15-150

Principles of Functional Programming

Lecture 3

January 20, 2026

Michael Erdmann

Recursion & Induction

Standard
Strong
Structural

```
(* square : int -> int
   REQUIRES: true
   ENSURES: square(n) ==> n^2
*)

fun square (n:int):int = n*n
```

square is bound to a function value.

square : int $\rightarrow$ int

"has type"

square 7 is an expression.

square 7 : int

square 7 $\xhookrightarrow{\quad}$ 49

"evaluates to"

49 : int

49 is a value (values are also expressions)

I sometimes abbreviate reductions

Instead of

square 7

$\Rightarrow$ [env when square was defined]
(fn (n:int) $\Rightarrow$ n*n) 7

$\Rightarrow$ [env...][7/n] n*n

$\Rightarrow$ [env...][7/n] 7*n

$\Rightarrow$ [env...][7/n] 7*7

$\Rightarrow$ 49

I may just write

square 7

$\Rightarrow$ 7*7

$\Rightarrow$ 49

I sometimes abbreviate reductions

Instead of

$$
\begin{aligned}
& \text{square } 7 \\
\Longrightarrow\ & [\text{env when square was defined}] \\
& (\text{fn } (n : \text{int}) \Rightarrow n * n)\ 7 \\
\Longrightarrow\ & [\text{env}\ldots][7/n]\ n * n \\
\Longrightarrow\ & [\text{env}\ldots][7/n]\ 7 * n \\
\Longrightarrow\ & [\text{env}\ldots][7/n]\ 7 * 7 \\
\Longrightarrow\ & 49
\end{aligned}
$$

Or even just

$$
\begin{aligned}
& \text{square } 7 \\
\Longrightarrow\ & 49
\end{aligned}
$$

```
(* power : int * int -> int
   REQUIRES : k ≥ 0
   ENSURES : power(n, k) ↪ n^k
              (let's define 0^0 = 1)
*)
```

```
fun power (n:int, 0:int) : int = 1
  | power (n, k) = n * power(n, k-1)
```

```
(* power : int * int -> int
   REQUIRES: k ≥ 0
   ENSURES: power(n, k) ↪ n^k
            (let's define 0^0 = 1)
*)
```

```
fun power (_ : int, 0 : int) : int = 1
  | power (n, k) = n * power(n, k-1)
```

```
(* power : int * int -> int
   REQUIRES : k ≥ 0
   ENSURES : power(n, k) ↪ n^k
             (let's define 0^0 = 1)
*)
```

```
fun power(n:int, 0:int): int = 1
  | power(n, k) = n * power(n, k-1)
```

$3^7 = 3 \cdot 3 \cdot 3 \cdot 3 \cdot 3 \cdot 3 \cdot 3 \cdot 1$

$= 2187$

O(k) recursive calls

```
(* even : int → bool
   REQUIRES: true
   ENSURES: even k evaluates to
              { true, if k is even;
              { false, if k is odd.
*)

fun even (k:int) : bool =
    (k mod 2) = 0
```

Typing & Evaluation of if…then…else.

- if e_1 then e_2 else e_3 : t

  if e_1 : bool,
  e_2 : t,
  & e_3 : t.

  (In particular, e_2 & e_3 must have the same type.)

- Evalution is left-to-right.

  e_2 is evaluated iff $e_1 \hookrightarrow$ true.

  e_3 is evaluated iff $e_1 \hookrightarrow$ false.

```
(* power : int * int -> int
   REQUIRES: k ≥ 0
   ENSURES: power(n, k) ↪ n^k
            (let's define 0^0 = 1)
*)

fun power(n:int, 0:int):int = 1
  | power (n, k) =
      if (even k)
      then square(power(n, k div 2))
      else n * power(n, k-1)
```

```
(* power : int * int -> int
   REQUIRES: k ≥ 0
   ENSURES: power(n, k) ↪ n^k
            (let's define 0^0 = 1)
*)

fun power(n:int, 0:int):int = 1
  | power (n, k) =
      if (even k)
      then square(power(n, k div 2))
      else n * power(n, k-1)
```

$$3^7 = 3\cdot(3\cdot(3\cdot 1)^2)^2$$

$$= 2187$$

$O(\log k)$ recursive calls

We would like to prove correctness for each implementation:

Theorem

For all values n: int & k: int, with $k \geq 0$, power$(n,k) \hookrightarrow n^k$.

During lecture:

We proved the theorem for our two implementations of `power`.

We used standard mathematical induction
for the first implementation and
strong induction for the second implementation.

Lists

Type t list for any type t.

Values $[v_1, \ldots, v_n]$, with each v_i a value of type t, & $n \geq 0$. (By $n=0$ we mean the empty list, written [ ] or nil.)

Expressions • All the values, &
• e::es, with $e : t$ & $es : t$ list.

For example 1::[2,3] which gives the list [1,2,3] : int list

Small comment

1::[2,3] & [1,2,3]

are simply two different ways of writing the same thing (same list value). Here is another way: 1::2::3::nil.

:: is right-associative.

So

$$1 :: 2 :: 3 :: \text{rest}$$

means

$$1 :: (2 :: (3 :: \text{rest})).$$

Type-checking

- [] : t list
- e::es : t list

  if e : t and es : t list

Evaluation

- $[\,]$ is a value (pronounced "nil")
  (same as nil)

- $e :: es \Rightarrow e' :: es$
  if $e \Rightarrow e'$

- $v :: es \Rightarrow v :: es'$
  if v is a value
  & $es \Rightarrow es'$.

(I.e., left-to-right evaluation for sequential evaluation.)

Can use list structure as patterns, with variables binding to different parts of the list.

During lecture:

We wrote a **`length`** function for lists.

We proved that **`length`** is total
We used *structural induction* for the proof.

Correspondence

Datastructure	Code	Proof
base case(s)	base case(s)	base case(s)
inductive/recursive definition(s)	recursive clause(s)	induction step(s)

Correspondence

Datastructure	Code	Proof
base case(s)	base case(s)	base case(s)
0	fun power(_, 0) = 1	power(n, 0) $\hookrightarrow$ 1
inductive/recursive definition(s)	recursive clause(s)	induction step(s)
(k-1) + 1	| power(n, k) = n * power(n, k-1)	

IH: power(n, k) $\hookrightarrow n^k$
NTS: power(n, k+1) $\hookrightarrow n^{k+1}$

Correspondence

Datastructure	Code	Proof
base case(s)	base case(s)	base case(s)
[ ]	0	length [ ] ⟿ 0
inductive/recursive definition(s)	recursive clause(s)	inductive case(s)
x::xs	1 + length(xs)	IH: length(xs) ⟿ v NTS: length(x::xs) ⟿ v'

Correspondence

Datastructure	Code	Proof
base case(s)	base case(s)	base case(s)
[ ]	0	length [ ] $\hookrightarrow$ 0
inductive/recursive definition(s)	recursive clause(s)	inductive case(s)
x::xs	1 + length(xs)	IH: length(xs) $\hookrightarrow$ v NTS: length(x::xs) $\hookrightarrow$ v'

There may be several base cases and/or several inductive cases.

That is all.

Have a good

Wednesday!