

**Inductive proofs over lists**

To prove  $P(xs)$  for any **finite**  $xs :: [a]$ :

**Case**  $P([])$ : Prove that  $P([])$  holds unconditionally.

**Case**  $P(x:s) \Rightarrow P(x:xs)$ : Prove that  
**if**  $P(xs)$  holds for some  $xs$   
**then**  $P(x:xs)$  holds.

Example:  $P(xs)$ :  $\text{len}(xs++ys) = \text{len } xs + \text{len } ys$

**Case**  $P([])$ :  $\text{len}([]++ys) = \text{len} [] + \text{len } ys$

$$\begin{aligned} \text{len}([]++ys) &= \text{len} [] + \text{len } ys \\ &= \{ ++.1, \text{len}.1 \} \\ \text{len } ys &= 0 + \text{len } ys \\ &= \{ \text{arithmetic, reflexivity of } = \} \\ &\text{True} \end{aligned}$$

**Case**  $P(xs) \Rightarrow P(x:xs)$ :

$$\begin{aligned} \text{len}(x:xs++ys) &= \text{len}(x:xs) + \text{len } ys \\ \text{len}(x:xs++ys) &= \{ ++.2 \} \\ \text{len}(x:(xs++ys)) &= \{ \text{len}.2 \rightarrow \} \\ 1 + \text{len}(xs++ys) &= \{ \text{induction hypothesis} \} \\ 1 + (\text{len } xs + \text{len } ys) &= \{ + \text{ associative} \} \\ (1 + \text{len } xs) + \text{len } ys &= \{ \text{len}.2 \leftarrow \} \\ \text{len}(x:xs) + \text{len } ys &\quad \blacksquare \end{aligned}$$

Example: Proof that  $(++)$  is associative— i.e., that for all lists  $xs, ys,$  and  $zs,$

$$P(xs): \quad xs ++ (ys++zs) = (xs++ys) ++ zs$$

The proof, by induction on  $xs$ :

**Case**  $P([])$ :  $[] ++ (ys++zs) = ([] ++ ys) ++ zs$

$$\begin{aligned} [] ++ (ys++zs) &= ([] ++ ys) ++ zs \\ &= \{ ++.1, \text{twice} \} \\ ys ++ zs &= ys ++ zs \\ &= \{ \text{reflexivity of } = \} \\ &\text{True} \end{aligned}$$

**Case**  $P(xs) \Rightarrow P(x:xs)$ :

$$\begin{aligned} (x:xs) ++ (ys++zs) &= ((x:xs)++ys) ++ zs \\ (x:xs) ++ (ys++zs) &= \{ ++.2 \rightarrow \} \\ x : (xs ++ (ys++zs)) &= \{ \text{induction hypothesis} \} \\ x : ((xs++ys) ++ zs) &= \{ ++.2 \leftarrow \} \\ (x:(xs++ys)) ++ zs &= \{ ++.2 \leftarrow \text{ again} \} \\ ((x:xs)++ys) ++ zs &\quad \blacksquare \end{aligned}$$

Example: *Double* induction on  $\text{Int}$  and  $[a]$

Given definitions

```
> take, drop :: Int -> [a] -> [a]
> take 0 _      = []
> take _ []     = []
> take k (t:ts) | k>0 = t : take (k-1) ts

> drop 0 vs     = vs
> drop _ []     = []
> drop k (_:vs) | k>0 = drop (k-1) vs
```

prove

$P(n,xs)$ :  $\text{take } n \text{ } xs ++ \text{drop } n \text{ } xs = xs$

Proof: by induction on  $n$  and  $xs$

**Case**  $P(0,xs)$ :  $\text{take } 0 \text{ } xs ++ \text{drop } 0 \text{ } xs = xs$

$$\begin{aligned} \text{take } 0 \text{ } xs ++ \text{drop } 0 \text{ } xs &= \{ \text{take}.1, \text{drop}.1 \} \\ [] ++ xs &= \{ ++.1 \} \\ xs & \end{aligned}$$

**Case**  $P(n,[])$ :

$$\begin{aligned} \text{take } n \text{ } [] ++ \text{drop } n \text{ } [] &= [] \\ \text{take } n \text{ } [] ++ \text{drop } n \text{ } [] &= \{ \text{take}.2, \text{drop}.2 \} \\ [] ++ [] &= \{ ++.1 \} \\ [] & \end{aligned}$$

**Case**  $P(n+1, x:xs)$ :

$$\begin{aligned} \text{take } (n+1) \text{ } (x:xs) ++ \text{drop } (n+1) \text{ } (x:xs) &= (x:xs) \\ \text{take } (n+1) \text{ } (x:xs) ++ \text{drop } (n+1) \text{ } (x:xs) &= \{ \text{take}.3, \text{drop}.3 \} \\ (x : \text{take } n \text{ } xs) ++ \text{drop } n \text{ } xs &= \{ ++.2 \rightarrow \} \\ x : (\text{take } n \text{ } xs ++ \text{drop } n \text{ } xs) &= \{ \text{induction hypothesis} \} \\ x : xs &\quad \blacksquare \end{aligned}$$

Points to note: In these proofs

- we have made heavy use of Haskell definitions *as equations*
- this approach works *only* under conditions of *referential transparency*

## Programming with Assignment

“Imperative” programming:

0. expressions +
1. **update** (= “assignment”) operation +
2. explicit **sequencing** (= time-ordering) of operations (a.k.a. “control flow”).

**program state**: central to all HL imperative languages.

In a true expression:

a variable’s value is *constant* throughout its scope.

but in an imperative program:

a variable can be bound to *different values* at different times (which correspond to different points in its scope).

**State**: the set of all of a program’s variable-value bindings; commonly expressed as a **function**:

$$state :: name \rightarrow value$$

[This definition is a *first approximation*, soon to be refined.]

Imperative processing’s three (idealized) stages:

0. **input**: create  $state_0$  from external data
1. **computation**: transform

$$state_0 \rightarrow state_1 \rightarrow state_2 \rightarrow \dots \rightarrow state_i \rightarrow \dots$$

until a desired condition on states is satisfied by some  $state_n$

2. **output**: extract **results** from  $state_n$

What users (and programmers) want is

**results**, not **states**

but imperative languages force us to deal with states.

So why do we continue to program imperatively?

- **habit** and **history**: Imperative HLLs were the first to grow out of machine-oriented programming. The outlines of the underlying machine —memory, program counter, i/o files— are still visible.
- **efficiency**: The closer correspondence between the imperative abstract machine and actual hardware makes efficient implementations easier.

## The Update Operation (imperative programming’s hallmark)

The operation

$$x := E \quad \text{assignment or update}$$

**binds** the value of  $E$  to variable  $x$  — **replacing**  $x$ ’s former value and thereby changing the program’s state (⚠ *update*  $\neq$  *initialize*).

Every variable has **two kinds of values**: **R** and **L**.

In the statement

$$x := y + 2$$

- what matters about  $y$  is its **R-value**
- what matters about  $x$  is its **L-value**.

**R-values** (right-hand-side values):

- data objects
- **contents** of storage locations

**L-values** (left-hand-side values)

- pointers
- **addresses** of storage locations

**Consequently**, the mapping from variable-names to values actually has *two parts*:

0. **environment**  $:: name \rightarrow location$
1. **state**  $:: location \rightarrow R\text{-value}$

## Control Flow

Consider two statements

$$x := 2 \quad \text{and} \quad x := 3.$$

The value bound to  $x$  after both statements have been executed depends on which of the two statements is executed **last**.

$\therefore$  Imperative programs must specify the time ordering of update operations *explicitly*.

A **compound expression** specifies the **functional dependencies** between its parts (and the (partial) time ordering of their operations follows directly).

Example:

$$(2+3) * (4+5)$$

The (+)’s must both precede the (\*), but they can be performed in either order (or even concurrently) with no effect on the expression’s value.

A **compound assignment statement** specifies the **time ordering** of its parts (but their functional dependencies do *not* follow directly).

Example:

$$x := 2; \quad x := x + 3; \quad y := 4; \quad y := y + 5; \quad x := x * y$$

Which statements can be performed concurrently?

What permutations of the statements yield the same final state?