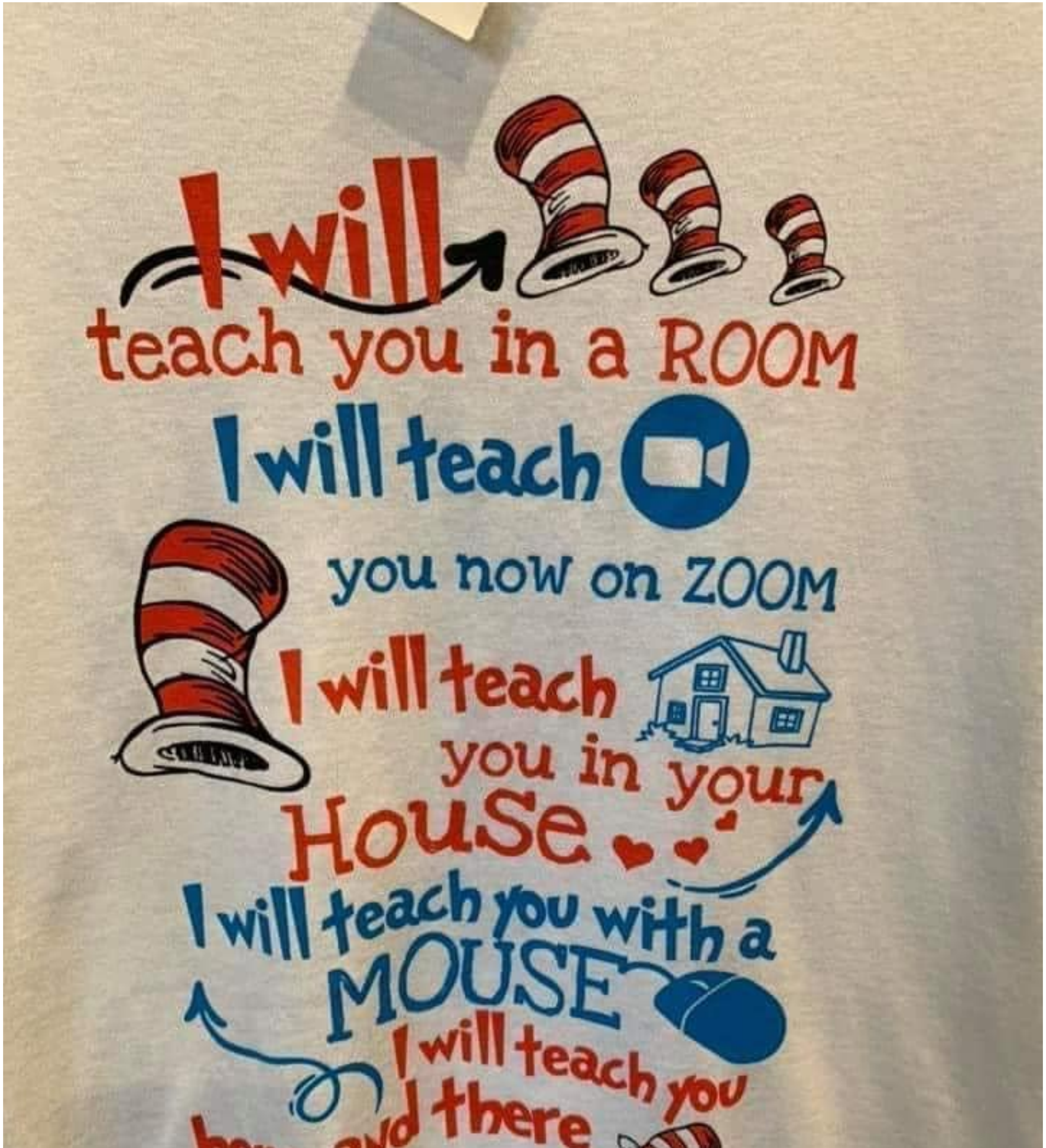
15-150 Fall 2020

Stephen Brookes

Lecture 3

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
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 *equality* type
- Equality types are built from

  int, **bool**, **- * -**, and **-list**

int list

e.g. **int * bool**

(int * bool) 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**

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

int, bool, - * -, and **-list**

```
- 1+1 = 2;
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
```

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

- (fn x => x+x) = (fn y => 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

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

  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
```

type variable ''a
stands for
any equality type

notation overload

- ML syntax uses = for several purposes

```
fn x => e
fun f(x) = e
val x = 2
val even = fn x => (x mod 2 = 0)
fun leq(x, y) = (x <= y)
fun geq(x, y) = (x >= y)
```

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

$$
\begin{aligned}
p ::= \; & \_ \mid x \mid \mathbf{n} \mid \mathbf{true} \mid \mathbf{false} \\
\mid \; & (p_1, \ldots, p_k) \\
\mid \; & p_1 :: p_2 \mid [p_1, \ldots, p_k]
\end{aligned}
$$

(can attach type $: t$ if desired)

Syntactic restriction:
each x occurs *at most once* in p

pattern matching

with values

- _ always matches v
- x always matches v (and binds x to v)
- **n** only matches **n**, **true** only matches **true**
- (p_1, p_2) matches (v_1, v_2)
  if p_1 matches v_1 and p_2 matches v_2
  (*combines* the bindings)

no ambiguity,
because of
variable constraint

- nil only matches the empty list
- p_1::p_2 matches non-empty lists v_1::v_2 for
  which p_1 matches v_1 and p_2 matches v_2
  (*combines* the bindings)

no ambiguity,
because of
variable constraint

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

$$
\begin{aligned}
d ::= &\ \textbf{val}\ p : t = e \\
|\ &\ \textbf{fun}\ f\ (p{:}t_1){:}t_2 = e \\
|\ &\ \textbf{fun}\ f\ (p_1 : t) : t' = e_1 \mid f\ p_2 = e_2
\end{aligned}
$$

et cetera

expressions

$$
\begin{aligned}
e ::= &\ \textbf{fn}\ (p{:}t_1){:}t_2 \Rightarrow e_2 \\
|\ &\ \textbf{case}\ e_0 : t\ \textbf{of}\ p_1 \Rightarrow e_1 \mid p_2 \Rightarrow e_2
\end{aligned}
$$

et cetera

optional : *type* annotations

fun, **fn** and **case** syntax allows k clauses

(all clauses must have the same type)

functions using patterns

fun $f\ p_1 = e_1 \mid \ldots \mid f\ p_k = e_k$

fn $p_1 \Rightarrow e_1 \mid \ldots \mid p_k \Rightarrow e_k$

$f\ v$

tries matching p_1 to v,
then $p_2, \ldots, p_k$
until the first match

functions using patterns

fun f p_1 = e_1 | … | f p_k = e_k

fn p_1 => e_1 | … | p_k => e_k

f v
tries matching p_1 to v,
then p_2,…, p_k
until the first match

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

functions using patterns

fun $f\ p_1 = e_1 \mid \ldots \mid f\ p_k = e_k$

fn $p_1 \Rightarrow e_1 \mid \ldots \mid p_k \Rightarrow e_k$

f v

tries matching p_1 to v,
then $p_2, \ldots, p_k$
until the first match

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

```
len [3]
  = 1 + len [ ]
  = 1 + 0
```

[3] doesn't match pattern [ ]

[3] matches pattern _::L, binding L to [ ]

examples

using patterns

```
fun fact 0 = 1
 |   fact 1 = 1
 |   fact n = n * fact (n-1)
```

fact : int -> int

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

length : 'a list -> int

```
fn [ ] => true | _ => false
```

: 'a list -> bool

```
val x::L = [1,2,3]
```

binds x to 1, L to [2,3]

rules of thumb

fun $f(p_1{:}t){:}t' = e_1$
$|\quad f(p_2) = e_2$
$|\quad f(p_3) = e_3$

case $e{:}t$ **of**
$p_1 \Rightarrow e_1$
$|\quad p_2 \Rightarrow e_2$
$|\quad p_3 \Rightarrow e_3$

- 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

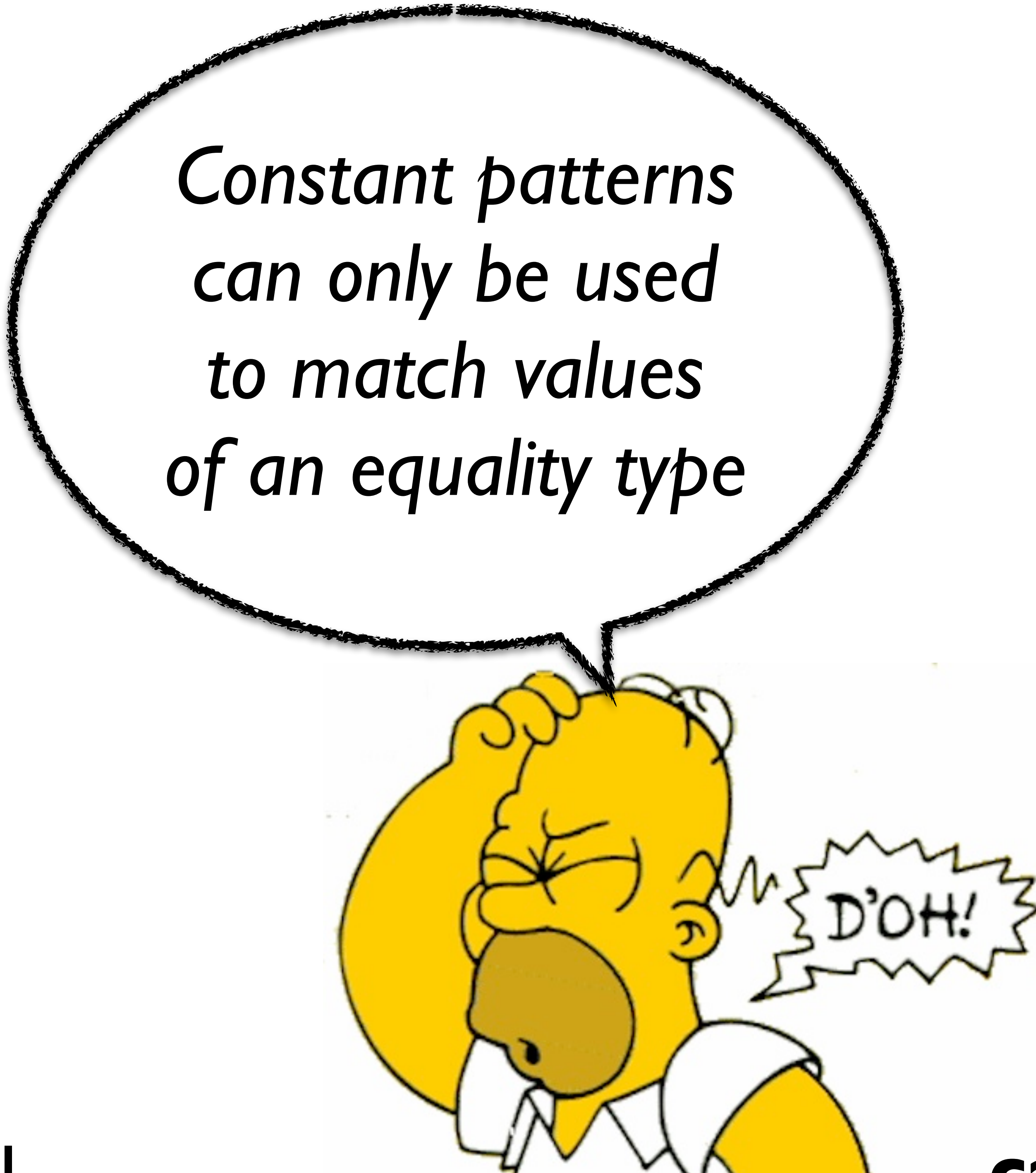
- Avoid *overlapping* patterns (unless it's safe)

  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



int

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

bool

case e **of**
   true => e_1
 | **false** => e_2

Constant patterns can only be used to match values of an equality type

int

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

bool

```
case e of
    true => e1
  | false => e2
```

if e **then** e_1 **else** e_2

Using patterns

divmod : int * int -> int * int

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

```
fun check (m:int, n:int): bool =
let
   val (q, r) = divmod (m, n)
in
   (q * n + r = m)
end
```

Using patterns

divmod : int * int -> int * int

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

```
fun check (m:int, n:int): bool =
let
   val (q, r) = divmod (m, n)
in
   (q * n + r = m)
end
```

What does this function do?

decimal : int -> int list

```
fun decimal (n:int) : int list =
    if n < 10 then [n]
              else (n mod 10) :: decimal (n div 10)
```

decimal : int -> int list

```
fun decimal (n:int) : int list =
    if n < 10 then [n]
              else (n mod 10) :: decimal (n div 10)
```

What does this function do?

decimal : int -> int list

```
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

```
fun eval ([ ]:int list) : int = 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] ⟹* 2 + 10 * (eval [4]) ⟹* 42

eval : int list -> int

```
fun eval ([ ]:int list) : int = 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] ⟹* 2 + 10 * (eval [4]) ⟹* 42

What does this function do?

log : int -> int

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

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?

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?
  - 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)

Format

```
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 ....    *)
```

type

assumption

guarantee

For all values x : int satisfying the assumption,
log x : int and its value satisfies the guarantee

Any ideas?

log spec

```
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)    *)
```

$\log x$ = the integer $k \geq 0$ such that $2^k \leq x < 2^{k+1}$

log spec

```
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 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 $\Longrightarrow^*$ or $=$ in specs
- Use *math and logic **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 is a power of 2                   *)
(* ENSURES  log x = the integer k                *)
(*                 such that 2^k = x             *)
```

another log spec

```
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?)

another log spec

```
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)

decimal 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 $n \geq 0$

ENSURES decimal n = the decimal digit list for n (with *least significant digit* first)

decimal 42 = [2,4]

eval spec

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

TYPE   eval : int list -> int

REQUIRES   R = the decimal digit list for n

ENSURES   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
REQUIRES	$n \geq 0$
ENSURES	eval(decimal n) = n

connection

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

TYPE	decimal : int -> int list eval : int list -> int
REQUIRES	$n \geq 0$
ENSURES	eval(decimal n) = n

NOTE: this spec tells us that
decimal n evaluates to a value, for $n \geq 0$

Evaluation

- Expression evaluation produces a value if it terminates
  - $e \Longrightarrow^{k} e'$ — *e evaluates to e' in k steps*
  - $e \Longrightarrow^{*} v$ — *e evaluates to v in finitely many steps*
- Declarations produce value bindings
  - $d \Longrightarrow^{*} x_1{:}v_1, \ldots, x_k{:}v_k$
- Matching a pattern to a value either *succeeds* with bindings, or *fails*
  - $\mathrm{match}(p, v) \Longrightarrow^{*} x_1{:}v_1, \ldots, x_k{:}v_k \mid$ **fail**

TYPE SAFE

Substitution

For bindings $x_1{:}v_1, \ldots, x_k{:}v_k$ and expression e

we write

$$[\![\; x_1{:}v_1, \ldots, x_k{:}v_k \;]\!]\; e$$

for the expression obtained by substituting

v_1 for $x_1, \ldots, v_k$ for x_k in e

(substitute for free occurrences, only)

$[\![\; x{:}2 \;]\!]\; (x + x)$	is	$2 + 2$
$[\![\; x{:}2 \;]\!]\; (\mathrm{fn}\ y => x + y)$	is	$\mathrm{fn}\ y => 2 + y$
$[\![\; x{:}2 \;]\!]\; (\mathrm{fn}\ x => x + x)$	is	$\mathrm{fn}\ x => x + x$

rules (mostly for sequential evaluation)

- For each syntactic construct we give *evaluation rules* for $\Longrightarrow$ ("*one-step-to*")
    - showing order-of-evaluation
- We derive *evaluation laws* for $\Longrightarrow^{*}$ ("*many-steps-to*")
    - *how* expressions evaluate
    - what is the value, if it terminates
- We can also count number of steps $\Longrightarrow^{(n)}$

  ("*takes n steps to*")

addition rules

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

$$\frac{e_1 \Longrightarrow e_1'}{e_1 + e_2 \Longrightarrow e_1' + e_2}$$

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

e_i, v_i : int

$$\frac{e_2 \Longrightarrow e_2'}{v_1 + e_2 \Longrightarrow v_1 + e_2'}$$

v_1+v_2
steps to
the numeral for v_1+v_2

$$\frac{}{v_1 + v_2 \Longrightarrow v}$$ where $v = v_1 + v_2$

addition law

(follows from the rules)

If

$e_1 \Longrightarrow^* v_1$ and $e_2 \Longrightarrow^* v_2$ and $v = v_1 + v_2$

then

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

$e_1 + e_2 \Longrightarrow^* v$

addition law

(follows from the rules)

If

$e_1 \Longrightarrow^* v_1$ and $e_2 \Longrightarrow^* v_2$ and $v = v_1 + v_2$

then

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

$e_1 + e_2 \Longrightarrow^* v$

$(2+2) + (3+3)$

$\Longrightarrow 4 + (3+3)$

$\Longrightarrow 4 + 6$

$\Longrightarrow 10$

addition law

(follows from the rules)

If

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

then

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

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

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

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

addition law

(follows from the rules)

If

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

then

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

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

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

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

$$(2+2) + (3+3) \Longrightarrow^{(3)} 10$$

addition law

(also follows from the rules)

If $e_1 + e_2 \Longrightarrow^* v$

there must be $v_1 : int$ and $v_2 : int$ such that

$e_1 \Longrightarrow^* v_1$ and $e_2 \Longrightarrow^* v_2$ and $v = v_1 + v_2$

and the evaluation looks like

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

(this shows the order of evaluation clearly!)

application rules

"a function always evaluates its argument"

$$\frac{e_1 \Longrightarrow e_1'}{e_1\ e_2 \Longrightarrow e_1'\ e_2}$$

$$\frac{e_2 \Longrightarrow e_2'}{(\mathbf{fn}\ x \texttt{=>}\ e)\ e_2 \Longrightarrow (\mathbf{fn}\ x \texttt{=>}\ e)\ e_2'}$$

$$\frac{}{(\mathbf{fn}\ x \texttt{=>}\ e)\ v \Longrightarrow [\![\ x{:}v\ ]\!]\ e}$$

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

(call-by-value)

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

application law

(follows from the rules)

If

$e_1 \Longrightarrow^* (\textbf{fn}\ x \Rightarrow e)$ and $e_2 \Longrightarrow^* v$

then

$e_1\ e_2 \Longrightarrow^* [\![x{:}v]\!]e$

application law

(follows from the rules)

If

$e_1 \Longrightarrow^* (\textbf{fn}\ x \Rightarrow e)$ and $e_2 \Longrightarrow^* v$

then

$e_1\ e_2 \Longrightarrow^* [\![x{:}v]\!]e$

this expression may need further evaluation

application law

(also follows from the rules)

If $e_1\ e_2 \Longrightarrow^* v$

there must be values

$(\textbf{fn}\ x \Rightarrow e) : t_1\text{->}t_2$ and $v_2 : t_1$

such that

$e_1 \Longrightarrow^* (\textbf{fn}\ x \Rightarrow e)$ and $e_2 \Longrightarrow^* v_2$

and

$$
\begin{aligned}
e_1\ e_2 &\Longrightarrow^* (\textbf{fn}\ x \Rightarrow e)\ e_2 \\
&\Longrightarrow^* (\textbf{fn}\ x \Rightarrow e)\ v_2 \\
&\Longrightarrow [\![x{:}v_2]\!]e \\
&\Longrightarrow^* v
\end{aligned}
$$

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.

More rules

Declaration rule

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

$$\frac{}{\texttt{f} \Longrightarrow (\textbf{fn}\ \texttt{p => e})}$$

Declaration rule

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

$$\frac{}{\texttt{f} \Longrightarrow (\textbf{fn}\ \texttt{p => e})}$$

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

Declaration rule

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

$$\frac{}{f \Longrightarrow (\textbf{fn}\ p \Rightarrow e)}$$

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

divmod (3,2)

Declaration rule

$$\frac{\text{In the scope of } \textbf{fun}\ f(p) = e,}{f \Longrightarrow (\textbf{fn}\ p \Rightarrow e)}$$

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

divmod (3,2)

$\Longrightarrow$ (**fn**(x, y) => (x **div** y, x **mod** y)) (3,2)

Declaration rule

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

$$\frac{}{f \Longrightarrow (\textbf{fn}\ p \Rightarrow e)}$$

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

divmod (3,2)

$\Longrightarrow$ (**fn**(x, y) => (x **div** y, x **mod** y)) (3,2)

$\Longrightarrow$ (3 **div** 2, 3 **mod** 2)

Declaration rule

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

$$\frac{}{\texttt{f} \Longrightarrow (\textbf{fn}\ \texttt{p => e})}$$

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

divmod (3,2)

⟹ (**fn**(x, y) => (x **div** y, x **mod** y)) (3,2)

⟹ (3 **div** 2, 3 **mod** 2)

⟹ (1, 3 **mod** 2)

Declaration rule

$$\frac{\text{In the scope of } \textbf{fun}\ f(p) = e,}{f \Longrightarrow (\textbf{fn}\ p \Rightarrow e)}$$

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

divmod (3,2)
⟹ (**fn**(x, y) => (x **div** y, x **mod** y)) (3,2)
⟹ (3 **div** 2, 3 **mod** 2)
⟹ (1, 3 **mod** 2)
⟹ (1, 1)

example

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

example

fun silly x = silly x;

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

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

example

fun silly x = silly x;

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

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

$\Longrightarrow$

example

fun silly x = silly x;

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

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

$\Longrightarrow$ (**fn** y => 0) ((**fn** x => silly x) 42)

example

fun silly x = silly x;

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

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

$\Longrightarrow$ (**fn** y => 0) ((**fn** x => silly x) 42)

$\Longrightarrow$

example

fun silly x = silly x;

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

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

$\Longrightarrow$ (**fn** y => 0) ((**fn** x => silly x) 42)

$\Longrightarrow$ (**fn** y => 0) (silly 42)

example

fun silly x = silly x;

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

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

$\Longrightarrow$ (**fn** y => 0) ((**fn** x => silly x) 42)

$\Longrightarrow$ (**fn** y => 0) (silly 42)

ad infinitum

example

fun silly x = silly x;

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

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

$\Longrightarrow$ (**fn** y => 0) ((**fn** x => silly x) 42)

$\Longrightarrow$ (**fn** y => 0) (silly 42)

ad infinitum

functions evaluate their argument

Comments

- Using $\Longrightarrow$ 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 : int and all values v:int,

if $e_1 + e_2 \Longrightarrow^* v$ then $e_2 + e_1 \Longrightarrow^* v$

Here we only care about the value

For all expressions e_1, e_2 : int,

$e_1 + e_2 = e_2 + e_1$

the same, more succinctly

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

$$v_1 =_{int} v_2 \Leftrightarrow v_1 = v_2$$ (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

$$f_1 =_{int\text{->}int} f_2 \Leftrightarrow$$

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

- ***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

$$f_1 =_{int\text{->}int} f_2 \Leftrightarrow$$

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

(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**

$$e + 0 =_{int} e$$

$$e_1 + e_2 =_{int} e_2 + e_1$$

$$e_1 + (e_2 + e_3) =_{int} (e_1 + e_2) + e_3$$

$$21 + 21 =_{int} 42$$

- **Boolean**

$$\textbf{if true then } e_1 \textbf{ else } e_2 =_t e_1$$

$$\textbf{if false then } e_1 \textbf{ else } e_2 =_t e_2$$

$$(0 < 1) =_{bool} \textbf{true}$$

Equations

- **Application**

only when the argument is a value

$$(\textbf{fn}\ x \Rightarrow e)\ v\ =\ [\![x{:}v]\!]e$$

- **Declaration**

In the scope of

$$\textbf{fun}\ f(x{:}t1){:}t2 = e$$

the equation

$$f =_{t1\text{->}t2} (\textbf{fn}\ x \Rightarrow e)$$

holds

Equations

- **Application**

$$(\textbf{fn}\ x \Rightarrow e)\ v \;=\; [\![x{:}v]\!]e$$

- **Declaration**

In the scope of

$$\textbf{fun}\ f(x{:}t1){:}t2 = e$$

the equation

$$f =_{t1\text{->}t2} (\textbf{fn}\ x \Rightarrow e)$$

holds

Equations

$$\textbf{let val } x = v \textbf{ in } e \textbf{ end} = [\![x{:}v]\!]e$$

- **Application**

$$(\textbf{fn } x \Rightarrow e)\ v = [\![x{:}v]\!]e$$

- **Declaration**

In the scope of

$$\textbf{fun } f(x{:}t1){:}t2 = e$$

the equation

$$f =_{t1\text{->}t2} (\textbf{fn } x \Rightarrow e)$$

holds

Compositionality

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

Key facts

evaluation *is consistent with* ***equivalence***

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

Standard ML of New Jersey

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

$f\ 3 \Longrightarrow^* 0$

$f\ 3 =_{int} 0$

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

"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*