Lecture 3
September 4
Stephen Brookes
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 `->

- (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

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

Can attach types if desired
Using patterns

Recall... \( \text{divmod} : \text{int} \times \text{int} \rightarrow \text{int} \times \text{int} \)

\[
\text{fun check (x:int, y:int):bool =}
\begin{align*}
\text{let} & \\
\text{val (q, r) = divmod (x, y)}
\end{align*}
\begin{align*}
\text{in} & \\
(x = q* y + r)
\end{align*}
\text{end}
\]

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

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

\textbf{What does this function do?}
**eval**: int list -> int

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

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

`eval [2,4] =>* 42`

**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 $\Rightarrow^*$ or $=$ in specs

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

• A function can have several specs…

different *assumptions* may lead to different *guarantee*
another log spec

**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**

\[
\textbf{fun} \quad \text{eval} \quad ([\quad] : \text{int list}) : \text{int} = 0 \\
| \quad \text{eval} \quad (d::L) \quad = d + 10 \times (\text{eval} \quad L)
\]

\textbf{TYPE} \quad \text{eval} : \text{int list} \rightarrow \text{int}

\textbf{REQUIRES} \quad R = \text{a list of decimal digits}

\textbf{ENSURES} \quad \text{eval} \quad R = \text{a non-negative integer}

\textbf{(not the best spec for eval... why not?)}

\textbf{(doesn’t say which non-negative integer!)}
**decimal spec**

```ml
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>eval</td>
<td>: int list -&gt; int</td>
</tr>
</tbody>
</table>

REQUIRES  n ≥ 0

ENSURES eval(decimal n) = n

(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
  - $e \Rightarrow^\ast v$: $e$ evaluates to $v$ in finitely many steps

- Declarations produce value bindings
  - $d \Rightarrow^\ast [x_1:v_1, \ldots, x_k:v_k]$

- Matching a pattern to a value either succeeds with bindings, or fails
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 \([ x_1:v_1, ..., x_k:v_k ]\) and expression \(e\) we write

\([ x_1:v_1, ..., x_k:v_k ] e\)

for the expression obtained by substituting

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

(for free occurrences, only)

\([ x:2 ] (x + x)\) is \(2 + 2\)

\([ x:2 ] (\text{fn } y => x + y)\) is \(\text{fn } y => 2 + y\)

\([ x:2 ] (\text{fn } x => x + x)\) is \(\text{fn } x => x + x\)
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

\[
\begin{align*}
  e_1 &= e_1' \\
  e_1 + e_2 &= e_1' + e_2 \\
  e_2 &= e_2' \\
  v_1 + e_2 &= v_1 + e_2' \\
  v_1 + v_2 &= v
\end{align*}
\]

\( e_i, v_i : \text{int} \)

+ 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
\[ e_1 + e_2 \Rightarrow^* v \]

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

\[(2+2) + (3+3) \Rightarrow^* 10 \]

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

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

*a function call evaluates its argument*
Application law

If

\[ e_1 \Rightarrow^* (\text{fn } x \Rightarrow e) \text{ and } e_2 \Rightarrow^* v \]

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 \textbf{fun} \( f(x) = e \),

\[
f \Rightarrow (\textbf{fn} \; x \Rightarrow e)
\]

\textbf{fun} \( \text{divmod}(x, y) = (x \; \text{div} \; y, x \; \text{mod} \; y) \)

\[
\text{divmod} \; (3,2)
\Rightarrow (\text{fn}(x, y) \Rightarrow (x \; \text{div} \; y, x \; \text{mod} \; y)) \; (3,2)
\Rightarrow (3 \; \text{div} \; 2, 3 \; \text{mod} \; 2)
\Rightarrow (1, 3 \; \text{mod} \; 2)
\Rightarrow (1, 1)
\]
Example

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

(* f : int -> int *)

(* REQUIRES x >= 0 *)

(* ENSURES f x = 1 *)
Example

In the scope of

\[
\text{fun } f(x) = \text{if } x = 0 \text{ then } 1 \text{ else } f(x-1)
\]

\[
f(1-1)
\]

\[
\Rightarrow \ (\text{fn } x \Rightarrow \text{if } x = 0 \text{ then } 1 \text{ else } f(x-1)) \ (1-1)
\]

\[
\Rightarrow \ (\text{fn } x \Rightarrow \text{if } x = 0 \text{ then } 1 \text{ else } f(x-1)) \ 0
\]

\[
\Rightarrow \ \text{if } 0 = 0 \text{ then } 1 \text{ else } f(0-1)
\]

\[
\Rightarrow \ \text{if } \text{true then } 1 \text{ else } f(0-1)
\]

\[
\Rightarrow \ 1
\]

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

- If matching $p_1$ to $v$ succeeds with $\lbrack B \rbrack$, 
  $$(\text{fn } p_1 \Rightarrow e_1 \mid p_2 \Rightarrow e_2) \, v \Rightarrow^* \lbrack B \rbrack \, e_1$$

- If matching $p_1$ to $v$ fails, 
  and matching $p_2$ to $v$ succeeds with $\lbrack B \rbrack$, 
  $$(\text{fn } p_1 \Rightarrow e_1 \mid p_2 \Rightarrow e_2) \, v \Rightarrow^* \lbrack B \rbrack \, e_2$$

- If matching $p_1$ to $v$ fails, 
  and matching $p_2$ to $v$ fails, 
  $$(\text{fn } p_1 \Rightarrow e_1 \mid p_2 \Rightarrow e_2) \, v \text{ fails}$$

  uncaught exception Match [nonexhaustive match failure]
So far

- Using $\Rightarrow$ and $\Rightarrow^\ast$ 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 $e_1 + e_2 \Rightarrow^\ast v$ then $e_2 + e_1 \Rightarrow^\ast v$

In such cases, equational specs may be better

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

fun addl (x, y) = x + y       fun addr(x, y) = y + x

addl and addr are indistinguishable

• Let E be a well-typed expression of type int

• Let E′ be obtained from E by replacing a call to addl with a call to addr

• E′ also has type int

• E and E′ have equal values

Not easy to prove 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

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

  if true then \( e_1 \) else \( e_2 \) = \( e_1 \)

  if false then \( e_1 \) else \( e_2 \) = \( e_2 \)

  \((0 < 1) = \text{true}\)
Equations
(when well-typed)

• Applications

\[(\text{fn } x \Rightarrow e) \, v = [x:v]e\]

- only when the argument is a value

• Declarations

In the scope of

\[\text{fun } f(x) = e\]

the equation

\[f = (\text{fn } x \Rightarrow e)\]

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

fun add1 (x, y) = x + y
fun addr(x, y) = y + x

• Let $E$ be a well-typed expression of type int containing a call to add1

• Let $E'$ be obtained by changing to 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

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

\[(\text{fn } p_1 \Rightarrow e_1 \mid p_2 \Rightarrow e_2) \; v \; = \; [B_2]e\]
if matching \(p_1\) to \(v\) fails
& matching \(p_2\) to \(v\) succeeds with bindings \([B_2]\)
• Declarations

In the scope of

fun f(p₁) = e₁ | f(p₂) = e₂

the equation

f = (fn p₁ => e₁ | p₂ => e₂)

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

• Be clear and precise

• Use bound variable names consistently

• Use $\Rightarrow^*$ (evaluation) and $=$ (equality) accurately

• Don’t leave assumptions hidden