Our class:
  - i. (* <functionName> : τ
    In: <formal parameters, expected invariants>
    Out: <description of semantics>
    *)
  - ii. (* fact : int -> int
    In: $x \geq 0$
    Out: x!
    *)
    let rec fact…

V. Composite Types

- a. Data structures built of the composition of basic types
- b. Products / Tuples
  - i. Type Form
    - 1. τ_1 * … * τ_n for n > 1
    - 2. * like Cartesian product
  - ii. Value Form: (v_1, …, v_n) for n > 1
  - iii. Values
    - 1. Infinite set
    - 2. Examples
      - a. (1, 2) : int * int *[homogeneous]*
      - b. (“hi”, 2.0) : string * float *[heterogeneous]*
      - c. ((fun x -> x + 1), 0) : (int -> int) * int
      - d. (1, 0, “a”) : int * int * string
      - e. ((1,0), 1.0) : (int * int) * float
    - 3. NB: $(\tau_1 * \tau_2) * \tau_3 \neq \tau_1 * \tau_2 * \tau_3$
  - iv. Binding Strengths
    - 1. So far we have `*` and `->` as type constructors
    - 2. * binds more strongly than arrow
    - 3. `int * int -> int = (int * int) -> int`
  - v. Operations
    - 1. Construction
      - a. Formation
      - b. $(e_1, \ldots, e_n) : \tau_1 * \ldots * \tau_n$ iff $e_i : \tau_i$ for all $1 \leq i \leq n$
    - 2. Deconstruction
      - a. Projection
      - b. $\texttt{fst(e)} : \tau_1$, $\texttt{snd(e)} : \tau_2$ iff $e : \tau_1 * \tau_2$
      - c. Note that these are valid for pairs only!
  - vi. Evaluation
    - 1. Construction: $(e_1, \ldots, e_n) \Downarrow (v_1, \ldots, v_n)$ iff for all $1 \leq i \leq n$ $e_i \Downarrow v_i$ in right-to-left order
    - 2. Deconstruction: $\texttt{fst(e)} \Downarrow v_1$, $\texttt{snd(e)}$ v_2 iff $e : \tau_1 * \tau_2$
  - vii. Example
    - 1. `(1 + 2, sqrt(4)) : int * float` ⇓ `(3, 2.0)`
    - 2. `fst(1 + 2, sqrt(4)) : int` ⇓ `3`
  - viii. Example
    - 1. We can now, *in effect* create functions with multiple arguments.
    - 2. `let add pair x = fst(x) + snd(x);;`
    - 3. `addpair(3, 5)` ⇓ `8`
- c. More composite types later.

ERROR: undefinedfilename
OFFENDING COMMAND: </FONT>

STACK: