Patterns and specifications
I will teach you in a ROOM
I will teach you now on ZOOM
I will teach you in your House
I will teach you with a MOUSE
I will teach you here and there
Patterns and specifications
Advice

• After class, study slides and lecture notes.

• Start homework early, plan to finish on time.

• Don’t use piazza as a first resort, or close to a handin deadline.

• Ask for help only after you’ve studied, and tried.

• Think before you write.
Today

- A brief remark about equality types
- Patterns and how to use them
- Specifying program behavior
  - ✔ evaluation and equivalence
equality in ML

\[ e_1 = e_2 \]

- Only for expressions whose type is an equality type
- Equality types are built from `int`, `bool`, `- * -`, and `-list`
equality in ML

\[ e_1 = e_2 \]

- Only for expressions whose type is an \textit{equality} type
- Equality types are built from \texttt{int}, \texttt{bool}, \texttt{- * -}, and \texttt{-list}

\texttt{int list}

\texttt{e.g. int * bool (int * bool) list}
equality in ML

\[ e_1 = e_2 \]

- Only for expressions whose type is an \textit{equality} type
- Equality types are built from \texttt{int}, \texttt{bool}, \texttt{- * -}, and \texttt{-list}
equality in ML

\[ e_1 = e_2 \]

- Only for expressions whose type is an equality type
- Equality types are built from int, bool, -, *, -, and list
  but NOT real or ->
equality in ML

e_1 = e_2

• Only for expressions whose type is an equality type

• Equality types are built from

  int, bool, - * -, and -list
equality in ML

\[ e_1 = e_2 \]

- Only for expressions whose type is an equality type
- Equality types are built from \textit{int}, \textit{bool}, \textit{- * -}, and \textit{-list} \\
  - \(1+1 = 2;\)  \\
  - \text{val it = true : bool}
equality in ML

\[ e_1 = e_2 \]

- Only for expressions whose type is an \textit{equality} type
- Equality types are built from \texttt{int}, \texttt{bool}, - * -, and \texttt{-list}

- \( 1+1 = 2; \)  
  val it = true : bool

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

\[ e_1 = e_2 \]

- Only for expressions whose type is an equality type
- Equality types are built from `int`, `bool`, `- * -`, and `-list`

- \( 1+1 = 2; \)
  val it = true : bool

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

- \((\text{fn } x \Rightarrow x+x) = (\text{fn } y \Rightarrow 2*y); \)
  Error: operator and operand don't agree [equality type required]
equality in ML

\[ e_1 = e_2 \]

- Only for expressions whose type is an equality type
- Equality types are built from `int`, `bool`, `- * -`, and `-list`

  - `1+1 = 2;`
  - `val it = true : bool`
  - `\[1,1] = (0+1)::[2-1];`
  - `val it = true : bool`
equality in ML

\[ e_1 = e_2 \]

- Only for expressions whose type is an equality type
- Equality types are built from \texttt{int, bool, - * -, and -list}

- \( 1+1 = 2; \)
  val it = true : bool

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

- fun equal(x,y) = (x=y);
  val equal = fn - : "a * "a -> bool
equality in ML

\[ e_1 = e_2 \]

- Only for expressions whose type is an equality type
- Equality types are built from \texttt{int}, \texttt{bool}, - * -, and -list

- \( 1+1 = 2; \)
  val it = true : bool

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

- fun equal(x,y) = (x=y);
  val equal = fn - : "a * "a -> bool

\text{type variable } "a \text{ stands for any equality type}
notation overload

• ML syntax uses \(=\) for several purposes

\[
\begin{align*}
\text{fn} & \quad x \Rightarrow e \\
\text{fun} & \quad f(x) = e \\
\text{val} & \quad x = 2 \\
\text{val} & \quad \text{even} = \text{fn} \ x \Rightarrow (x \mod 2 = 0) \\
\text{fun} & \quad \text{leq}(x, y) = (x \leq y) \\
\text{fun} & \quad \text{geq}(x, y) = (x \geq y)
\end{align*}
\]

We also use \(=\) in math for “equality”
patterns

• ML includes _patterns_, for _matching_ with _values_

• Matching \( p \) to value \( v \) either _fails_, or _succeeds_ and binds names to values

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

(cannot attach type : \( t \) if desired)

**Syntactic restriction:**
each \( x \) occurs _at most once_ in \( p \)
pattern matching

- _ always matches v
- x always matches v (and binds x to v)
- n only matches n, true only matches true
- (p₁, p₂) matches (v₁, v₂) if p₁ matches v₁ and p₂ matches v₂ (combines the bindings)
- nil only matches the empty list
- p₁::p₂ matches non-empty lists v₁::v₂ for which p₁ matches v₁ and p₂ matches v₂ (combines the bindings)
utility

• When a value of a given type is expected, code can use patterns specific to that type

  integers… 0, 42, …, x, x:int…
  booleans… true, false, x, x:bool…
  3-tuples… (x, y, z), (0, true, _), …
  lists… nil, x::L, [x, y, z], …

  (x:int, L:int list)
  x::(y::L)
syntax

using patterns

declarations

\[
d ::= \textbf{val} \ p : t = e \\
| \quad \textbf{fun} \ f \ (p : t_1) : t_2 = e \\
| \quad \textbf{fun} \ f \ (p_1 : t) : t' = e_1 \ | f \ p_2 = e_2
\]

et cetera

expressions

\[
e ::= \textbf{fn} \ (p : t_1) : t_2 \Rightarrow e_2 \\
| \quad \textbf{case} \ e_0 : t \ of \ p_1 \Rightarrow e_1 \ | p_2 \Rightarrow e_2
\]

et cetera

optional : type annotations

fun, fn and case syntax allows \( k \) clauses
(all clauses must have the same type)
functions using patterns

\[
\text{\textbf{fun}} \ f \ p_1 = e_1 \mid ... \mid f \ p_k = e_k
\]

\[
\text{\textbf{fn}} \ p_1 \Rightarrow e_1 \mid ... \mid p_k \Rightarrow e_k
\]

\[f \ v \]

tries matching \( p_1 \) to \( v \),
then \( p_2, \ldots, p_k \)
until the first match
functions using patterns

\[
\begin{align*}
\textbf{fun} & \quad f \ p_1 = e_1 \mid \ldots \mid f \ p_k = e_k \\
\textbf{fn} & \quad p_1 \Rightarrow e_1 \mid \ldots \mid p_k \Rightarrow e_k
\end{align*}
\]

\[
\textbf{fun} \quad \text{len} \ [ \ ] = 0 \\
\mid \quad \text{len} \ (\_::L) = 1 + \text{len} \ L
\]

\[f \ v\]
tries matching \(p_1\) to \(v\), then \(p_2, \ldots, p_k\) until the first match
functions using patterns

\[
\text{fun } f \quad p_1 = e_1 \mid \ldots \mid f \quad p_k = e_k \\
\text{fn } p_1 \Rightarrow e_1 \mid \ldots \mid p_k \Rightarrow e_k
\]

\[
\text{fun } \text{len } [\ ] = 0 \\
\quad | \quad \text{len } (_::L) = 1 + \text{len } L
\]

len [3] = 1 + len [ ]

[3] doesn’t match pattern [ ]

len [3] = 1 + 0

[3] matches pattern _::L, binding L to [ ]
fun fact 0 = 1
  | fact 1 = 1
  | fact n = n * fact (n-1)

fun length [] = 0
  | length (_::L) = 1 + length L

fn [] => true | _ => false : 'a list -> bool

val x::L = [1,2,3] binds x to 1, L to [2,3]
rules of thumb

• Pay attention to clause order
  Tries \( p_1 \), then \( p_2 \), then \( p_3 \)  
  First match “wins”

• Use exhaustive patterns
  Every value of type \( t \) matches at least one of \( p_1, p_2, p_3 \)

• Avoid overlapping patterns (unless it’s safe)
  Every value of type \( t \) matches at most one of \( p_1, p_2, p_3 \)
  Or, if \( v \) matches \( p_i \) and \( p_j \) make sure \( e_i \) and \( e_j \) will be equal

• Can use \( _ \) when the binding is irrelevant
  Sometimes it’s convenient to use \( _ \) in the final clause

\[
\text{fun } f(p_1 : t) : t' = e_1 \\
| \quad f(p_2) = e_2 \\
| \quad f(p_3) = e_3
\]
Constant patterns can only be used to match values of an equality type

```plaintext
fun f(0) = 1
| f(1) = 1
| f(n) = f(n-1) + f(n-2)

case e of
  true => e_1
| false => e_2
```
Constant patterns can only be used to match values of an equality type.

```plaintext
fun f(0) = 1
|   f(1) = 1
|   f(n) = f(n-1) + f(n-2)

case e of
   true => e₁
   false => e₂

if e then e₁ else e₂
```
Using patterns

**divmod** : int * int -> int * int

```ml
fun divmod (x:int, y:int): int*int = (x div y, x mod y)
```

**fun check** (m:int, n:int): bool =

```ml
let
  val (q, r) = divmod (m, n)
in
  (q * n + r = m)
end
```
Using patterns

\[ \text{divmod : } \text{int} \times \text{int} \rightarrow \text{int} \times \text{int} \]

\[
\text{fun divmod (x:int, y:int): int*int = (x div y, x mod y)}
\]

\[
\text{fun check (m:int, n:int): bool =}
\]
\[
\text{let}
\]
\[
\text{val (q, r) = divmod (m, n)}
\]
\[
\text{in}
\]
\[
(q * n + r = m)
\]
\[
\text{end}
\]

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

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

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

What does this function do?
eval : int list -> int

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

\[
\begin{align*}
\text{fun } \text{eval} & \ (\text{[ ]}: \text{int list}) : \text{int} = 0 \\
| \quad & \text{eval} \ (d::L) = d + 10 \times (\text{eval} \ L)
\end{align*}
\]

\[
\text{eval} \ [2,4] \implies* 2 + 10 \times (\text{eval} \ [4]) \implies* 42
\]
eval : int list -> int

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

fun eval ([ ]:int list) : int = 0
\[\begin{align*}
| \text{eval (d::L)} &= d + 10 \times (\text{eval L}) \\
\end{align*}\]

\[\text{eval [2,4] } \Rightarrow* 2 + 10 \times (\text{eval [4]}) \Rightarrow* 42\]

What does this function do?
fun \(\log\) (\(x:\text{int}\)) : \text{int} = 
\begin{cases} 
0 & \text{if } x = 1 \\
1 + \log (x \div 2) & \text{otherwise}
\end{cases}

\log 3 = ???
log : int -> int

fun log (x:int) : int =  
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…
log : int -> int

fun log (x:int) : int =
  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?
fun log (x:int) : int =
  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**
  (showing *argument type* and *result type*)

• **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} *)
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 $2^k \leq x < 2^{k+1}$  *)

For all integers x such that x>0, the value of log x is an integer k such that $2^k \leq x < 2^{k+1}$
notes

• Can use $\Rightarrow^*$ or $=$ in specs

• Use math and logic 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 is a power of 2 *)

(* ENSURES log x = the integer k *)

(* such that 2^k = x *)
another log spec

```haskell
fun log (x:int) : int = 
  if x=1 then 0 else 1 + log (x div 2)

(* log : int -> int *)
(* REQUIRE x is a power of 2 *)
(* ENSURES log x = the integer k *)
(* such that 2^k = x *)
```

(a weaker spec … why?)
fun log (x:int) : int = 
   if x=1 then 0 else 1 + log (x div 2)

(* log : int -> int *)

(* REQUIRES x is a power of 2 *)

(* ENSURES log x = the integer k *)

(* such that 2^k = x *)

(a weaker spec ... why?)

(it's actually implied by the previous spec)
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 = the decimal digit list for n
            (with least significant digit first)

decimal 42 = [2,4]
eval spec

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

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

\textbf{REQUIRES} \quad R = \text{the decimal digit list for } n

\textbf{ENSURES} \quad \text{eval} \ R = n
connection

• **eval** and **decimal** are designed to fit together

• They satisfy a *combined spec*

| TYPE          | decimal : int -> int list  
|              | eval : int list -> int   |
| REQUIREES    | n ≥ 0                     |
| ENSURES      | eval(decimal n) = n       |
**connection**

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

```plaintext
TYPE    decimal : int -> int list
        eval : int list -> int

REQUIRES n ≥ 0

ENSURES eval(decimal n) = n
```

**NOTE:** this spec tells us that `decimal n` evaluates to a value, for `n ≥ 0`
Evaluation

- Expression evaluation produces a value if it terminates
  - $e \xrightarrow{k} e'$  e evaluates to $e'$ in $k$ steps
  - $e \xrightarrow{*} v$  e evaluates to $v$ in finitely many steps

- Declarations produce value bindings
  - $d \xrightarrow{*} x_1:v_1, \ldots, x_k:v_k$

- Matching a pattern to a value either succeeds with bindings, or fails
  - $\text{match}(p, v) \xrightarrow{*} x_1:v_1, \ldots, x_k:v_k | \text{fail}$
Substitution

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

\[
\left[ x_1: v_1, \ldots, x_k: v_k \right] e
\]

for the expression obtained by substituting \( v_1 \) for \( x_1, \ldots, v_k \) for \( x_k \) in \( e \)

(substitute for free occurrences, only)

\[
\begin{align*}
\left[ x: 2 \right] (x + x) & \quad \text{is} \quad 2 + 2 \\
\left[ x: 2 \right] (\text{fn } y \Rightarrow x + y) & \quad \text{is} \quad \text{fn } y \Rightarrow 2 + y \\
\left[ x: 2 \right] (\text{fn } x \Rightarrow x + x) & \quad \text{is} \quad \text{fn } x \Rightarrow x + x
\end{align*}
\]
rules

(mostly for sequential evaluation)

• For each syntactic construct we give evaluation rules for \( \Rightarrow \) ("one-step-to")
  • showing order-of-evaluation

• We derive evaluation laws for \( \Rightarrow^* \) ("many-steps-to")
  • how expressions evaluate
  • what is the value, if it terminates

• We can also count number of steps \( \Rightarrow^{(n)} \) ("takes \( n \) steps to")
addition rules

$e_1 + e_2$ evaluates from left-to-right

$e_1 \Rightarrow e_1'$

$\frac{\bullet}{e_1 + e_2 \Rightarrow e_1' + e_2}$

if $e_1$ steps to $e_1'$, then $e_1 + e_2$ steps to $e_1' + e_2$

$e_2 \Rightarrow e_2'$

$\frac{\bullet}{v_1 + e_2 \Rightarrow v_1 + e_2'}$

$e_i, v_i : \text{int}$

$v_1 + v_2$

steps to

the numeral for $v_1 + v_2$

$v_1 + v_2 \Rightarrow v$

where $v = v_1 + v_2$
addition law
(follows from the rules)

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

then
\[ e_1 + e_2 \implies^* v_1 + e_2 \implies^* v_1 + v_2 \implies v \]
\[ e_1 + e_2 \implies^* v \]
addition law
(follows from the rules)

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

(2+2) + (3+3)
\[ \implies 4 + (3+3) \]
\[ \implies 4 + 6 \]
\[ \implies 10 \]
addition law
(follows from the rules)

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

\[(2+2) + (3+3) \]  \[\implies (2+2) + (3+3) \implies^* 10 \]
\[\implies 4 + (3+3) \]
\[\implies 4 + 6 \]
\[\implies 10 \]
addition law

(follows from the rules)

If

$$e_1 \implies^* v_1 \quad \text{and} \quad e_2 \implies^* v_2 \quad \text{and} \quad v = v_1 + v_2$$

then

$$e_1 + e_2 \implies^* v_1 + e_2 \implies^* v_1 + v_2 \implies v$$

$$e_1 + e_2 \implies^* v$$

$$(2+2) + (3+3) \implies 4 + (3+3) \implies 4 + 6 \implies 10$$

$$(2+2) + (3+3) \implies^* 10$$

$$(2+2) + (3+3) \implies^{(3)} 10$$
addition law
(also follows from the rules)

If $e_1 + e_2 \implies^* v$

there must be $v_1 : \text{int}$ and $v_2 : \text{int}$ such that

$$e_1 \implies^* v_1 \quad \text{and} \quad e_2 \implies^* v_2 \quad \text{and} \quad v = v_1 + v_2$$

and the evaluation looks like

$$e_1 + e_2 \implies^* v_1 + e_2 \implies^* v_1 + v_2 \implies v$$

(this shows the order of evaluation clearly!)
application rules

“a function always evaluates its argument”

\[
\begin{align*}
e_1 & \Rightarrow e_1' \\
e_1 \, e_2 & \Rightarrow e_1' \, e_2 \\
e_2 & \Rightarrow e_2'
\end{align*}
\]

\((\textsf{fn} \, x \Rightarrow e) \, e_2 \Rightarrow (\textsf{fn} \, x \Rightarrow e) \, e_2'
\]

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

*(this rule only applicable when function and argument have been evaluated to values)*

\(e_1 \, e_2\)

evaluates \(e_1\) to a function, evaluates \(e_2\) to a value, substitutes the value into the function body, then evaluates the body

*(call-by-value)*
application law

(follows from the rules)

If

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

then

\[ e_1 \ e_2 \Rightarrow^* [x:v]e \]
application law

(follows from the rules)

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

this expression may need further evaluation
application law
(also follows from the rules)

If \( e_1 \ e_2 \Rightarrow^* v \)
there must be values

\[(\texttt{fn} \ x \Rightarrow \ e) : t_1 \rightarrow t_2 \text{ and } v_2 : t_1\]
such that

\[e_1 \Rightarrow^* (\texttt{fn} \ x \Rightarrow \ e) \text{ and } e_2 \Rightarrow^* v_2\]
and

\[e_1 \ e_2 \Rightarrow^* (\texttt{fn} \ x \Rightarrow \ e) \ e_2\]
\[\Rightarrow^* (\texttt{fn} \ x \Rightarrow \ e) \ v_2\]
\[\Rightarrow \ [x:v_2] e\]
\[\Rightarrow^* v\]
More rules

• **div** and **mod** evaluate from left to right

• **List expressions**
  
  
  \([e_1, \ldots, e_n], e_1 :: e_2, \text{ and } e_1 @ e_2\)
  
  all evaluate from left to right

• **Tuple expressions** \((e_1, \ldots, e_n)\)
  
  can be evaluated from left to right, or (as we’ll see later) in parallel.
More rules

- `div` and `mod` evaluate from left to right
- List expressions `\[e_1, \ldots, e_n\]`, `e_1 :: e_2`, and `e_1 @ e_2` all evaluate from left to right
- Tuple expressions `(e_1, \ldots, e_n)` can be evaluated from left to right, or (as we’ll see later) in parallel.
In the scope of \textbf{fun} \( f(p) = e, \)

\[
\begin{align*}
\quad f & \iff (\textbf{fn} \ p \ => \ e) \\
\end{align*}
\]
Declaration rule

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

\[
\begin{align*}
& f \iff (\texttt{fn} \ p \Rightarrow e) \\
\end{align*}
\]

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

In the scope of $\textbf{fun } f(p) = e$, 

\[ f \implies (\textbf{fn } p => e) \]

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

divmod (3,2)
Declaration rule

In the scope of `fun f(p) = e`,

\[ f \implies (\text{fn } p \implies e) \]

`fun` `divmod(x, y) = (x div y, x mod y)`

`divmod (3,2)`

\[ \implies (\text{fn}(x, y) \implies (x \text{ div } y, x \text{ mod } y)) (3,2) \]
Declaration rule

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

\[
  f \implies (\texttt{fn} \ p \Rightarrow e)
\]

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

\( \text{divmod} \ (3,2) \)

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

\[
  \implies (3 \ \texttt{div} \ 2, 3 \ \texttt{mod} \ 2)
\]
**Declaration rule**

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

\[
\begin{array}{c}
f \iff \texttt{(fn} \ p \Rightarrow e) \\
\end{array}
\]

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

\texttt{divmod} (3,2)

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

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

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

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

\[
\begin{align*}
f & \implies (\texttt{fn}~p~\Rightarrow~e)
\end{align*}
\]

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

\[
\begin{align*}
\texttt{divmod}~(3,2) \\
& \implies (\texttt{fn}(x, y) \Rightarrow (x \texttt{ div} y, x \texttt{ mod} y))~(3,2) \\
& \implies (3 \texttt{ div} 2, 3 \texttt{ mod} 2) \\
& \implies (1, 3 \texttt{ mod} 2) \\
& \implies (1, 1)
\end{align*}
\]
example

fun silly x = silly x;
(fn y => 0) (silly 42) doesn’t terminate
fun silly x = silly x;
(fn y => 0) (silly 42)  
doesn’t terminate

(fn y => 0) (silly 42)
example

```plaintext
fun silly x = silly x;
(fn y => 0) (silly 42)  
doesn’t terminate

(fn y => 0) (silly 42)

⇒
```
fun silly x = silly x;
(fn y => 0) (silly 42)  doesn’t terminate

(fn y => 0) (silly 42)

⟹  (fn y => 0) ((fn x => silly x) 42)
example

\[
\text{fun silly } x = \text{ silly } x; \\
(fn \ y \Rightarrow 0) \ \text{ (silly 42)} \quad \text{doesn't terminate}
\]

\[
(fn \ y \Rightarrow 0) \ \text{ (silly 42)} \\
\implies (fn \ y \Rightarrow 0) \ ((fn \ x \Rightarrow \text{ silly } x) \ 42)
\]
fun silly x = silly x;

(fn y => 0) (silly 42)  
doesn’t terminate

(fn y => 0) (silly 42)

⟹ (fn y => 0) ((fn x => silly x) 42)

⟹ (fn y => 0) (silly 42)
example

\[
\textbf{fun} \ \textit{silly} \ x = \textit{silly} \ x;
\]

\[
(\textbf{fn} \ y \Rightarrow 0) \ (\textit{silly} \ 42) \quad \text{doesn’t terminate}
\]

\[
(\textbf{fn} \ y \Rightarrow 0) \ (\textit{silly} \ 42)
\]

\[
(\textbf{fn} \ y \Rightarrow 0) \ (\textbf{fn} \ x \Rightarrow \textit{silly} \ x) \ 42)
\]

\[
(\textbf{fn} \ y \Rightarrow 0) \ (\textit{silly} \ 42)
\]

\textit{ad infinitum}
example

fun silly x = silly x;

(fn y => 0) (silly 42)  

doesn’t terminate

(fn y => 0) (silly 42)

⟹ (fn y => 0) ((fn x => silly x) 42)

⟹ (fn y => 0) (silly 42)  

functions evaluate their argument

ad infinitum
Comments

• Using \( \implies \) we can talk about evaluation order and the number of steps

• But we may want to ignore such details...

For all expressions \( e_1, e_2 : \text{int} \) and all values \( v : \text{int} \),

if \( e_1 + e_2 \implies^* v \) then \( e_2 + e_1 \implies^* v \)

**Here we only care about the value**

For all expressions \( e_1, e_2 : \text{int} \),

\[ e_1 + e_2 = e_2 + e_1 \]
Equivalence

(it’s all about the value…)

• 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
Equivalence

(it’s all about the value…)

• For each type \( t \) there is a mathematical notion of equivalence (or equality) \( =_t \) for values of type \( t \)

\[ \forall v_1 =_{\text{int}} v_2 \iff v_1 = v_2 \] (as expected!)

• **Expressions** of type \( t \) are equivalent iff they evaluate to equivalent values, or both diverge
Equivalence

(it’s all about the value…)

• 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
Equivalence
(it’s all about the value…)

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

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

• *Expressions* of type $t$ are equivalent iff they evaluate to *equivalent* values, or both diverge
Equivalence
(it’s all about the value…)

• For each type \( t \) there is a mathematical notion of equivalence (or equality) \( =_t \) for values of type \( t \)

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

(equivalent functions map equal arguments to equal results)

• **Expressions** of type \( t \) are equivalent iff they evaluate to equivalent values, or both diverge
Equations

• **Arithmetic**

\[ e + 0 =_{\text{int}} e \]
\[ e_1 + e_2 =_{\text{int}} e_2 + e_1 \]
\[ e_1 + (e_2 + e_3) =_{\text{int}} (e_1 + e_2) + e_3 \]
\[ 21 + 21 =_{\text{int}} 42 \]

• **Boolean**

\[ \text{if true then } e_1 \text{ else } e_2 =_t e_1 \]
\[ \text{if false then } e_1 \text{ else } e_2 =_t e_2 \]
\[ (0 < 1) =_{\text{bool}} \text{true} \]
Equations

- Application

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

- Declaration

In the scope of

\[
\text{fun } f(x:tl):t2 = e
\]

the equation

\[
f =_{tl->t2} (\text{fn } x => e)
\]

holds
Equations

- Application

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

- Declaration

In the scope of

\[\text{fun } f(x:t_1):t_2 = e\]

the equation

\[f =_{t_1 \rightarrow t_2} (\text{fn } x \Rightarrow e)\]

holds
Equations

let val x = v in e end = [x:v]e

• Application

(fn x => e) v = [x:v]e

• Declaration

In the scope of

fun f(x:t1):t2 = e

the equation

f =_{t1->t2} (fn x => e)

holds
• 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
Key facts

evaluation is consistent with equivalence
Key facts

*evaluation is consistent with equivalence*

- $e : t$ and $e \Rightarrow^* v$ implies $v : t$ and $e =_t v$
- $e \Rightarrow^* v$ implies $(\text{fn } x \Rightarrow E) e = [x:v] E$
Key facts

evaluation is consistent with equivalence

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

Standard ML of New Jersey

```ml
fun f(x:int) = 0;
...
- f 3;
- val it = 0 : int
```

```
f 3 \Rightarrow^* 0
f 3 =_{\text{int}} 0
```
Summary

• Patterns allow *elegant* function design
  - patterns match subset of values
  - function tries its clauses in order
  - so be careful about clause order

• Specifications can serve as *clear* documentation
  - TYPE + REQUIRES and ENSURES
  - equality and evaluation
Testing may be helpful, but usually cannot cover all cases.

How to prove that a function meets its specification...

Proof methods use induction.

Coming soon

"I want a computer that does what I want it to do, not what I tell it to do!"
Coming soon

• Testing may be helpful, but usually cannot cover all cases

• How to prove that a function meets its specification...

• Proof methods use induction