Benjamin Fenster
CS-103 (Skalka)
21 September 2003

Notes – Functions

I. Function Types
   a. In procedural programming, there's a distinction between functions and data.
   b. The basic view in functional programming is that functions *are* data.
   c. Functions can be passed as arguments from functions or returned as results.
   d. Form of Function Types
      i. τ_1 -> τ_2
      ii. τ_1 is the "domain type"
      iii. τ_2 is the "range type"
      iv. abs : int -> int
      v. sqrt : float -> float
   e. Values (1)
      i. Primitives (functional constants)
      ii. e.g. abs, sqrt
      iii. Provided with the language implementation
      iv. Types are pre-assigned
      v. We will let c range over primitives.
   f. Operations
      i. Application – using the function
         1. $e_1e_2 : \tau$ iff $e_1 : \tau' \text{ -> } \tau$ and $e_2 : \tau$
         2. For primitives, evaluation of application is pre-defined
      ii. No other operations necessary
   g. Values (2)
      i. "lambda" functions or functional abstractions
      ii. Form: fun x : τ -> e
      iii. Take note that these are anonymous functions.
      iv. Type Checking
         1. When we assert that the argument has a certain type, the type checker will assert that we've used it that way.
         2. Extend the type environment with x : τ
         3. Type-check e in the extended environment
         4. Yield e : τ' if successful
         5. Retract e : τ from environment, yield that the entire function has type τ -> τ'
      v. Definition: The scope of x in `(fun x : τ -> e)` is e
   h. Evaluation
      i. Since functions are treated as values and values evaluate as themselves, any function `(fun x : τ -> e)` $\Downarrow$ `(fun x : τ -> e)`
      ii. This is rather counterintuitive. Why? Consider an example:
      iii. `(fun x : int -> 1 + 2)` $\Downarrow$ `(fun x : int -> 1 + 2)` *not 3!*
      iv. Lambda abstraction *freezes* computation.
      v. The function is "not ready" for a value yet. It's still abstract since it still needs a value.
   i. Application Evaluation
      i. $e_1e_2 \Downarrow v$ iff $e_1 \Downarrow$ `(fun x : τ -> e)` and $e_2 \Downarrow v'$ and $e \Downarrow v$ after temporarily extending the environment with x = v'
      ii. After evaluation of e_1e_2, remove x = v from the environment.
   j. Definition: In e_1e_2 we call e_2 the argument of e_1
   k. Examples of Scope
      i. `let x : int = 5 in (fun y : int -> x + y)3` $\Downarrow$ `8`
      ii. `(fun x : int -> let y = 5 in x + y)3` $\Downarrow$ `8`
      iii. `let x : int = 1 in (fun x : int -> 2 * x)` $\Downarrow$ `0`
   l. Example

i. `(fun (f : int -> int) -> f0)`
ii. Takes another function, applies it to 0.
iii. `(fun (f : int -> int) -> f0)(fun x : int -> x + 1) : int` $\Downarrow$ `1`
iv. Type `(int -> int) -> int`
v. `(fun (f : int -> int) -> f)(fun (x : int) -> x + 1)3` $\Downarrow$ `4`

II. Function Declarations
  a. Normal
    i. `let double : int -> int = (fun x : int -> 2 * x);;`
    ii. `double (x)` $\Downarrow$ `4`
    iii. We're just binding a variable to the function "value" since functions are data!
    iv. Remember that the function does not have access to its own name (see notes about variable declaration) so we can't do recursion yet.
  b. Recursive Declarations
    i. Introduction
      1. A function that calls itself.
      2. Without the rec keyword, the scope of let declarations does not extend to the statement itself
      3. Base Case (basis): Case in which the argument causes immediate termination (no recursive call)
      4. Recursive Case: A case in which the argument causes a recursive call.
      5. Well-foundedness: A well-defined recursive function must have a basis.
      6. The real power in a function language is in recursion (as opposed to a procedural language, where the power comes from iteration).
    ii. Form: let rec x : τ = e
    iii. NB: Scope of x extends to e
    iv. `let rec expt : int -> int = (fun (x : int) -> if x = 0 then 1 else 2 * expt(x – 1))`
    v. `let rec fact : int -> int = (fun (x : int) -> if x = 0 then 1 else x * fact(x – 1))`
    vi. Without the rec keyword, the scope of let declarations does not extend to the statement following them.

III. Function Application
  a. Call by Value
    i. $e_1e_2 \Downarrow v$ iff $e_1 \Downarrow$ (fun (x : τ) -> e) and $e_2 \Downarrow v'$ and $e \Downarrow v$ in an environment extended with x = v'
    ii. Call by Value because e_2 is evaluated first.
    iii. Nothing requires this – we could just use e_2 itself in place of x
    iv. Called "eager strategy", "strict strategy"
    v. This is what OCaml uses.
  b. Call by Name
    i. $e_1e_2 \Downarrow v$ iff $e_1 \Downarrow$ `(fun (x : τ) -> e)` and $e \Downarrow v$ in an environment extended with x = e_2
    ii. Called "lazy strategy"
    iii. Note that e_2 now needs to be evaluated everywhere x is used. If e_2 is complicated, that could cause an efficiency problem.
  c. Call by Need
    i. Essentially the same as "call by name" except that e_2 is evaluated the first time x is encountered, then the result is saved to be used for all future occurrences.
    ii. This eliminates the (potentially huge) efficiency drop in call by name.
  d. Example

         i. `e   (fun (x : int) -> 1) e’`
         ii.   means “is defined equal to”
         iii. Assume e’ does not terminate
         iv. Eage: Won’t terminate since it will begin by trying to evaluate e’. Diverges.
         v. e ⇑ (diverges)
         vi. Lazy Strategy: e ⇓ 1

IV. Recursive Functions
   a. One of our main course topics is reasoning about programs.
   b. Functional programming is great for this, and functional programming gets its strength from recursion.
   c. Overview
      i. Base Case: No recursive calls
      ii. Recursive Calls: Results in recursive call
   d. Factorial Example
      i. `let rec fact = (fun (x : int) -> if x = 0 then 1 else x * fact(x – 1))`
      ii. Base Case: 0
      iii. Recursive Case: n > 0
   e. Definition: A function is <u>total</u> with regard to (wrt) a domain S iff f(x) ⇓ v for all x ∈ S.
   f. fact is total wrt {0, 1, 2, …} or ℕ
   g. fact is not total wrt ℤ

V. Mathematical Induction
   a. Really closely related to recursion. In some sense, they’re the same thing.
   b. Suppose we want to prove that pa property P holds for ℕ
   c. Use proof by Mathematical Induction
      i. Prove base case: Prove that P holds for zero (prove that P(0) holds)
      ii. Make an induction <u>hypothesis</u>: P(j) holds for all 0 < j < n
      iii. Prove the induction step: Given the induction hypothesis (IH), prove P(n)
   d. Factorial Example
      i. let rec fact = (fun (x : int) -> if x = 0 then 1 else x * fact(x – 1))
      ii. Proposition: That for all n ∈ ℕ fact(n) = “n!”.
      iii. Proof (by Mathematical Induction)
         1. Base Case
            a. Prove that fact(0) ⇓ “0!” ⇓ 1
            b. This is obvious by the function’s definition.
         2. Induction Hypothesis: fact(j) ⇓ “j!” for 0 ≤ j < n
         3. Prove fact(n) ⇓ “n!”
            a. By definition of fact, fact(n) = n * fact(n – 1)
            b. fact(n – 1) ⇓ “(n – 1)!” by the IH
            c. So fact(n) = n * “(n – 1)!” by the definition of ⇓.
            d. Hence, fact(n) ⇓ “n!”
   e. Example
      i. let rec expt = (fun (x : int) -> if x = 0 then 1 else 2 * expt(x – 1))
      ii. Proposition: For all n ∈ ℕ expt(n) ⇓ “2^{n}”
      iii. Base Case
         1. Prove expt(0) ⇓ 1
         2. This is obvious by the definition of the function
         3. IH: Assume expt(j) ⇓ “2^{j}” for 0 ≤ j < n
      iv. Induction Step: Prove expt(n) ⇓ “2^{n}”
      v. By definition, expt(n) = 2 * expt(n – 1)
      vi. By IH, expt(n – 1) ⇓ “2^{n-1}”, hence exp(n) = 2 * “2^{n-1}”
      vii. By definition of ⇓ we have 2 * “2^{n-1}” = “$2 * 2^{n-1}$” = “2^{n}”
   f. The moral: When programming recursively, think inductively.
      i. Don’t try to visualize the entire call tree.

ii. Imagine that all previous recursive calls have worked perfectly and decide what needs to be done in *this* call to make the result come out properly.

ERROR: undefinedfilename
OFFENDING COMMAND: </FONT>

STACK: