Example: proof by cases

\[
\begin{align*}
\text{rem } x \; \text{[]} &= \; \text{[]} \\
\text{rem } x \; (y : ys) &| x = y = \text{rem } x \; ys \\
&| \text{otherwise} = y : \text{rem } x \; ys
\end{align*}
\]

**Lemma** \(\text{rem } z \; (xs ++ ys) = \text{rem } z \; xs ++ \text{rem } z \; ys\)

**Proof** by structural induction on \(xs\)
Example: proof by cases

\[
\text{rem } x \ [ ] = [] \\
\text{rem } x \ (y:ys) \ | \ x=y = \text{rem } x \ ys \\
\quad | \ \text{otherwise} = y : \text{rem } x \ ys
\]

**Lemma** \( \text{rem } z \ (xs ++ ys) = \text{rem } z \ xs ++ \text{rem } z \ ys \)

**Proof** by structural induction on \( xs \)

**Base case:**

\[
\text{rem } x \ [ ] = [] \\
\text{rem } x \ (y:ys) \ | \ x=y = \text{rem } x \ ys \\
\quad | \ \text{otherwise} = y : \text{rem } x \ ys
\]

**Induction step:**

To show: \( \text{rem } z \ ((x:xs)++ys) = \text{rem } z \ (x:xs) ++ \text{rem } z \ ys \)

**Proof by cases:**

**Case** \( z = x: \)

\[
\text{rem } z \ ((x:xs) ++ ys) \\
= \text{rem } z \ (xs ++ ys) \quad \text{-- by def of ++ and rem}
\]

**Induction step:**

To show: \( \text{rem } z \ ((x:xs)++ys) = \text{rem } z \ (x:xs) ++ \text{rem } z \ ys \)

**Proof by cases:**

**Case** \( z = x: \)

\[
\text{rem } z \ ((x:xs) ++ ys) \\
= \text{rem } z \ (xs ++ ys) \quad \text{-- by def of ++ and rem}
\]

\[
= \text{rem } z \ xs ++ \text{rem } z \ ys \quad \text{-- by IH}
\]

---

**Lemma** \( \text{rem } z \ (xs ++ ys) = \text{rem } z \ xs ++ \text{rem } z \ ys \)

**Proof** by structural induction on \( xs \)

**Base case:**

\[
\text{rem } x \ [ ] = [] \\
\text{rem } x \ (y:ys) \ | \ x=y = \text{rem } x \ ys \\
\quad | \ \text{otherwise} = y : \text{rem } x \ ys
\]

**Induction step:**

To show: \( \text{rem } z \ ((x:xs)++ys) = \text{rem } z \ (x:xs) ++ \text{rem } z \ ys \)

**Proof by cases:**

**Case** \( z = x: \)

\[
\text{rem } z \ ((x:xs) ++ ys) \\
= \text{rem } z \ (xs ++ ys) \quad \text{-- by def of ++ and rem}
\]

\[
= \text{rem } z \ xs ++ \text{rem } z \ ys \quad \text{-- by IH}
\]

---

**Lemma** \( \text{rem } z \ (xs ++ ys) = \text{rem } z \ xs ++ \text{rem } z \ ys \)

**Proof** by structural induction on \( xs \)

**Base case:**

\[
\text{rem } x \ [ ] = [] \\
\text{rem } x \ (y:ys) \ | \ x=y = \text{rem } x \ ys \\
\quad | \ \text{otherwise} = y : \text{rem } x \ ys
\]

**Induction step:**

To show: \( \text{rem } z \ ((x:xs)++ys) = \text{rem } z \ (x:xs) ++ \text{rem } z \ ys \)

**Proof by cases:**

**Case** \( z = x: \)

\[
\text{rem } z \ ((x:xs) ++ ys) \\
= \text{rem } z \ (xs ++ ys) \quad \text{-- by def of ++ and rem}
\]

\[
= \text{rem } z \ xs ++ \text{rem } z \ ys \quad \text{-- by IH}
\]
The given text describes two induction steps. The first induction step is:

**Induction step:**
To show: \( \text{rem } z ((x:xs)++ys) = \text{rem } z (x:xs) ++ \text{rem } z ys \)

**Proof by cases:**

Case \( z = x \):
- \( \text{rem } z ((x:xs) ++ ys) = \text{rem } z (xs ++ ys) \) -- by def of ++ and rem
- \( = \text{rem } z xs ++ \text{rem } z ys \) -- by IH
- \( \text{rem } z (xs:xs) ++ \text{rem } z ys \) -- by def of rem
- \( = \text{rem } z xs ++ \text{rem } z ys \) -- by def of rem

Case \( z \neq x \):
- \( \text{rem } z ((x:xs) ++ ys) = x : \text{rem } z (xs ++ ys) \) -- by def of ++ and rem
- \( = x : \text{rem } z xs ++ \text{rem } z ys \) -- by IH

The second induction step is similar, but with the definition of \( z \) reversed.
Proof by cases

Works just as well for if-then-else, for example

\[
\begin{align*}
\text{rem } x &= [] \\
\text{rem } x (y:ys) &= \begin{cases} 
\text{if } x == y \text{ then } \text{rem } x ys \\
\text{else } y : \text{rem } x ys
\end{cases}
\end{align*}
\]

Inefficiency of reverse

reverse [1,2,3]

= reverse [2,3] ++ [1]
= (reverse [3] ++ [2]) ++ [1]
= ((reverse [] ++ [3]) ++ [2]) ++ [1]

Inefficiency of reverse

reverse [1,2,3]

= reverse [2,3] ++ [1]
= (reverse [3] ++ [2]) ++ [1]
= ((([] ++ [3]) ++ [2]) ++ [1]
Inefficiency of reverse

reverse [1,2,3]
= reverse [2,3] ++ [1]
= (reverse [3] ++ [2]) ++ [1]
= (reverse [] ++ [3]) ++ [2] ++ [1]
= (reverse [] ++ [3]) ++ [2] ++ [1]
= ([3] ++ [2]) ++ [1]

An improvement: itrev

itrev :: [a] -> [a] -> [a]
An improvement: itrev

\[
\text{itrev} :: [a] \to [a] \to [a] \\
\text{itrev} \ [] \ xs \quad = \ xs \\
\text{itrev} \ (x:xs) \ ys \ = \ \text{itrev} \ xs \ (x:ys)
\]

\[
\begin{align*}
\text{itrev} \ [1,2,3] \ [] & = \text{itrev} \ [2,3] \ [1] \\
& = \text{itrev} \ [3] \ [2,1] \\
& = \text{itrev} \ [] \ [3,2,1] \\
& = [3,2,1]
\end{align*}
\]
Lemma \textit{itrev} \(xs\) \([\,]\) = \textit{reverse} \(xs\)

Proof by structural induction on \(xs\)
Induction step fails:
To show: \(\textit{itrev} \,(x\,::{\,}xs)\) \([\,]\) = \textit{reverse} \,(x\,::{\,}xs)\)
\[= \textit{itrev} \,xs \,[x]\] -- by def of \textit{itrev}
**Proof attempt**

**Lemma** \( \text{itrev} \; xs \; [] = \text{reverse} \; xs \)

**Proof** by structural induction on \( xs \)

Induction step fails:

To show: \( \text{itrev} \; (x:xs) \; [] = \text{reverse} \; xs \)

\( \text{itrev} \; (x:xs) \; [] \)

\( = \text{itrev} \; xs \; [x] \quad \text{-- by def of itrev} \)

\( \text{reverse} \; (x:xs) \)

\( = \text{reverse} \; xs \; ++ \; [x] \quad \text{-- by def of reverse} \)

Problem: IH not applicable because too specialized: \( [] \)

---

**Generalization**

**Lemma** \( \text{itrev} \; xs \; ys = \text{reverse} \; xs \; ++ \; ys \)

**Proof** by structural induction on \( xs \)

Induction step:

To show: \( \text{itrev} \; (x:xs) \; ys = \text{reverse} \; (x:xs) \; ++ \; ys \)

\( \text{itrev} \; (x:xs) \; ys \)

\( = \text{itrev} \; xs \; (x:ys) \quad \text{-- by def of itrev} \)
**Lemma** \( \text{itrev } xs \ ys = \text{reverse } xs ++ ys \)

**Proof** by structural induction on \( xs \)

Induction step:
To show: \( \text{itrev } (x:xs) \ ys = \text{reverse } (x:xs) ++ ys \)
\[ \text{itrev } (x:xs) \ ys \]
\[ = \text{itrev } xs (x:ys) \quad \text{-- by def of itrev} \]
\[ = \text{reverse } xs ++ (x:ys) \quad \text{-- by IH} \]

\[ \text{reverse } (x:xs) ++ ys \]
\[ = (\text{reverse } xs ++ [x]) ++ ys \quad \text{-- by def of reverse} \]

**Lemma** \( \text{itrev } xs \ ys = \text{reverse } xs ++ ys \)

**Proof** by structural induction on \( xs \)

Induction step:
To show: \( \text{itrev } (x:xs) \ ys = \text{reverse } (x:xs) ++ ys \)
\[ \text{itrev } (x:xs) \ ys \]
\[ = \text{itrev } xs (x:ys) \quad \text{-- by def of itrev} \]
\[ = \text{reverse } xs ++ (x:ys) \quad \text{-- by IH} \]
\[ \text{reverse } (x:xs) ++ ys \]
\[ = (\text{reverse } xs ++ [x]) ++ ys \quad \text{-- by def of reverse} \]
\[ = \text{reverse } xs ++ ([x] ++ ys) \quad \text{-- by Lemma app.assoc} \]
\[ = \text{reverse } xs ++ (x:ys) \quad \text{-- by def of ++} \]
Lemma: \( \text{itrev} \; xs \; ys = \text{reverse} \; (x:xs) \; ++ \; ys \)

Proof: by structural induction on \( xs \)

Induction step:
To show: \( \text{itrev} \; (x:xs) \; ys = \text{reverse} \; (x:xs) \; ++ \; ys \)
\[
\begin{align*}
\text{itrev} \; (x:xs) \; ys \\
= \text{itrev} \; x : (ys) & \quad \text{--- by def of \text{itrev}} \\
= \text{reverse} \; x : (ys) & \quad \text{--- by \text{IH}} \\
= \text{reverse} \; (x:xs) \; ++ \; ys & \quad \text{--- by def of \text{reverse}} \\
= \text{reverse} \; (x:xs) \; ++ \; ([x] \; ++ \; ys) & \quad \text{--- by Lemma \text{app.assoc}} \\
= \text{reverse} \; (x:xs) \; ++ \; (x:ys) & \quad \text{--- by def of \text{++}} \\
\end{align*}
\]

Note: \( \text{IH} \) is used with \( x : ys \) instead of \( ys \)

When using the \( \text{IH} \), variables may be replaced by arbitrary expressions, only the induction variable must stay fixed.

Justification: all variables are implicitly \( \forall \)-quantified, except for the induction variable.
When using the IH, variables may be replaced by arbitrary expressions, only the induction variable must stay fixed.

Induction on the length of a list

\[ \text{qsort} :: \text{Ord a} \Rightarrow [a] \to [a] \]

\[ \text{qsort} [] = [] \]

\[ \text{qsort} \ (x:xs) = \text{qsort below} ++ [x] ++ \text{qsort above} \]

\[ \quad \text{where} \ \text{below} = [y \mid y \leftarrow xs, y \leq x] \]

\[ \quad \text{above} = [z \mid y \leftarrow xs, x < z] \]\n
**Lemma** \( \text{qsort} \ xs \) is sorted
Induction on the length of a list

qsort :: Ord a => [a] -> [a]
qsort [] = []
qsort (x:xs) = qsort below ++ [x] ++ qsort above
  where below = [y | y <- xs, y <= x]
  above = [z | z <- xs, z < x]

Lemma qsort xs is sorted
Proof by induction on the length of the argument of qsort.

Induction step: In the call qsort (x:xs) we have length below <= length xs < length(x:xs) (also for above).

Lemma qsort xs is sorted
Proof by induction on the length of the argument of qsort.

Induction step: In the call qsort (x:xs) we have length below <= length xs < length(x:xs) (also for above).
Therefore qsort below and qsort above are sorted by IH.
Induction on the length of a list

qsort :: Ord a => [a] -> [a]
qsort [] = []
qsort (x:xs) = qsort below ++ [x] ++ qsort above
   where below = [y | y <- xs, y <= x]
        above = [z | z <- xs, x < z]

Lemma qsort xs is sorted
Proof by induction on the length of the argument of qsort.
Induction step: In the call qsort (x:xs) we have length below <= length xs < length(x:xs) (also for above).
Therefore qsort below and qsort above are sorted by IH.
By construction below contains only elements (<=x).
Therefore qsort below contains only elements (<=x) (proof!).
Induction on the length of a list

\[
\text{qsort} :: \text{Ord a} \Rightarrow [a] \rightarrow [a]
\]
\[
\text{qsort} [] = []
\]
\[
\text{qsort} (x:xs) = \text{qsort below} :: [] ++ [x] ++ \text{qsort above}
\]
\[
\text{where below} = [y \mid y \leftarrow xs, y \leq x]
\]
\[
\text{above} = [z \mid z \leftarrow xs, x < z]
\]

**Lemma** \(\text{qsort} \; x\!s\; \text{is sorted}\)

**Proof** by induction on the length of the argument of \(\text{qsort}\).

Induction step: In the call \(\text{qsort} \; (x:xs)\) we have \(\text{length below} \leq \text{length} \; x\!s \leq \text{length}(x:xs)\) (also for above).

Therefore \(\text{qsort} \; \text{below}\) and \(\text{qsort} \; \text{above}\) are sorted by IH.

By construction \(\text{below}\) contains only elements \((\leq x)\).

Therefore \(\text{qsort} \; \text{below}\) contains only elements \((\leq x)\) (proof!).

Analogously for \(\text{above}\) and \((x)<\).

Therefore \(\text{qsort} \; (x:xs)\) is sorted.

Is that all? Or should we prove something else about sorting?

How about this sorting function?

\[
\text{superquicksort} \; _{} = []
\]

Is that all? Or should we prove something else about sorting?

How about this sorting function?

\[
\text{superquicksort} \; _{} = []
\]

Every element should occur as often in the output as in the input!
5.2 Definedness
Simplifying assumption, implicit so far:

No undefined values

Two kinds of undefinedness:

head [] raises exception
f x = f x + 1 does not terminate

Two kinds of undefinedness:

head [] raises exception
f x = f x + 1 does not terminate

Undefiniteness can be handled, too.
5.2 Definedness
Simplifying assumption, implicit so far:

No undefined values

Two kinds of undefinedness:

head [] raises exception
if x = f x + 1 does not terminate

Undefinedness can be handled, too.
But it complicates life

What is the problem?

Many familiar laws no longer hold unconditionally:

\[ x - x = 0 \]

is true only if \( x \) is a defined value.

Two examples:
- Not true: head [] - head [] = 0

What is the problem?

Many familiar laws no longer hold unconditionally:

\[ x - x = 0 \]

is true only if \( x \) is a defined value.

Two examples:
- Not true: head [] - head [] = 0
- From the nonterminating definition
  if x = f x + 1
  we could conclude that 0 = 1.
Termination of a function means termination for all inputs.

Restriction:
The proof methods in this chapter assume that all recursive definitions under consideration terminate.

Most Haskell functions we have seen so far terminate.
**How to prove termination**

Example

reverse [] = []
reverse (x:xs) = reverse xs ++ [x]

terminates because ++ terminates and with each recursive call of reverse, the length of the argument becomes smaller.

Example

reverse [] = []
reverse (x:xs) = reverse xs ++ [x]

terminates because ++ terminates and with each recursive call of reverse, the length of the argument becomes smaller.

A function \( f : : \mathbb{T}_1 \rightarrow \mathbb{T} \) terminates if there is a *measure function* \( m : : \mathbb{T}_1 \rightarrow \mathbb{N} \) such that

- for every defining equation \( f \ p = t \)
- and for every recursive call \( f \ r \) in \( t \): \( m \ p > m \ r \).

**Example**

reverse [] = []
reverse (x:xs) = reverse xs ++ [x]

terminates because ++ terminates and with each recursive call of reverse, the length of the argument becomes smaller.

A function \( f : : \mathbb{T}_1 \rightarrow \mathbb{T} \) terminates if there is a *measure function* \( m : : \mathbb{T}_1 \rightarrow \mathbb{N} \) such that

- for every defining equation \( f \ p = t \)
- and for every recursive call \( f \ r \) in \( t \): \( m \ p > m \ r \).

**Note:**
- All primitive recursive functions terminate.
How to prove termination

Example
reverse [] = []
reverse (x:xs) = reverse xs ++ [x]
terminates because ++ terminates and with each recursive call of reverse, the length of the argument becomes smaller.

A function \( f : : T_1 \rightarrow T \) terminates if there is a measure function \( m : : T_1 \rightarrow \mathbb{N} \) such that
- for every defining equation \( f \ p = t \)
- and for every recursive call \( f \ r \) in \( t \): \( m \ p > m \ r \).

Note:
- All primitive recursive functions terminate.
- \( m \) can be defined in Haskell or mathematics.

More generally: \( f : : T_1 \rightarrow \ldots \rightarrow T_n \rightarrow T \) terminates if there is a measure function \( m : : T_1 \rightarrow \ldots \rightarrow T_n \rightarrow \mathbb{N} \) such that
  - for every defining equation \( f \ p_1 \ldots \ p_n = t \)
  - and for every recursive call \( f \ r_1 \ldots \ r_n \) in \( t \):
    \( m \ p_1 \ldots \ p_n > m \ r_1 \ldots \ r_n \).

How to prove termination

Example
reverse [] = []
reverse (x:xs) = reverse xs ++ [x]
terminates because ++ terminates and with each recursive call of reverse, the length of the argument becomes smaller.

A function \( f : : T_1 \rightarrow T \) terminates if there is a measure function \( m : : T_1 \rightarrow \mathbb{N} \) such that
- for every defining equation \( f \ p = t \)
- and for every recursive call \( f \ r \) in \( t \): \( m \ p > m \ r \).

Note:
- All primitive recursive functions terminate.
- \( m \) can be defined in Haskell or mathematics.
- The conditions above can be refined to take special Haskell features into account, e.g., sequential pattern matching.
More generally: $f :: T_1 \rightarrow \ldots \rightarrow T_n \rightarrow T$ terminates if there is a measure function $m :: T_1 \rightarrow \ldots \rightarrow T_n \rightarrow \mathbb{N}$ such that

- for every defining equation $f \ p_1 \ldots \ p_n = t$
- and for every recursive call $f \ r_1 \ldots \ r_n$ in $t$:
  
  $$m \ p_1 \ldots \ p_n > m \ r_1 \ldots \ r_n.$$

Of course, all other functions that are called by $f$ must also terminate.

Haskell allows infinite values, in particular infinite lists.

Example: [1, 1, 1, ...]

Infinite objects must be constructed by recursion:
ones = 1 : ones

Because we restrict to terminating definitions in this chapter, infinite values cannot arise.
Infinite values

Haskell allows infinite values, in particular infinite lists.
Example: \([1, 1, 1, \ldots]\)

Infinite objects must be constructed by recursion:
\[
\text{ones} = 1 : \text{ones}
\]

Because we restrict to terminating definitions in this chapter, infinite values cannot arise.

Note:
- By termination of functions we really mean termination on \textit{finite} values.
- For example \texttt{reverse} terminates only on finite lists.

---

Exceptions

If we use arithmetic equations like \(x - x = 0\) unconditionally, we can “lose” exceptions:

\[
\text{head } xs - \text{head } xs = 0
\]
Exceptions

If we use arithmetic equations like $x - x = 0$ unconditionally, we can “lose” exceptions:

$$\text{head } xs - \text{head } xs = 0$$

is only true if $xs /= []$

In such cases, we can prove equations $e1 = e2$ that are only partially correct:

Summary

- In this chapter everything must terminate
- This avoids undefined and infinite values
Summary

- In this chapter everything must terminate
- This avoids undefined and infinite values
- This simplifies proofs

6. Higher-Order Functions

Recall [Pic is short for Picture]

\[
\begin{align*}
\text{alterH} & : \text{Pic} \rightarrow \text{Pic} \rightarrow \text{Int} \rightarrow \text{Pic} \\
\text{alterH} \ p \ c \ i & = c \\
\text{alterH} \ p \ c \ i \ n & = \text{beside} \ p1 \ (\text{alterH} \ c \ i \ (n-1)) \\
\text{alterV} & : \text{Pic} \rightarrow \text{Pic} \rightarrow \text{Int} \rightarrow \text{Pic} \\
\text{alterV} \ p \ c & = c \\
\text{alterV} \ p \ c \ n & = \text{above} \ p1 \ (\text{alterV} \ c \ (n-1))
\end{align*}
\]

Very similar.
alterH :: Pic -> Pic -> Int -> Pic
alterH pic1 pic2 1 = pic1
alterH pic1 pic2 n = beside pic1 (alterH pic2 pic1 (n-1))

alterV :: Pic -> Pic -> Int -> Pic
alterV pic1 pic2 1 = pic1
alterV pic1 pic2 n = above pic1 (alterV pic2 pic1 (n-1))

Very similar. Can we avoid duplication?

alt f pic1 pic2 1 = pic1
alt f pic1 pic2 n = f pic1 (alt f pic2 pic1 (n-1))
Higher-order functions:
Functions that take functions as arguments
... -> (... -> ...) -> ...

Recall [Pic is short for Picture]
alterH :: Pic -> Pic -> Int -> Pic
alterH pic1 pic2 1 = pic1
alterH pic1 pic2 n = beside pic1 (alterH pic2 pic1 (n-1))

alterV :: Pic -> Pic -> Int -> Pic
alterV pic1 pic2 1 = pic1
alterV pic1 pic2 n = above pic1 (alterV pic2 pic1 (n-1))

Very similar. Can we avoid duplication?

alt :: (Pic -> Pic -> Pic) -> Pic -> Pic -> Int -> Pic
alt f pic1 pic2 1 = pic1
alt f pic1 pic2 n = f pic1 (alt f pic2 pic1 (n-1))

alterH pic1 pic2 n = alt beside pic1 pic2 n
alterV pic1 pic2 n = alt above pic1 pic2 n

Higher-order functions:
Functions that take functions as arguments
... -> (... -> ...) -> ...

Higher-order functions capture patterns of computation
6.1 Applying functions to all elements of a list: map

Example

map even [1, 2, 3]
= [False, True, False]

map toLower "R2-D2"
= "r2-d2"
6.1 Applying functions to all elements of a list: map

Example

map even [1, 2, 3]
= [False, True, False]

map toLower "R2-D2"
= "r2-d2"

map reverse ["abc", "123"]
= ["cba", "321"]

What is the type of map?

map :: (a -> b) -> [a] -> [b]

map: The mother of all higher-order functions

Predefined in Prelude.
map: The mother of all higher-order functions

Predefined in Prelude. Two possible definitions:

\[
\text{map } f \ \text{xs} = [ f \ x \mid x \ <- \ \text{xs} ]
\]

\[
\text{map } f \ [\ ] = [\ ]
\]

\[
\text{map } f \ (x:xs) = f \ x : \text{map } f \ xs
\]

Evaluating map

\[
\text{map } f \ [\ ] = [\ ]
\]

\[
\text{map } f \ (x:xs) = f \ x : \text{map } f \ xs
\]

\[
\text{map } \text{sqr} \ [1, \ -2]
\]
map f [] = []
map f (x:xs) = f x : map f xs

map sqr [1, -2]
= map sqr (1 : -2 : [])

map f [] = []
map f (x:xs) = f x : map f xs

map sqr [1, -2]
= map sqr (1 : -2 : [])
= sqr 1 : map sqr (-2 : [])

map f [] = []
map f (x:xs) = f x : map f xs

map sqr [1, -2]
= map sqr (1 : -2 : [])
= sqr 1 : map sqr (-2 : [])

length (map f xs) =
Some properties of map

\[
\text{length } (\text{map } f \text{ } \text{xs}) = \text{length } \text{xs}
\]

\[
\text{length } (\text{map } f \text{ } \text{xs}) = \text{length } \text{xs}
\]

\[
\text{map } f \text{ } (\text{xs } \text{++ } \text{ys}) = 
\]

\[
\text{length } (\text{map } f \text{ } \text{xs}) = \text{length } \text{xs}
\]

\[
\text{map } f \text{ } (\text{xs } \text{++ } \text{ys}) = \text{map } f \text{ } \text{xs } \text{++ } \text{map } f \text{ } \text{ys}
\]
Some properties of map

length (map f xs) = length xs
map f (xs ++ ys) = map f xs ++ map f ys
map f (reverse xs) = reversed (map f xs)

Proofs by induction

QuickCheck and function variables

QuickCheck does not work automatically for properties of function variables
QuickCheck and function variables

QuickCheck does not work automatically for properties of function variables

It needs to know how to generate and print functions.

Cheap alternative: replace function variable by specific function(s)

Some properties of map

\[ \text{length (map } f \text{ xs) = length xs} \]
\[ \text{map } f \text{ (xs ++ ys) = map } f \text{ xs ++ map } f \text{ ys} \]
\[ \text{map } f \text{ (reverse xs) = reverse (map } f \text{ xs)} \]

Proofs by induction

QuickCheck and function variables

QuickCheck does not work automatically for properties of function variables

It needs to know how to generate and print functions.

Cheap alternative: replace function variable by specific function(s)

Example

\[ \text{prop_map_even :: [Int] -> [Int] -> Bool} \]
\[ \text{prop_map_even xs ys = map even (xs ++ ys) = map even xs ++ map even ys} \]
6.2 Filtering a list: filter

Example

filter even [1, 2, 3] = [2]

filter isAlpha "R2-D2" = "RD"

filter null [[] , [1,2] , []] = [[] , []]
6.2 Filtering a list: filter

Example

filter even [1, 2, 3] = [2]
filter isAlpha "R2-D2" = "RD"
filter null [[]] = [[]]

What is the type of filter?

\[
\text{filter} :: (a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow [a]
\]

Predefined in Prelude.
Two possible definitions:

\[
\text{filter} \ p \ \text{xs} = [x \mid x \leftarrow \text{xs}, \ p \ x]
\]

filter p [] = []
filter p (x:xs) | p x = x : filter p xs |
\text{otherwise} = \text{filter} p \ \text{xs}
Some properties of `filter`

\[
\text{filter } p \ (x :: y :: xs) = \text{filter } p \ x :: \text{filter } p \ y :: \text{filter } p \ xs
\]

\[
\text{filter } p \ (\text{reverse } xs) = \text{reverse } (\text{filter } p \ xs)
\]

Proofs by induction