CSE 341: Programming Languages

Spring 2007

Lecture 2 — ML Functions, Pairs and Lists

Goals for today

- More ML essentials
- Discuss some "first-week" gotchas
  - We will learn more and better constructs soon

Note: These slides (and most slides all quarter) will make much more sense in conjunction with the corresponding code file (`lec02.sml`).

Recall a program is a sequence of bindings...

Function Definitions

... A second kind of binding is for functions

Syntax: `fun x0 (x1 : t1, ..., xn : tn) = e`

Typing rules:

1. Context for `e` is (the function's context extended with) `x1:t1, ..., xn:tn` *and*:
2. `x0 : (t1 * ... * tn) -> t` *where*:
3. `e` has type `t` in this context

(This "definition" is circular because functions can call themselves and the type-checker "guessed" `t`.)

(It turns out in ML there is always a "best guess" and the type-checker can always "make that guess". For now, it's magic.)

Evaluation: *A FUNCTION IS A VALUE.*

Function Applications (a.k.a. Calls)

Syntax: `e0 (e1,...,en)`

Typing rules (all in the application's context):

1. `e0` must have some type `(t1 * ... * tn) -> t`
2. `ei` must have type `ti` (for i = 1, ..., n)
3. `e0 (e1,...,en)` has type `t`

Evaluation rules:

1. `e0` evaluates to a function f in the applicaton's environment
2. `ei` evaluates to value `vi` in the application's environment
3. result is f's body evaluated in an environment extended to bind `xi` to `vi` (for i = 1, ..., n).

("an environment" is actually the environment where f was defined)

Some Gotchas

- The * between argument types (and pair-type components) has nothing to do with the * for multiplication
- In practice, you almost never have to write argument types
  - But occasionally needed; maybe for homework 1
  - Sometimes improves error messages and clarity of code
  - But *type inference* is a very cool thing in ML
  - Types unneeded for other variables or function return-types
- Context and environment for a function body includes:
  - Previous bindings
  - Function arguments
  - The function itself
  - But *not* later bindings

Recursion

- A function can be defined in terms of itself.
- Of course, the recursive calls must solve "smaller" or "simpler" problems.
- This is more powerful than loops and often more convenient.
- Many, many examples to come in 341.

Pairs

Our first way to build *compound data* out of simpler data:

- Syntax to build a pair: `(e1,e2)`
- If `e1` has type `t1` and `e2` has type `t2` (in current context), then `(e1,e2)` has type `t1*t2`.
  - (It might be better if it were `(t1,t2)`, but it isn't.)
- If `e1` evaluates to `v1` and `e2` evaluates to `v2` (in current environment), then `(e1,e2)` evaluates to `(v1,v2)`.
  - (Pairs of values are values.)
- Syntax to get part of a pair: `#1 e` or `#2 e`.
- Type rules for getting part of a pair: _____
- Evaluation rules for getting part of a pair: _____