Script **generated by TTT**

Title: Nipkow: Info2 (12.11.2013)

Date: Tue Nov 12 15:32:23 CET 2013

Duration: 86:03 min

Pages: 136

ttt/TTT Tue 15:32 Tobias Nipkow

Example: proof by cases

```
rem x []  =  []
rem x (y:ys) | x==y       =  rem x ys
             | otherwise  =  y : rem x ys
```

Example: proof by cases

```
rem x []  =  []
rem x (y:ys) | x==y       =  rem x ys
             | otherwise  =  y : rem x ys
```

Lemma `rem z (xs ++ ys) = rem z xs ++ rem z ys`

Example: proof by cases

```
rem x []  =  []
rem x (y:ys) | x==y       =  rem x ys
             | otherwise  =  y : rem x ys
```

Lemma `rem z (xs ++ ys) = rem z xs ++ rem z ys`

Proof by structural induction on xs

Example: proof by cases

```
rem x []  =  []
rem x (y:ys) | x==y       =  rem x ys
             | otherwise  =  y : rem x ys
```

Lemma `rem z (xs ++ ys) = rem z xs ++ rem z ys`

Proof by structural induction on `xs`

Base case:

```
rem x []  =  []
rem x (y:ys) | x==y       =  rem x ys
             | otherwise  =  y : rem x ys
```

Induction step:

To show: `rem z ((x:xs)++ys) = rem z (x:xs) ++ rem z ys`

Proof by cases:

Case `z == x`:

```
rem x []  =  []
rem x (y:ys) | x==y       =  rem x ys
             | otherwise  =  y : rem x ys
```

Induction step:

To show: `rem z ((x:xs)++ys) = rem z (x:xs) ++ rem z ys`

Proof by cases:

Case `z == x`:

```
 rem z ((x:xs) ++ ys)
 = rem z (xs ++ ys)          -- by def of ++ and rem
```

```
rem x []  =  []
rem x (y:ys) | x==y       =  rem x ys
             | otherwise  =  y : rem x ys
```

Induction step:

To show: `rem z ((x:xs)++ys) = rem z (x:xs) ++ rem z ys`

Proof by cases:

Case `z == x`:

```
 rem z ((x:xs) ++ ys)
 = rem z (xs ++ ys)          -- by def of ++ and rem
 = rem z xs ++ rem z ys      -- by IH
```

```
rem x []     =  []
rem x (y:ys) | x==y        =  rem x ys
             | otherwise   =  y : rem x ys
```

Induction step:

To show: `rem z ((x:xs)++ys) = rem z (x:xs) ++ rem z ys`

Proof by cases:

Case `z == x`:

```
rem z ((x:xs) ++ ys)
= rem z (xs ++ ys)          -- by def of ++ and rem
= rem z xs ++ rem z ys      -- by IH
rem z (x:xs) ++ rem z ys
= rem z xs ++ rem z ys      -- by def of rem
```

```
rem x []     =  []
rem x (y:ys) | x==y        =  rem x ys
             | otherwise   =  y : rem x ys
```

Induction step:

To show: `rem z ((x:xs)++ys) = rem z (x:xs) ++ rem z ys`

Proof by cases:

Case `z == x`:

```
rem z ((x:xs) ++ ys)
= rem z (xs ++ ys)          -- by def of ++ and rem
= rem z xs ++ rem z ys      -- by IH
rem z (x:xs) ++ rem z ys
= rem z xs ++ rem z ys      -- by def of rem
```

Case `z /= x`:

```
rem z ((x:xs) ++ ys)
= x : rem z (xs ++ ys)          -- by def of ++ and rem
```

```
rem x []     =  []
rem x (y:ys) | x==y        =  rem x ys
             | otherwise   =  y : rem x ys
```

Induction step:

To show: `rem z ((x:xs)++ys) = rem z (x:xs) ++ rem z ys`

Proof by cases:

Case `z == x`:

```
rem z ((x:xs) ++ ys)
= rem z (xs ++ ys)          -- by def of ++ and rem
= rem z xs ++ rem z ys      -- by IH
rem z (x:xs) ++ rem z ys
= rem z xs ++ rem z ys      -- by def of rem
```

Case `z /= x`:

```
rem z ((x:xs) ++ ys)
= x : rem z (xs ++ ys)          -- by def of ++ and rem
= x : (rem z xs ++ rem z ys)    -- by IH
```

```
rem x []     =  []
rem x (y:ys) | x==y        =  rem x ys
             | otherwise   =  y : rem x ys
```

Induction step:

To show: `rem z ((x:xs)++ys) = rem z (x:xs) ++ rem z ys`

Proof by cases:

Case `z == x`:

```
rem z ((x:xs) ++ ys)
= rem z (xs ++ ys)          -- by def of ++ and rem
= rem z xs ++ rem z ys      -- by IH
rem z (x:xs) ++ rem z ys
= rem z xs ++ rem z ys      -- by def of rem
```

Case `z /= x`:

```
rem z ((x:xs) ++ ys)
= x : rem z (xs ++ ys)          -- by def of ++ and rem
= x : (rem z xs ++ rem z ys)    -- by IH
rem z (x:xs) ++ rem z ys
= x : (rem z xs ++ rem z ys)    -- by def of rem and ++
```

Proof by cases

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

```
rem x []     =  []
rem x (y:ys)  =  if x == y then rem x ys
                 else y : rem x ys
```

Inefficiency of `reverse`

```
reverse [1,2,3]
```

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]
= ((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]
= (([] ++ [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]
= (([] ++ [3]) ++ [2]) ++ [1]
= ([3] ++ [2]) ++ [1]
= (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]
= (([] ++ [3]) ++ [2]) ++ [1]
= ([3] ++ [2]) ++ [1]
= (3 : ([] ++ [2])) ++ [1]
= [3,2] ++ [1]
= 3 : ([2] ++ [1])
= 3 : (2 : ([] ++ [1]))
= [3,2,1]
```

An improvement: itrev

```
itrev :: [a] -> [a] -> [a]
```

An improvement: itrev

```
itrev :: [a] -> [a] -> [a]
itrev [] xs      = xs
```

An improvement: itrev

```
itrev :: [a] -> [a] -> [a]
itrev [] xs      = xs
itrev (x:xs) ys = itrev xs (x:ys)
```

An improvement: itrev

```
itrev :: [a] -> [a] -> [a]
itrev [] xs      = xs
itrev (x:xs) ys = itrev xs (x:ys)

itrev [1,2,3] []
= itrev [2,3] [1]
= itrev [3] [2,1]
= itrev [] [3,2,1]
```

An improvement: itrev

```
itrev :: [a] -> [a] -> [a]
itrev [] xs      = xs
itrev (x:xs) ys = itrev xs (x:ys)

itrev [1,2,3] []
= itrev [2,3] [1]
= itrev [3] [2,1]
= itrev [] [3,2,1]
= [3,2,1]
```

Proof attempt

Lemma `itrev xs [] = reverse xs`

Proof attempt

Lemma `itrev xs [] = reverse xs`

Proof by structural induction on `xs`

Induction step fails:

To show: `itrev (x:xs) [] = reverse xs`

Proof attempt

Lemma `itrev xs [] = reverse xs`

Proof by structural induction on `xs`

Induction step fails:

To show: `itrev (x:xs) [] = reverse xs`

```
itrev (x:xs) []
= itrev xs [x]        -- by def of itrev
```

Proof attempt

Lemma `itrev xs [] = reverse xs`

Proof by structural induction on `xs`

Induction step fails:

To show: `itrev (x:xs) [] = reverse xs`

```
itrev (x:xs) []
= itrev xs [x]        -- by def of itrev
reverse (x:xs)
= reverse xs ++ [x]   -- by def of reverse
```

Proof attempt

Lemma `itrev xs [] = reverse xs`

Proof by structural induction on `xs`

Induction step fails:

To show: `itrev (x:xs) [] = reverse xs`

```
itrev (x:xs) []
= itrev xs [x]        -- by def of itrev
reverse (x:xs)
= reverse xs ++ [x]   -- by def of reverse
```

Problem: IH not applicable because too specialized: `[]`

Generalization

Lemma `itrev xs ys =`

Generalization

Lemma `itrev xs ys = reverse xs ++ ys`

Generalization

Lemma `itrev xs ys = reverse xs ++ ys`

Proof by structural induction on `xs`

Induction step:

To show: `itrev (x:xs) ys = reverse (x:xs) ++ ys`

```
itrev (x:xs) ys
= itrev xs (x:ys)            -- by def of itrev
```

Generalization

Lemma `itrev xs ys = reverse xs ++ ys`

Proof by structural induction on xs

Induction step:

To show: `itrev (x:xs) ys = reverse (x:xs) ++ ys`

```
itrev (x:xs) ys
= itrev xs (x:ys)             -- by def of itrev
= reverse xs ++ (x:ys)        -- by IH
```


Generalization

Lemma `itrev xs ys = reverse xs ++ ys`

Proof by structural induction on xs

Induction step:

To show: `itrev (x:xs) ys = reverse (x:xs) ++ ys`


```
itrev (x:xs) ys
= itrev xs (x:ys)             -- by def of itrev
= reverse xs ++ (x:ys)        -- by IH
reverse (x:xs) ++ ys
```


Generalization

Lemma `itrev xs ys = reverse xs ++ ys`

Proof by structural induction on xs

Induction step:

To show: `itrev (x:xs) ys = reverse (x:xs) ++ ys`


```
itrev (x:xs) ys
= itrev xs (x:ys)             -- by def of itrev
= reverse xs ++ (x:ys)        -- by IH
reverse (x:xs) ++ ys
= (reverse xs ++ [x]) ++ ys   -- by def of reverse
```


Generalization

Lemma `itrev xs ys = reverse xs ++ ys`

Proof by structural induction on xs

Induction step:

To show: `itrev (x:xs) ys = reverse (x:xs) ++ ys`


```
itrev (x:xs) ys
= itrev xs (x:ys)             -- by def of itrev
= reverse xs ++ (x:ys)        -- by IH
reverse (x:xs) ++ ys
= (reverse xs ++ [x]) ++ ys   -- by def of reverse
= reverse xs ++ ([x] ++ ys)   -- by Lemma app_assoc
= reverse xs ++ (x:ys)        -- by def of ++
```

Generalization

Lemma `itrev xs ys = reverse xs ++ ys`

Proof by structural induction on `xs`

Induction step:

To show: `itrev (x:xs) ys = reverse (x:xs) ++ ys`

```
itrev (x:xs) ys
= itrev xs (x:ys)            -- by def of itrev
= reverse xs ++ (x:ys)       -- by IH
reverse (x:xs) ++ ys
= (reverse xs ++ [x]) ++ ys  -- by def of reverse
= reverse xs ++ ([x] ++ ys)  -- by Lemma app_assoc
= reverse xs ++ (x:ys)       -- by def of ++
```

Note: IH is used with `x:ys` instead of `ys`

When using the IH, variables may be replaced by arbitrary expressions, only the induction variable must stay fixed.

When using the IH, variables may be replaced by arbitrary expressions, only the induction variable must stay fixed.

Justification: all variables are implicitly ∀-quantified, except for the induction variable.

Generalization

Lemma `itrev xs ys = reverse xs ++ ys`

Proof by structural induction on `xs`

Induction step:

To show: `itrev (x:xs) ys = reverse (x:xs) ++ ys`

```
itrev (x:xs) ys
= itrev xs (x:ys)            -- by def of itrev
= reverse xs ++ (x:ys)       -- by IH
reverse (x:xs) ++ ys
= (reverse xs ++ [x]) ++ ys  -- by def of reverse
= reverse xs ++ ([x] ++ ys)  -- by Lemma app_assoc
= reverse xs ++ (x:ys)       -- by def of ++
```

Note: IH is used with `x:ys` instead of `ys`

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

```
qsort :: Ord a => [a] -> [a]
```

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 | y <- xs, x < z]
```

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 | y <- xs, x < z]
```

Lemma `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 | y <- xs, x < z]
```

Lemma `qsort xs` is sorted

Proof by induction on the length of the argument of qsort.

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 | y <- 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)`

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 | y <- 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).

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 | y <- 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.

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 | y <- 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)`.

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 | y <- 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)`

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 | y <- 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

```
qsort :: Ord a => [a] -> [a]
qsort []       =  []
qsort (x:xs)   = qsort below ++ [x] ++ qsort above
    where below = [y | y <- xs, y <= x]
          above = [z | y <- 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!).
Analogously for `above` and `(x<)`.

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 | y <- 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!).
Analogously for `above` and `(x<)`.
Therefore `qsort (x:xs)` is sorted.

Is that all?

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

How about this sorting function?

```
superquicksort _ = []
```

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

How about this sorting function?

```
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

5.2 Definedness

Simplifying assumption, implicit so far:

No undefined values

Two kinds of undefinedness:

`head []`	raises exception
`f x = f x + 1`	

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

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

Undefinedness can be handled, too.

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

Undefinedness can be handled, too.
But it complicates life

What is the problem?

Many familiar laws no longer hold unconditionally:

`x - x = 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`

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
  `f x = f x + 1`
  we could conclude that $0 = 1$.

Termination

Termination of a function means termination for all inputs.

Termination

Termination of a function means termination for all inputs.

Restriction:

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

Termination

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]
```

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.

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 :: T1 -> T` terminates
if there is a *measure function* `m :: T1 ->` $\mathbb{N}$ such that

- for every defining equation `f p = t`

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 :: T1 -> T` terminates
if there is a *measure function* `m :: T1 ->` $\mathbb{N}$ such that

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

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 :: T1 -> T` terminates
if there is a *measure function* `m :: T1 ->` $\mathbb{N}$ 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 :: T1 -> T` terminates
if there is a *measure function* `m :: T1 -> 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.

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 :: T1 -> T` terminates
if there is a *measure function* `m :: T1 -> 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, eg sequential pattern matching.

More generally: `f :: T1 -> ... -> Tn -> T` terminates
if there is a measure function `m :: T1 -> ... -> Tn -> ℕ`
such that

- for every defining equation `f p1 ... pn = t`

More generally: `f :: T1 -> ... -> Tn -> T` terminates
if there is a measure function `m :: T1 -> ... -> Tn -> ℕ`
such that

- for every defining equation `f p1 ... pn = t`
- and for every recursive call `f r1 ... rn` in `t`:
  `m p1 ... pn > m r1 ... rn`.

More generally: `f :: T1 -> ... -> Tn -> T` terminates if there is a measure function `m :: T1 -> ... -> Tn -> ℕ` such that

- for every defining equation `f p1 ... pn = t`
- and for every recursive call `f r1 ... rn` in `t`: `m p1 ... pn > m r1 ... rn`.

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

Infinite values

Infinite values

Haskell allows infinite values, in particular infinite lists.

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

Infinite objects must be constructed by recursion:

```
ones = 1 : ones
```

Infinite values

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, ...]`

Infinite objects must be constructed by recursion:

```
ones = 1 : ones
```

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

Note:

- By termination of functions we really mean termination on *finite* values.
- For example `reverse` terminates only on finite lists.

Infinite values

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.

Note:

- By termination of functions we really mean termination on *finite* values.
- For example `reverse` terminates only on finite lists.

This is fine because we can only construct finite values anyway.

How can infinite values be useful?
Because of "lazy evaluation".

Exceptions

If we use arithmetic equations like `x - x = 0` unconditionally, we can "lose" exceptions:

```
head xs - head xs = 0
```

Exceptions

If we use arithmetic equations like `x - x = 0` unconditionally, we can "lose" exceptions:

`head xs - head xs = 0`
is only true if `xs /= []`

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

Exceptions

If we use arithmetic equations like `x - x = 0` unconditionally, we can "lose" exceptions:

`head xs - head xs = 0`
is only true if `xs /= []`

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

If `e1` and `e2` do not produce a runtime exception then they evaluate to the same value.

Summary

- In this chapter everything must terminate

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]

```
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))
```

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.

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?

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 f pic1 pic2 1 = pic1
alt f pic1 pic2 n = f pic1 (alt f pic2 pic1 (n-1))
```

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 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:
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:
Functions that take functions as arguments

```
... -> (... -> ...) -> ...
```

Higher-order functions capture patterns of computation

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
```

6.1 Applying functions to all elements of a list: `map`

6.1 Applying functions to all elements of a list: `map`

Example

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

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"]
```


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 ::
```


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:

```
map f xs  =  [ f x | x <- xs ]
```


map: The mother of all higher-order functions

Predefined in Prelude.
Two possible definitions:

map f xs = [f x | x <- xs]


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

Evaluating map

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

Evaluating map

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


```
map sqr [1, -2]
```

Evaluating map

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

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

Evaluating map

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

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

Evaluating map

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

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

Some properties of map

```
length (map f xs) =
```

Some properties of map

```
length (map f xs) = length xs
```

Some properties of map

```
length (map f xs) = length xs

map f (xs ++ ys) =
```

Some properties of map

```
length (map f xs) = length xs

map f (xs ++ ys) = map f xs ++ map f ys
```

Some properties of map

```
length (map f xs) = length xs
```

Some properties of map

```
length (map f xs) = length xs

map f (xs ++ ys) = map f xs ++ map f 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) =
```

Some properties of map

```
length (map f xs) = length xs

map f (xs ++ ys) = map f xs ++ map f ys

map f (reverse xs) = reverse (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.

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`

```
length (map f xs) = length xs

map f (xs ++ ys) = map f xs ++ map f ys

map f (reverse xs) = reverse (map f 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

```
prop_map_even :: [Int] -> [Int] -> Bool
prop_map_even xs ys =
  map even (xs ++ ys) = map even xs ++ map even ys
```

6.2 Filtering a list: `filter`

6.2 Filtering a list: filter

Example

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

6.2 Filtering a list: filter
Example
filter even [1, 2, 3]
= [2]

```
filter isAlpha "R2-D2"
= "RD"
```

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 [[], [1,2], []]
= [[], []]
```

What is the type of `filter`?

```
filter :: (a -> Bool) ->      ->
```


6.2 Filtering a list: `filter`

Example

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

filter isAlpha "R2-D2"
= "RD"

filter null [[], [1,2], []]
= [[], []]
```

What is the type of `filter`?


```
filter :: (a -> Bool) -> [a] -> [a]
```

`filter`

Predefined in `Prelude`.
Two possible definitions:

```
filter p xs  =  [ x | x <- xs, p x ]
```


`filter`

Predefined in `Prelude`.
Two possible definitions:

```
filter p xs  =  [ x | x <- xs, p x ]
```


```
filter p []                    =  []
filter p (x:xs) | p x          =  x : filter p xs
                | otherwise    =  filter p xs
```

Some properties of `filter`

```
filter p (xs ++ ys) = filter p xs ++ filter p ys
```

```
filter p (reverse xs) = reverse (filter p xs)
```

Proofs by induction