Script generated by TTT

Title: Nipkow: Info2 (07.11.2014)

Date: Fri Nov 07 07:30:13 GMT 2014

Duration: 88:45 min

Pages: 124

Inefficiency of `reverse`

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

Inefficiency of `reverse`

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

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]
itrev [] xs     = xs
```

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:

IH: `itrev xs [] = reverse xs`

Proof attempt

Lemma `itrev xs [] = reverse xs`

Proof by structural induction on xs

Induction step fails:

IH: `itrev xs [] = reverse xs`

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

```
itrev (x:xs) []
```

Proof attempt

Lemma `itrev xs [] = reverse xs`

Proof by structural induction on xs

Induction step fails:

IH: `itrev xs [] = reverse xs`

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

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

Proof attempt

Lemma `itrev xs [] = reverse xs`

Proof by structural induction on xs

Induction step fails:

IH: `itrev xs [] = reverse xs`

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

IH: `itrev xs [] = reverse xs`

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

IH: `itrev xs ys = reverse xs ++ ys`

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:

IH: `itrev xs ys = reverse xs ++ ys`

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:
IH: `itrev xs ys = reverse xs ++ ys`
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:
IH: `itrev xs ys = reverse xs ++ ys`
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
```


Generalization

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

Proof by structural induction on xs

Induction step:
IH: `itrev xs ys = reverse xs ++ ys`
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 ++
```

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.

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.

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

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

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

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`

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 $0 = 1$.

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.

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 -> 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.

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 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.

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

How can infinite values be useful?
Because of “lazy evaluation”.
More later.

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 /= []`

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*:

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

5.3 Interlude: Type inference/reconstruction

How to infer/reconstruct the type of an expression
(and all subexpressions)

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

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

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`

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"]
```

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] ->
```

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 ]
```

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 : [])
= sqr 1 : sqr (-2) : (map sqr [])
= sqr 1 : sqr (-2) : []
= 1 : 4 : []
= [1, 4]
```

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
```

```
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.

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
```

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)

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], []]
= [[], []]
```

What is the type of `filter`?

```
filter ::           ->     ->
```

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] ->
```

`filter`

Predefined in `Prelude`.

`filter`

Predefined in `Prelude`.
Two possible definitions:

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

Some properties of `filter`

True or false?

Some properties of `filter`

True or false?

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

Some properties of `filter`

True or false?

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

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

Some properties of `filter`

True or false?

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

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

```
filter p (map f xs) = map f (filter p xs)
```

5.3 Interlude: Type inference/reconstruction

How to infer/reconstruct the type of an expression
(and all subexpressions)

5.3 Interlude: Type inference/reconstruction

How to infer/reconstruct the type of an expression
(and all subexpressions)

Given: an expression e

Type inference:

5.3 Interlude: Type inference/reconstruction

How to infer/reconstruct the type of an expression
(and all subexpressions)

Given: an expression e

Type inference:

1. Give all variables and functions in e their most general type

5.3 Interlude: Type inference/reconstruction

How to infer/reconstruct the type of an expression
(and all subexpressions)

Given: an expression e

Type inference:

1. Give all variables and functions in e their most general type
2. From e set up a system of equations between types

5.3 Interlude: Type inference/reconstruction

How to infer/reconstruct the type of an expression
(and all subexpressions)

Given: an expression *e*

Type inference:

1. Give all variables and functions in *e* their most general type
2. From *e* set up a system of equations between types
3. Simplify the equations

Example: `concat (replicate x y)`

Example: `concat (replicate x y)`

Initial type table:

```
x ::  a
```

Example: `concat (replicate x y)`

Initial type table:

```
x ::  a
y ::  b
replicate ::  Int -> c -> [c]
```

Example: `concat (replicate x y)`

Initial type table:

```
x :: a
y :: b
replicate :: Int -> c -> [c]
concat :: [[d]] -> [d]
```

Example: `concat (replicate x y)`

Initial type table:

```
x :: a
y :: b
replicate :: Int -> c -> [c]
concat :: [[d]] -> [d]
```

For each subexpression $f\ e_1 \ldots e_n$ generate n equations:

Example: `concat (replicate x y)`

Initial type table:

```
x :: a
y :: b
replicate :: Int -> c -> [c]
concat :: [[d]] -> [d]
```

For each subexpression $f\ e_1 \ldots e_n$ generate n equations:

`a = Int`,

Example: `concat (replicate x y)`

Initial type table:

```
x :: a
y :: b
replicate :: Int -> c -> [c]
concat :: [[d]] -> [d]
```

For each subexpression $f\ e_1 \ldots e_n$ generate n equations:

`a = Int`, `b = c`

Example: concat (replicate x y)

Initial type table:

```
x :: a
y :: b
replicate :: Int -> c -> [c]
concat :: [[d]] -> [d]
```

For each subexpression $f\ e_1 \ldots e_n$ generate n equations:

```
a = Int, b = c
[c] = [[d]]
```


Example: concat (replicate x y)

Initial type table:

```
x :: a
y :: b
replicate :: Int -> c -> [c]
concat :: [[d]] -> [d]
```

For each subexpression $f\ e_1 \ldots e_n$ generate n equations:

```
a = Int, b = c
[c] = [[d]]
```


Simplify equations:

Example: concat (replicate x y)

Initial type table:

```
x :: a
y :: b
replicate :: Int -> c -> [c]
concat :: [[d]] -> [d]
```

For each subexpression $f\ e_1 \ldots e_n$ generate n equations:

```
a = Int, b = c
[c] = [[d]]
```

Simplify equations:

`[c] = [[d]]` ⇝

Example: concat (replicate x y)

Initial type table:

```
x :: a
y :: b
replicate :: Int -> c -> [c]
concat :: [[d]] -> [d]
```

For each subexpression $f\ e_1 \ldots e_n$ generate n equations:

```
a = Int, b = c
[c] = [[d]]
```

Simplify equations: `[c] = [[d]]` ⇝

`c = [d]`

Example: `concat (replicate x y)`

Initial type table:

```
x :: a
y :: b
replicate :: Int -> c -> [c]
concat :: [[d]] -> [d]
```

For each subexpression $f\ e_1 \ldots e_n$ generate n equations:

```
a = Int, b = c
[c] = [[d]]
```

Simplify equations: `[c] = [[d]]` ⇝ `c = [d]`
`b = c` ⇝ `b = [d]`

Example: `concat (replicate x y)`

Initial type table:

```
x :: a
y :: b
replicate :: Int -> c -> [c]
concat :: [[d]] -> [d]
```

For each subexpression $f\ e_1 \ldots e_n$ generate n equations:

```
a = Int, b = c
[c] = [[d]]
```

Simplify equations: `[c] = [[d]]` ⇝ `c = [d]`
`b = c` ⇝ `b = [d]`

Solution to equation system: `a = Int, b = [d], c = [d]`

Final type table:

```
x :: Int
y :: [d]
replicate :: Int -> [d] -> [[d]]
```

Example: `concat (replicate x y)`

Initial type table:

```
x :: a
y :: b
replicate :: Int -> c -> [c]
concat :: [[d]] -> [d]
```

For each subexpression $f\ e_1 \ldots e_n$ generate n equations:

```
a = Int, b = c
[c] = [[d]]
```

Simplify equations: `[c] = [[d]]` ⇝ `c = [d]`
`b = c` ⇝ `b = [d]`

Solution to equation system: `a = Int, b = [d], c = [d]`

Final type table:

```
x :: Int
y :: [d]
replicate :: Int -> [d] -> [[d]]
concat :: [[d]] -> [d]
```

Algorithm

1. Give the variables $x_1, \ldots, x_n$ in e the types $a_1, \ldots a_n$ where the a_i are distinct type variables.
2. Give *each occurrence* of a function $f :: \tau$ in e a new type τ' that is a copy of τ with fresh type variables.

Algorithm

1. Give the variables $x_1, \ldots, x_n$ in e the types $a_1, \ldots a_n$ where the a_i are distinct type variables.

Algorithm

1. Give the variables $x_1, \ldots, x_n$ in e the types $a_1, \ldots a_n$ where the a_i are distinct type variables.
2. Give *each occurrence* of a function $f :: \tau$ in e a new type τ' that is a copy of τ with fresh type variables.

Algorithm

1. Give the variables $x_1, \ldots, x_n$ in e the types $a_1, \ldots a_n$ where the a_i are distinct type variables.
2. Give *each occurrence* of a function $f :: \tau$ in e a new type τ' that is a copy of τ with fresh type variables.
3. For each subexpression $f\ e_1 \ldots e_n$ of e where $f :: \tau_1 \to \cdots \to \tau_n \to \tau$ and where e_i has type σ_i generate the equations $\sigma_1 = \tau_1,\ \ldots,\ \sigma_n = \tau_n$.

Algorithm

1. Give the variables $x_1, \ldots, x_n$ in e the types $a_1, \ldots a_n$ where the a_i are distinct type variables.
2. Give *each occurrence* of a function $f :: \tau$ in e a new type τ' that is a copy of τ with fresh type variables.

Algorithm

1. Give the variables $x_1, \ldots, x_n$ in e the types $a_1, \ldots a_n$ where the a_i are distinct type variables.
2. Give *each occurrence* of a function $f :: \tau$ in e a new type τ' that is a copy of τ with fresh type variables.
3. For each subexpression $f\ e_1 \ldots e_n$ of e where $f :: \tau_1 \to \cdots \to \tau_n \to \tau$ and where e_i has type σ_i generate the equations $\sigma_1 = \tau_1, \ \ldots, \ \sigma_n = \tau_n$.
4. Simplify the equations with the following rules as long as possible:

Algorithm

1. Give the variables $x_1, \ldots, x_n$ in e the types $a_1, \ldots a_n$ where the a_i are distinct type variables.
2. Give *each occurrence* of a function $f :: \tau$ in e a new type τ' that is a copy of τ with fresh type variables.
3. For each subexpression $f\ e_1 \ldots e_n$ of e where $f :: \tau_1 \to \cdots \to \tau_n \to \tau$ and where e_i has type σ_i generate the equations $\sigma_1 = \tau_1, \ \ldots, \ \sigma_n = \tau_n$.
4. Simplify the equations with the following rules as long as possible:
   - $a = \tau$ or $\tau = a$: replace type variable a by τ everywhere (if a does not occur in τ)

Algorithm

1. Give the variables $x_1, \ldots, x_n$ in e the types $a_1, \ldots a_n$ where the a_i are distinct type variables.
2. Give *each occurrence* of a function $f :: \tau$ in e a new type τ' that is a copy of τ with fresh type variables.
3. For each subexpression $f\ e_1 \ldots e_n$ of e where $f :: \tau_1 \to \cdots \to \tau_n \to \tau$ and where e_i has type σ_i generate the equations $\sigma_1 = \tau_1, \ \ldots, \ \sigma_n = \tau_n$.
4. Simplify the equations with the following rules as long as possible:
   - $a = \tau$ or $\tau = a$: replace type variable a by τ everywhere (if a does not occur in τ)
   - $T\ \sigma_1 \ldots \sigma_n = T\ \tau_1 \ldots \tau_n \ \leadsto\ \sigma_1 = \tau_1, \ \ldots, \ \sigma_n = \tau_n$ (where T is a type constructor, e.g. `[.]`, `.->.`, etc)

Algorithm

1. Give the variables $x_1, \ldots, x_n$ in e the types $a_1, \ldots a_n$ where the a_i are distinct type variables.
2. Give *each occurrence* of a function $f :: \tau$ in e a new type τ' that is a copy of τ with fresh type variables.
3. For each subexpression $f\ e_1 \ldots e_n$ of e where $f :: \tau_1 \to \cdots \to \tau_n \to \tau$ and where e_i has type σ_i generate the equations $\sigma_1 = \tau_1, \ \ldots, \ \sigma_n = \tau_n$.
4. Simplify the equations with the following rules as long as possible:
   - $a = \tau$ or $\tau = a$: replace type variable a by τ everywhere (if a does not occur in τ)
   - $T\ \sigma_1 \ldots \sigma_n = T\ \tau_1 \ldots \tau_n \ \leadsto\ \sigma_1 = \tau_1, \ \ldots, \ \sigma_n = \tau_n$ (where T is a type constructor, e.g. `[.]`, `.->.`, etc)
   - $a = T \ldots a \ldots$ or $T \ldots a \ldots = a$: type error!

Algorithm

1. Give the variables $x_1, \ldots, x_n$ in e the types $a_1, \ldots a_n$ where the a_i are distinct type variables.
2. Give *each occurrence* of a function $f :: \tau$ in e a new type τ' that is a copy of τ with fresh type variables.
3. For each subexpression $f\ e_1 \ldots e_n$ of e where $f :: \tau_1 \to \cdots \to \tau_n \to \tau$ and where e_i has type σ_i generate the equations $\sigma_1 = \tau_1, \ldots, \sigma_n = \tau_n$.
4. Simplify the equations with the following rules as long as possible:
   - $a = \tau$ or $\tau = a$: replace type variable a by τ everywhere (if a does not occur in τ)
   - $T\ \sigma_1 \ldots \sigma_n = T\ \tau_1 \ldots \tau_n \rightsquigarrow \sigma_1 = \tau_1, \ldots, \sigma_n = \tau_n$ (where T is a type constructor, e.g. `[.]`, `.->.`, etc)
   - $a = T \ldots a \ldots$ or $T \ldots a \ldots = a$: type error!
   - $T \ldots = T' \ldots$ where $T \neq T'$: type error!

- For simple expressions you should be able to infer types "durch scharfes Hinsehen"
- Use the algorithm if you are unsure or the expression is complicated

- For simple expressions you should be able to infer types "durch scharfes Hinsehen"
- Use the algorithm if you are unsure or the expression is complicated
- Or use the Haskell interpreter

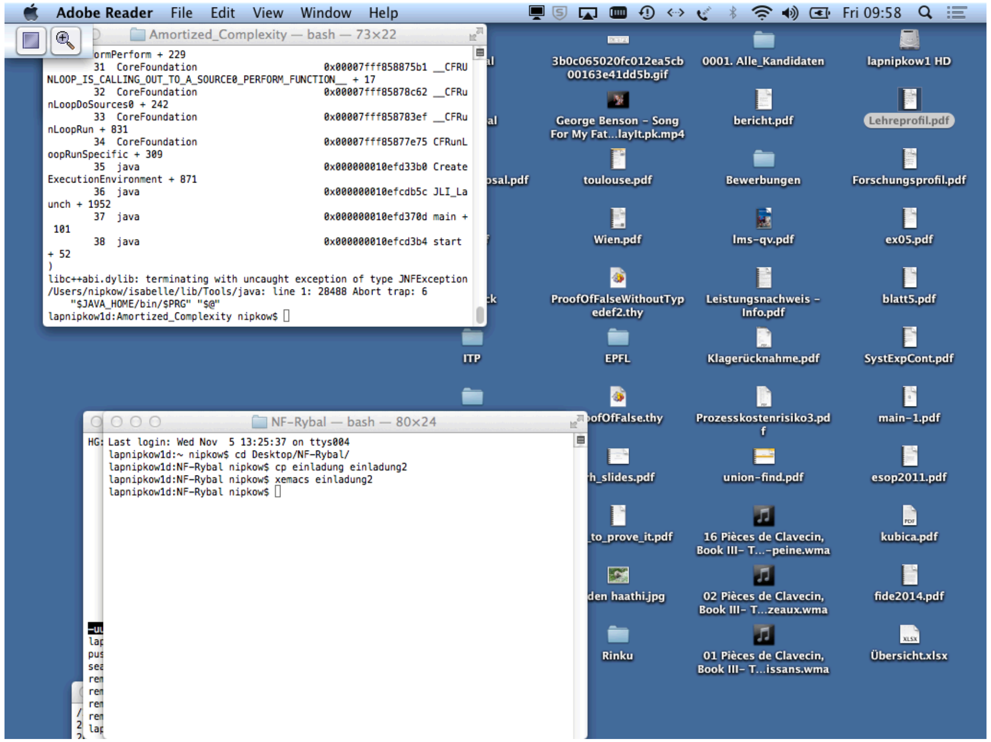
Some properties of `filter`

True or false?

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

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

filter p (map f xs) = map f (filter p xs)
```


Adobe Reader File Edit View Window Help
Fri 09:58
Amortized_Complexity — bash — 73×22
ormPerform + 229
31 CoreFoundation 0x00007fff858875b1 __CFRU
NLOOP_IS_CALLING_OUT_TO_A_SOURCE0_PERFORM_FUNCTION__ + 17
32 CoreFoundation 0x00007fff85878c62 __CFRu
nLoopDoSources0 + 242
33 CoreFoundation 0x00007fff858783ef __CFRu
nLoopRun + 831
34 CoreFoundation 0x00007fff85877e75 CFRunL
oopRunSpecific + 309
35 java 0x000000010efd33b0 Create
ExecutionEnvironment + 871
36 java 0x000000010efcdb5c JLI_La
unch + 1952
37 java 0x000000010efd370d main +
101
38 java 0x000000010efcd3b4 start
+ 52
)
libc++abi.dylib: terminating with uncaught exception of type JNFException
/Users/nipkow/isabelle/lib/Tools/java: line 1: 28488 Abort trap: 6
"$JAVA_HOME/bin/$PRG" "$@"
lapnipkow1d:Amortized_Complexity nipkow$
NF-Rybal — bash — 80×24
Last login: Wed Nov 5 13:25:37 on ttys004
lapnipkow1d:~ nipkow$ cd Desktop/NF-Rybal/
lapnipkow1d:NF-Rybal nipkow$ cp einladung einladung2
lapnipkow1d:NF-Rybal nipkow$ xemacs einladung2
lapnipkow1d:NF-Rybal nipkow$
3b0c065020fc012ea5cb00163e41dd5b.gif
0001. Alle_Kandidaten
lapnipkow1 HD
George Benson - Song For My Fat...laylt.pk.mp4
bericht.pdf
Lehreprofil.pdf
toulouse.pdf
Bewerbungen
Forschungsprofil.pdf
Wien.pdf
lms-qv.pdf
ex05.pdf
ProofOfFalseWithoutTypedef2.thy
Leistungsnachweis - Info.pdf
blatt5.pdf
ITP
EPFL
Klagerücknahme.pdf
SystExpCont.pdf
Prozesskostenrisiko3.pdf
main-1.pdf
union-find.pdf
esop2011.pdf
16 Pièces de Clavecin, Book III- T...-peine.wma
kubica.pdf
02 Pièces de Clavecin, Book III- T...zeaux.wma
fide2014.pdf
Rinku
01 Pièces de Clavecin, Book III- T...issans.wma
Übersicht.xlsx