Recursion and induction
So far

- Using \( => \) and \( =>^* \) we can talk precisely about program behavior

- But we may not care about evaluation order...

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

\[
\text{if } e_1 + e_2 =>^* v \text{ then } e_2 + e_1 =>^* v
\]

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

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 $\text{int}$
• 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 directly using $\Rightarrow^*$
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

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

two function values are equivalent iff they map equivalent arguments to equivalent results
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 \( e_2 \) = \( e_2 \)

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

- **Applications**
  \[(\text{fn } x \Rightarrow e) \; v \; = \; [x:v]e\]

- **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

function addl(x, y) = x + y
function 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 $\text{addl} = \text{int} * \text{int} \rightarrow \text{int}$ $\text{addr}$

• 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

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

\[(\text{fn } p_1 \Rightarrow e_1 \mid p_2 \Rightarrow e_2) \ v = [B_2]e_2 \]
  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 $=$ to specify the *applicative behavior* of functional programs

• Equality is *compositional*

• Equality is *defined* in terms of evaluation

• $=>^*$ is *consistent* with $=$ and ML evaluation
Example

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

(* f : int -> int                *)
(* REQUIRES  x \geq 0      *)
(* ENSURES   f x = 1     *)

f x = 1

means the same as

f x =>* 1
fun eval ([ ]:int list) : int = 0
| eval (d::L) = d + 10 * (eval L);

REQUIRES:
  every integer in L is a decimal digit

ENSURES:
  eval(L) evaluates to a non-negative integer
fun decimal (n:int) : int list = 
  if n<10 then [n]
  else (n mod 10) :: decimal (n div 10);

decimal : int -> int list

REQUIRES:  n >= 0

ENSURES: 
decimal(n) = a list L of decimal digits, such that eval(L) = n
Proofs

How to show that a specification is valid

- **Proofs** by **induction**
  - templates to help with accuracy
- Focus on **examples**

program structure guides proof
What is a proof?

A proof is a logical sequence of steps, leading to a conclusion.

- Each step must follow logically from math facts, or the results of earlier steps.
Simple induction

• To prove a property of the form
  \[ \forall n \geq 0. \, P(n) \]

• First, prove \( P(0) \).

• Then show that, for all \( k \geq 0 \),
  \( P(k+1) \) follows logically from \( P(k) \).
Why this works

• P(0) gets a direct proof  \textit{base}

• P(0) implies P(1)  \textit{step}

• P(1) implies P(2)  \textit{step}

• ...

For each n \geq 0 we can establish P(n)

(follows from base after n uses of step)
Example

\textbf{fun}\ f(x:\text{name}):\text{int} = \textbf{if}\ x=0\ \textbf{then}\ \text{1}\ \textbf{else}\ f(x-1)

\text{REQUIRES}\ n \geq 0
\text{ENSURES}\ f(n) = 1

\textbullet\ To prove:

\text{For all}\ n:\text{int}\ \text{such that}\ n \geq 0,\ f(n) = 1

\text{type} \quad \text{REQUIRES} \quad \text{ENSURES}

\text{(same as: for all}\ n \geq 0,\ \text{for all}\ e\ \text{such that}\ e=n,\ f(e) = 1)
proof (part 1)

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

Let P(n) be “f(n) = 1”

Theorem: ∀n≥0. P(n)

Proof: By simple induction on n.

• **Base**: we prove P(0). Here’s a proof:

  f 0 = (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

So f(0) = 1. That’s P(0).
proof (part 2)

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

• Inductive step:
  Let \( k \geq 0 \) and assume \( P(k) \), \( f \ k = 1 \).
  We prove \( P(k+1) \), \( f(k+1) = 1 \).

• Let \( v \) be the value of \( k+1 \), so \( v = k+1 \).
  \[
  f(k+1) = (\text{fn } x => \text{if } x=0 \text{ then } 1 \text{ else } f(x-1))(k+1)
  \]
  \[
  = (\text{fn } x => \text{if } x=0 \text{ then } 1 \text{ else } f(x-1))(v)
  \]
  \[
  = \text{if } v=0 \text{ then } 1 \text{ else } f(v-1)
  \]
  \[
  = \text{if false then } 1 \text{ else } f(v-1)
  \]
  \[
  = f(v-1)
  \]
  \[
  = f(k) \quad \text{since } v=k+1
  \]
  \[
  = 1 \quad \text{by assumption } P(k)
  \]

So \( P(k+1) \) holds.
Notes

• State the induction hypothesis clearly

• Use induction hypothesis only when justified

• Use equations and rules only when justified

• Use math and logic accurately

• Give explanation for non-trivial steps
Is this a proof of \( f 0 = 1 \)?

\[
\begin{align*}
f 0 &= 1 \\
(fn \ x \Rightarrow \text{if } x=0 \text{ then } 1 \text{ else } f(x-1)) \ 0 &= 1 \\
\text{if } 0=0 \text{ then } 1 \text{ else } f(0-1) &= 1 \\
\text{if true then } 1 \text{ else } f(0-1) &= 1 \\
1 &= 1 \\
\text{true}
\end{align*}
\]

No, it’s a proof of “if \( f 0 = 1 \) then true”

- A proof is a sequence of steps.
- Each step must follow logically from math facts or the results of earlier steps.
obviously wrong

\[
\begin{align*}
2 &= 1 \\
1 &= 2 & \text{by symmetry} \\
2+1 &= 1+2 & \text{by adding} \\
3 &= 3 & \text{by arithmetic} \\
\text{true}
\end{align*}
\]

Is this a proof that \(2 = 1\)?
Every non-empty set of dinosaurs has the same color.

**Theorem**

All dinosaurs are the same colour!

Base case: any one dinosaur is the same colour as itself.

**Induction step:**

Assume that any group of n dinosaurs is the same colour. Consider a group of n+1 dinosaurs. The first n (dino 1 to n) are all the same colour.

And the LAST n (dino 2 to n+1) must all be the same colour! So all n+1 are the same colour.

And by induction, all dinosaurs are the same colour!

No, WAIT! There is a lesson here!

Holy cow! Math is BROKEN.

Hey guys! All dinosaurs are the SAME COLOUR!
This proof is wrong

- The *inductive step* is inaccurate

\[ P(2) \text{ does not follow from } P(1) \]
Comments

• The spec and proof for $\forall n \geq 0. f(n) = 1$ used *equational* reasoning

• We could have worked with *evaluational* reasoning, but the details would be different
Remarks

• In *equational* reasoning we don’t always have to mimic *evaluation* order

• Sometimes we can do *parallel* analysis steps that don’t reflect actual evaluation of code

• This may yield a shorter proof

```haskell
fun f(x:int):int = if x=0 then 1 else f(x-1) + f(x-1)
```

For all \( n: \text{int} \) such that \( n \geq 0 \), \( f(n) = 2^n \)

For all \( n: \text{int} \) such that \( n \geq 0 \), \( f(n) \Rightarrow^* 2^n \)
Using simple induction

• Q: When can I use *simple* induction to prove a property of a recursive function $f$?
• A: When we can find a *non-negative* measure of *argument size* and show that if $f(x)$ calls $f(y)$ then $\text{size}(y) = \text{size}(x) - 1$

pick a notion of size appropriate for $f$
fun fact (x : int) : int = if x=0 then 1 else x * fact(x-1)

fun sum (L : int list) : int =
  case L of
  [ ] => 0
  | (x::R) => x + sum R

Which of these can be proven by simple induction?

For all \( n \geq 0 \), \( \text{fact } n \) evaluates to an integer value

\( \text{fact is total} \quad \text{For all } n \geq 0, \text{fact } n > n \)

\( \text{sum is total} \quad \text{For all } n > 2, \text{fact } n > n \)
Example

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

(The length of the argument list decreases in the recursive call)

To prove:

For all values L:int list there is an integer n such that
eval L =>* n
Exercise

• Prove the specification for eval

• It’s easy using simple induction on the length of the argument list

(this proof shows that eval : int list -> int is a total function)
Life’s not always simple

You cannot use **simple** induction on n for

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

Why not?
Strong induction

• To prove a property of the form

\[ P(n) \], for all non-negative integers \( n \)

Show that, for all \( k \geq 0 \),

\[ P(k) \] follows logically from \( P(0), \ldots, P(k-1) \).

**inductive step**

you can use any, all, or none to establish \( P(k) \)
Why this works

- P(0) gets a direct proof
- P(0) implies P(1)
- P(0), P(1) imply P(2)
- P(0), P(1), P(2) imply P(3)

For each \( k \geq 0 \) we can establish P(k) with \( k \) uses of step
Using strong induction

• Q: When can I use strong induction to prove a property of a recursive function $f$?

• A: When we can find a non-negative measure of argument size and show that if $f(x)$ calls $f(y)$ then $\text{size}(y) < \text{size}(x)$
Notes

• Sometimes, even for *simple* induction, it’s convenient to handle several “base” cases at the same time

• A proof using *strong* induction may not need a separate “base” case analysis
  • can sometimes handle *all* possible arguments in the “inductive step”
Example

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

( when n≥10, we get 0 ≤ n div 10 < n,
  so the argument value decreases
  in the recursive call )

To prove:

For all values n≥0,
  eval(decimal n) = n
Proof by strong induction

• For $0 \leq n < 10$, show directly that
  $\text{eval(decimal n)} = n$

• For $n \geq 10$, assume that
  For each $m$ such that $0 \leq m < n$,
    $\text{eval(decimal m)} = m$
  Then show that
  $\text{eval(decimal n)} = n$

multiple base cases handled together

use inductive analysis
for cases that make a recursive call
**Reminder**

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

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

For all values $n \geq 0$,

$$\text{eval(decimal n)} = n$$

Proof: by strong induction on $n$
Proof sketch
(the base cases)

• For $0 \leq n < 10$ we have

$$\text{eval(decimal n)}$$
$$= \text{eval [n]}$$
$$= n$$

(that was easy!)
Proof sketch  
(the inductive part)

• For \( n \geq 10 \) let \( r = n \mod 10 \), \( q = n \div 10 \).

\[
\text{eval(decimal n)} = \text{eval } ((n \mod 10) :: \text{decimal}(n \div 10)) = \text{eval } (r :: \text{decimal } q)
\]

• Since \( 0 \leq q < n \) it follows from IH that \( \text{eval(decimal } q) = q \)

• Hence there is a list value \( Q \) such that \( \text{decimal } q = Q \) and \( \text{eval } Q = q \)

So \( \text{eval } (r :: \text{decimal } q) = \text{eval } (r::Q) = r + 10 * \text{eval } Q = r + 10 * q = n \)

This shows that \( \text{eval(decimal } n) = n \)
Proof sketch

(Conclusion)

Let \( P(n) \) be “eval(decimal n) = n”

- The base analysis shows \( P(0), P(1), \ldots, P(9) \)

- The inductive analysis shows that for \( n \geq 10 \), \( P(n) \) follows from \( \{P(0), \ldots P(n-1)\} \)

- Hence, for all \( n \geq 0 \), \( P(n) \) holds
Notes

- We used equational reasoning to show that for all values $n \geq 0$, eval(decimal $n$) = $n$
- It follows that for all expressions $e: \text{int}$, if $e \Rightarrow^* n$ and $n \geq 0$, then eval(decimal $e$) $\Rightarrow^* n$
- It’s also possible to use evaluational reasoning to prove this result, inductively.
So far

• Simple and strong induction
• Examples of their use
• Just the beginning…

Next

• Another example
• What would you do?
fun log(x:int):int = 
    if x=1 then 0 else 1 + log(x div 2)

log : int -> int
REQUIRES n > 0

ENSURES log n evaluates to an integer k such that $2^k \leq n < 2^{k+1}$
Exercise

• Show that for each integer $n > 0$, there is a unique integer $k$ such that $2^k \leq n < 2^{k+1}$

  • this $k$ is called the logarithm (base 2) of $n$

• Prove the spec for $\log$

This shows that $\log$ computes logarithms (base 2)