Today

- A brief remark about equality types
- Using patterns
- Specifying what a function does
equality in ML

e_1 = e_2

• Only for expressions whose type is an equality type

• Equality types include all types built from

  int, bool, *, -list  but NOT real or ->

  e.g.

  int list
  int * bool
  (int * bool) list

- 1+1 = 2;
  val it = true : bool

- [1,1] = (0+1)::[2-1];
  val it = true : bool

- (fn x => x+x) = (fn y => 2*y);
  Error: operator and operand don't agree [equality type required]
patterns

• We introduced *patterns*, to be used for *matching* with *values*

• Matching \( p \) to value \( v \) either *fails*, or *succeeds* and binds names to values

\[
p ::= _ | x | n | \text{true} | \text{false} \\
| (p_1, \ldots, p_n) \\
| p_1::p_2 | [p_1, \ldots, p_n]
\]

Can attach types if desired
Using patterns

Recall… \( \text{divmod} : \text{int} \times \text{int} \to \text{int} \times \text{int} \)

```ocaml
fun check (x:int, y:int) :bool =
  let
    val (q, r) = divmod (x, y)
  in
    (x = q*y + r)
  end
```

Introduces \( \text{check} : \text{int} \times \text{int} \to \text{bool} \)

Binds \( \text{check} \) to a function value

What does this function do?
This definition uses list patterns

- `[ ]` matches (only) the empty list
- `d::L` matches a non-empty list, binds `d` to head of the list, `L` to its tail

What does this function do?
fun decimal n = if n < 10 then [n]
  else (n mod 10) :: decimal (n div 10)

Why didn’t I define this function using integer patterns?

- decimal 42 = [2,4]
- decimal 0 = [0]

What does this function do?
log : int -> int

fun log x = 
  if x = 1 then 0 else 1 + log (x div 2)

log 3 = ???

• Q: How can we describe this function?

• A: Specify its applicative behavior…
  - For what argument values does it terminate?
  - How does the output relate to the input?
Specifications

For each function definition we specify:

- **Type**
  (of the function’s `argument` and `result`)

- **Assumption**
  (about `argument` value)

- **Guarantee**
  (about `result` value, when assumption holds)
fun log (x:int) : int = 
  if x=1 then 0 else 1 + log (x div 2)

(* TYPE log : int -> int *)
(* REQUIRES ... x ... *)
(* ENSURES ... log x .... *)

For all values x : int satisfying the assumption, log x : int and its value satisfies the guarantee
fun log (x:int) : int = 
   if x=1 then 0 else 1 + log (x div 2)

(* TYPE    log : int -> int *)
(* REQUIRES x > 0 *)
(* ENSURES   log x = the integer k ≥ 0 *)
(* such that 2^k ≤ x < 2^{k+1} *)

For all integers x>0, log x evaluates to an integer k such that 2^k ≤ x < 2^{k+1}
notes

• Can use \( \rightarrow^\ast \) or \( = \) in specs

• Use *math notation* and *math facts*, *accurately*!

• A function can have several specs…

  different *assumptions* may lead to different *guarantee*
fun log (x:int) : int =
  if x=1 then 0 else 1 + log (x div 2)

(* log : int -> int *)

(* REQUIRES x = a power of 2 *)

(* ENSURES log x = the integer k *)

(* such that \(2^k = x\) *)

(a weaker spec ... why?)

(actually implied by original spec)
**eval spec**

```plaintext
fun eval ([ ] : int list) : int = 0
| eval (d::L) = d + 10 * (eval L)
```

TYPE eval : int list -> int

REQUIRES R = a list of decimal digits

ENSURES eval R = a non-negative integer

(not the best spec for eval… why not?)

(doesn’t say which non-negative integer!)
fun decimal (n:int) : int list =
   if n<10 then [n]
   else (n mod 10) :: decimal (n div 10)

TYPE      decimal : int -> int list
REQUIRES  n ≥ 0
ENSURES   decimal n = a list of decimal digits

(again, not the best spec…)
connection

- **eval** and **decimal** are designed to fit together
- They satisfy a **connection spec**

<table>
<thead>
<tr>
<th>TYPE</th>
<th>decimal : int -&gt; int list</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>eval : int list -&gt; int</td>
</tr>
<tr>
<td><strong>REQUIRES</strong></td>
<td>n ≥ 0</td>
</tr>
<tr>
<td><strong>ENSURES</strong></td>
<td>eval(decimal n) = n</td>
</tr>
</tbody>
</table>

(says “which list” and “which integer”)
Evaluation

• Expression evaluation produces a value if it terminates
  • \( e \Rightarrow^k e' \) \( e \) evaluates to \( e' \) in \( k \) steps
  • \( e \Rightarrow^* v \) \( e \) evaluates to \( v \) in finitely many steps

• Declarations produce value bindings
  • \( d \Rightarrow^* [x_1:v_1, \ldots, x_k:v_k] \)

• Matching a pattern to a value either succeeds with bindings, or fails

TYPE SAFETY
Basic properties

\[ e \Rightarrow^* e' \text{ if and only if } \exists k \geq 0. \, e \Rightarrow^k e' \]

\[ e \Rightarrow^0 e \]

If \( e_1 \Rightarrow^m e_2 \) and \( e_2 \Rightarrow^n e_3 \)
then \( e_1 \Rightarrow^{m+n} e_3 \)
Substitution

For bindings \( \llbracket x_1 : v_1, ..., x_k : v_k \rrbracket \) and expression \( e \) we write

\[
\llbracket x_1 : v_1, ..., x_k : v_k \rrbracket e
\]

for the expression obtained by substituting

\( v_1 \) for \( x_1, ..., v_k \) for \( x_k \)

in \( e \)

(for free occurrences, only)

\[
\llbracket x : 2 \rrbracket (x + x) \quad \text{is} \quad 2 + 2
\]

\[
\llbracket x : 2 \rrbracket (\text{fn } y \Rightarrow x + y) \quad \text{is} \quad \text{fn } y \Rightarrow 2 + y
\]

\[
\llbracket x : 2 \rrbracket (\text{fn } x \Rightarrow x + x) \quad \text{is} \quad \text{fn } x \Rightarrow x + x
\]
Explaining evaluation

• For each syntactic construct we give evaluation rules for $\Rightarrow$
  • showing order-of-evaluation

• We derive evaluation laws for $\Rightarrow^*$
  • how expressions evaluate
  • what is the value, if it terminates

• We can also count number of steps $\Rightarrow^{(n)}$
Addition rules

\[
\begin{align*}
e_1 & \Rightarrow e_1' \\
\hline
\frac{e_1 + e_2}{\Rightarrow e_1' + e_2}
\end{align*}
\]

\[
\begin{align*}
e_2 & \Rightarrow e_2' \\
\hline
\frac{v_1 + e_2}{\Rightarrow v_1 + e_2'}
\end{align*}
\]

\[
\frac{v_1 + v_2}{\Rightarrow v \quad \text{where} \quad v = v_1 + v_2}
\]

+ evaluates from left-to-right
**Addition law**

If
\[
e_1 \Rightarrow^* v_1 \text{ and } e_2 \Rightarrow^* v_2 \text{ and } v = v_1 + v_2
\]
then
\[
e_1 + e_2 \Rightarrow^* v
\]

\[
(2+2) + (3+3) \\
\Rightarrow 4 + (3+3) \\
\Rightarrow 4 + 6 \\
\Rightarrow 10
\]

\[
(2+2) + (3+3) \Rightarrow^{(3)} 10
\]
Application rules

\[
\begin{align*}
& \quad e_1 \Rightarrow e_1' \\
& \quad e_1 e_2 \Rightarrow e_1' e_2 \\
& \quad e_2 \Rightarrow e_2' \\
& \quad (\text{fn } x \Rightarrow e) \ e_2 \Rightarrow (\text{fn } x \Rightarrow e) \ e_2' \\
& \quad (\text{fn } x \Rightarrow e) \ v \Rightarrow [\ x:v \ ] e
\end{align*}
\]

*a function call evaluates its argument*
Application law

If
\[ e_1 \Rightarrow^* (\text{fn } x \Rightarrow e) \quad \text{and} \quad e_2 \Rightarrow^* v \]
then
\[ e_1 \ e_2 \Rightarrow^* [x:v]e \]
Other rules

• div and mod evaluate from left to right
• Tuples evaluate from left to right
• Lists evaluate from left to right
Declaration rule

In the scope of \texttt{fun} \( f(x) = e \),

\[
\frac{}{f \Rightarrow (\texttt{fn} \ x \Rightarrow e)}
\]

\texttt{fun} \ \texttt{divmod}(x, y) = (x \div y, x \mod y)

\texttt{divmod} (3,2)

\[
\Rightarrow (\texttt{fn}(x, y) \Rightarrow (x \div y, x \mod y)) (3,2)
\]

\[
\Rightarrow (3 \div 2, 3 \mod 2)
\]

\[
\Rightarrow (1, 3 \mod 2)
\]

\[
\Rightarrow (1, 1)
\]
**Example**

fun \( f(x) = \begin{cases} 1 & \text{if } x = 0 \\ f(x-1) & \text{else} \end{cases} \)

\((*) \quad f : \text{int} \rightarrow \text{int} \quad (*)\)

\((*) \text{ REQUIRES } x \geq 0 \quad (*)\)

\((*) \text{ ENSURES } f(x) = 1 \quad (*)\)
Example

In the scope of

```plaintext
fun f(x) = if x=0 then 1 else f(x-1)
```

\[ f(1-1) \]

\[
=> (fn x => if x=0 then 1 else f(x-1)) (1-1)
\]

\[
=> (fn x => if x=0 then 1 else f(x-1)) 0
\]

\[
=> if 0=0 then 1 else f(0-1)
\]

\[
=> if true then 1 else f(0-1)
\]

\[
=> 1
\]

(\textit{justified by the rules given earlier!})
Patterns

• If matching $p_1$ to $v$ succeeds with $[B]$, 
  $$(\text{fn } p_1 => e_1 | p_2 => e_2) \ v = \Rightarrow^* \ [B] \ e_1$$

• If matching $p_1$ to $v$ fails, 
  and matching $p_2$ to $v$ succeeds with $[B]$, 
  $$(\text{fn } p_1 => e_1 | p_2 => e_2) \ v = \Rightarrow^* \ [B] \ e_2$$

• If matching $p_1$ to $v$ fails, 
  and matching $p_2$ to $v$ fails, 
  $$(\text{fn } p_1 => e_1 | p_2 => e_2) \ v \text{ fails}$$ 

  uncaught exception Match [nonexhaustive match failure]
So far

• Using $\Rightarrow$ and $\Rightarrow^*$ we can talk precisely about program behavior

• But we may want to ignore evaluation order...

For all expressions $e_1, e_2 : \text{int}$ and all values $v : \text{int}$,
if $e_1 + e_2 \Rightarrow^* v$ then $e_2 + e_1 \Rightarrow^* v$

In such cases, equational specs may be better

For all expressions $e_1, e_2 : \text{int}$,
$e_1 + e_2 = e_2 + e_1$

the same, more succinctly
Example

\[
\text{fun } \text{addl}(x, y) = x + y \quad \text{fun } \text{addr}(x, y) = y + x
\]

\text{addl} \text{ and } \text{addr} \text{ are indistinguishable}

- Let \( E \) be a well-typed expression of type \( \text{int} \)
- Let \( E' \) be obtained from \( E \) by replacing a call to \( \text{addl} \) with a call to \( \text{addr} \)
- \( E' \) also has type \( \text{int} \)
- \( E \) and \( E' \) have equal values

Not easy to prove \text{directly} using \( =>* \)
Equivalence

- For each type $t$ there is a *mathematical* notion of equivalence (or equality) $=t$ for *values* of type $t$

- *Expressions* of type $t$ are equivalent iff they evaluate to *equivalent* values, or both diverge

\[ \forall v_1,v_2: \text{int.} \ (v_1 =_{\text{int}} v_2 \implies f_1 \ v_1 =_{\text{int}} f_2 \ v_2) \]

\[ v_1 =_{\text{int}} v_2 \iff v_1 = v_2 \ (\text{equal integers}) \]

\[ f_1 =_{\text{int->int}} f_2 \iff \]

\[ f_1(\text{int}) (v_1) = v_1 \]
Extensionality

• When $e_1$ and $e_2$ are values of type $t \rightarrow t'$

\[ e_1 = e_2 \]

if and only if

for all values $v_1$, $v_2$ of type $t$

\[ v_1 = v_2 \implies e_1 \ v_1 = e_2 \ v_2 \]
Equations

(when well-typed)

- Arithmetic

\[ e + 0 = e \]
\[ e_1 + e_2 = e_2 + e_1 \]
\[ e_1 + (e_2 + e_3) = (e_1 + e_2) + e_3 \]
\[ 21 + 21 = 42 \]

- Boolean

\[
\text{if true then } e_1 \text{ else } e_2 = e_1 \\
\text{if false then } e_1 \text{ else } e_2 = e_2 \\
(0 < 1) = \text{true}
\]
Equations
(when well-typed)

- Applications

\[(fn \ x \Rightarrow \ e) \ v = [x:v]e\]

- Declarations

In the scope of

\[fun \ f(x) = e\]

the equation

\[f = (fn \ x \Rightarrow \ e)\]

holds
Compositionality
(when well-typed)

• Substitution of equals
  • If $e_1 = e_2$ and $e_1' = e_2'$
    then $(e_1 e_1') = (e_2 e_2')$
  • If $e_1 = e_2$ and $e_1' = e_2'$
    then $(e_1 + e_1') = (e_2 + e_2')$

and so on
Equivalence

\[
\text{fun addl}(x, y) = x + y \\
\text{fun addr}(x, y) = y + x
\]

• Let \( E \) be a well-typed expression of type \( \text{int} \) containing a call to \( \text{addl} \)

• Let \( E' \) be obtained by changing to \( \text{addr} \)

• Easy to show that \( \text{addl} = \text{int} \times \text{int} \rightarrow \text{int} \) \( \text{addr} \)

• By compositionality, \( E = \text{int} E' \)

• Hence, if \( E \rightarrow^* 42 \) then also \( E' \rightarrow^* 42 \)

Easy to prove using =
Equations
(when well-typed)

- Applications

\[
(fn \ p_1 \Rightarrow e_1 \mid p_2 \Rightarrow e_2) \ v \ = \ [B_1]e
\]
if \ matching \ p_1 \ to \ v \ succeeds \ with \ bindings \ [B_1]

\[
(fn \ p_1 \Rightarrow e_1 \mid p_2 \Rightarrow e_2) \ v \ = \ [B_2]e
\]
if \ matching \ p_1 \ to \ v \ fails
& matching \ p_2 \ to \ v \ succeeds \ with \ bindings \ [B_2]
Equations

• Declarations

In the scope of

\[
\text{fun } f(p_1) = e_1 \mid f(p_2) = e_2
\]

the equation

\[
f = (\text{fn } p_1 \Rightarrow e_1 \mid p_2 \Rightarrow e_2)
\]

holds
Useful facts

• $e \Rightarrow^* v$ implies $e = v$

• $e \Rightarrow^* v$ implies $(\text{fn } x \Rightarrow E) e = [x:v] E$

evaluation
is consistent with
equivalence
So far

- Can use equivalence or $=\equiv$ to specify the *applicative behavior* of functional programs
- Equality is *compositional*
- Equality is *defined* in terms of evaluation
- $=>^*$ is *consistent* with $=\equiv$ and ML evaluation
Guidelines

- Be clear and precise
- Use bound variable names consistently
- Use $=>^\ast$ (evaluation) and $=$ (equality) accurately
- Don’t leave assumptions hidden