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

\[
\begin{align*}
\text{itrev} :: [a] \rightarrow [a] \rightarrow [a] \\
\text{itrev} [] \, x s & = x s \\
\text{itrev} (x:x s) \, y s & = \text{itrev} \, x s \, (x:y s)
\end{align*}
\]

\[
\begin{align*}
itrev [1,2,3] [] \\
& = \text{itrev} [2,3] [1] \\
& = \text{itrev} [3] [2,1] \\
& = \text{itrev} [] [3,2,1] \\
& = [3,2,1]
\end{align*}
\]

Proof attempt

**Lemma** \( \text{itrev} \, x s \, [] = \text{reverse} \, x s \)

**Proof** by structural induction on \( x s \)

Induction step fails:

IH: \( \text{itrev} \, x s \, [] = \text{reverse} \, x s \)
Lemma \( \text{itrev} \ [x:] = \text{reverse} \ [x:] \)

Proof by structural induction on \( x \)
Induction step fails:
IH: \( \text{itrev} \ [x:] = \text{reverse} \ [x:] \)
To show: \( \text{itrev} \ (x:x:) \ [x:] = \text{reverse} \ (x:x:) \)
\( \quad = \text{itrev} \ [x:] \quad -- \text{by def of itrev} \)
\( \quad = \text{reverse} \ [x:] \quad -- \text{by def of reverse} \)

Problem: IH not applicable because too specialized: \( [] \)
Lemma \textit{itrev} \textit{xs} \textit{ys} = \textit{reverse} \textit{xs} ++ \textit{ys} \\
\textbf{Proof} by structural induction on \textit{xs} \\
Induction step: \\
IH: \textit{itrev} \textit{xs} \textit{ys} = \textit{reverse} \textit{xs} ++ \textit{ys} \\
To show: \textit{itrev} \ (\textit{x}:\textit{xs}) \textit{ys} = \textit{reverse} \ (\textit{x}:\textit{xs}) ++ \textit{ys} \\
\hspace{1cm} \begin{align*} 
\textit{itrev} \ (\textit{x}:\textit{xs}) \textit{ys} \\
= \textit{itrev} \textit{xs} \ (\textit{x}:\textit{ys}) \quad -- \text{by def of \textit{itrev}} 
\end{align*}
\begin{align*} 
\textit{reverse} \ (\textit{x}:\textit{xs}) ++ \textit{ys} 
\end{align*}
**Lemma** \( \text{itrev } xs \ ys = \text{reverse } xs \ ++ \ ys \)

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

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

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 \(\forall\)-quantified, except for the induction variable.

Induction on the length of a list

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

\[
\begin{align*}
\text{qsort} & [] = [] \\
\text{qsort} (x:xs) = \text{qsort} \ \text{below} \ ++ \ [x] \ ++ \ \text{qsort} \ \text{above} \\
& \text{where } \text{below} = [y \mid y \leftarrow xs, y \leq x] \\
& \text{above} = [z \mid z \leftarrow xs, x < z]
\end{align*}
\]
Induction on the length of a list

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

**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?

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

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
- \( f \ x = f \ x + 1 \)
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.

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: \( \text{head} [] - \text{head} [] = 0 \)
- 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: \( \text{head } [] - \text{head } [] = 0 \)

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.
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 \( t \): \( m \ p > m \ r \).
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.

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.

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

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”.

More later.

Exceptions

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

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

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 $xs \neq []$
Exceptions

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

$$\text{head } \text{xs} - \text{head } \text{xs} = 0$$
$$\text{is only true if } \text{xs} \neq []$$

In such cases, we can prove equations $\text{e1} = \text{e2}$ that are only *partially correct*:

Summary

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

Summary

5.3 Interlude: Type inference/reconstruction

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

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

\[ \ldots \rightarrow (\ldots \rightarrow \ldots) \rightarrow \ldots \]

Higher-order functions capture patterns of computation

6.1 Applying functions to all elements of a list: \texttt{map}

Example

\begin{verbatim}
map even [1, 2, 3] = [False, True, False]
\end{verbatim}
6.1 Applying functions to all elements of a list: `map`

**Example**

```haskell
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`?

```haskell
map :: (a -> b) -> [a] -> [b]
```
map: The mother of all higher-order functions

Predefined in Prelude.

Two possible definitions:

\[
\text{map } f \ x \ s = [ f \ x \mid x \leftarrow s ]
\]

Evaluating map

\[
\begin{align*}
\text{map } f \ [] & = [] \\
\text{map } f \ (x:xs) & = f \ x : \text{map } f \ xs \\
\text{map } \text{sqr} \ [1, -2] & = \text{map } \text{sqr} \ (1 : -2 : [])
\end{align*}
\]
Evaluating maps

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) = length xs

Evaluating maps

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
map f (xs ++ ys) =
Some properties of \texttt{map}

- \texttt{length (map f xs) = length xs}
- \texttt{map f (xs ++ ys) = map f xs ++ map f ys}
- \texttt{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

\begin{verbatim}
prop_map_even :: [Int] \rightarrow [Int] \rightarrow Bool
prop_map_even xs ys =
  map even (xs ++ ys) = map even xs ++ map even ys
\end{verbatim}
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`

Example

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

```
filter isAlpha "R2-D2"
= "RD"
```
6.2 Filtering a list: \texttt{filter}

Example

\begin{verbatim}
filter even [1, 2, 3] = [2]
filter isAlpha "R2-D2" = "RD"
filter null [[]] = [[], []]
\end{verbatim}

What is the type of \texttt{filter}?

\texttt{filter :: (a -> Bool) -> [a] -> [a]}

Predefined in Prelude.

Predefined in Prelude.  
Two possible definitions:

\begin{verbatim}
filter p xs = [ x | x <- xs, p x ]
\end{verbatim}
Some properties of `filter`:

True or false?

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

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

True or false?

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

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

\[
\text{filter } p \ (\text{map } f \ xs) = \text{map } f \ (\text{filter } p \ xs)
\]
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)`

Initial type table:

\[
\begin{align*}
  x & \mapsto a \\
  y & \mapsto b \\
  \text{replicate} & \mapsto \text{Int} \to c \to [c]
\end{align*}
\]
Example: \texttt{concat} (\texttt{replicate} \texttt{x} \texttt{y})

Initial type table:
\begin{verbatim}
    x :: a
    y :: b
    replicate :: Int -> c -> [c]
    concat :: [[d]] -> [d]
\end{verbatim}

For each subexpression \( f \ e_1 \ldots e_n \) generate \( n \) equations:
\begin{verbatim}
    a = Int,
\end{verbatim}
Example: `concat (replicate x y)`

Initial type table:

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

For each subexpression `f e₁…eₙ` 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 e₁…eₙ` 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 e₁…eₙ` 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 e₁…eₙ` generate `n` equations:

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

Simplify equations:
Example: \texttt{concat (replicate x y)}

Initial type table:
\begin{align*}
x &::= a \\
y &::= b \\
\text{replicate} &::= \text{Int} \rightarrow c \rightarrow [c] \\
\text{concat} &::= [[d]] \rightarrow [d]
\end{align*}

For each subexpression \( f e_1 \ldots e_n \) generate \( n \) equations:
\begin{align*}
a &= \text{Int}, b = c \\
[c] &= [[d]]
\end{align*}

Simplify equations:
\begin{align*}
[c] &= [[d]] \sim c = [d] \\
b &= c \sim b = [d]
\end{align*}

Solution to equation system:
\begin{align*}
a &= \text{Int}, b = [d], c = [d]
\end{align*}

Final type table:
\begin{align*}
x &::= \text{Int} \\
y &::= [d] \\
\text{replicate} &::= \text{Int} \rightarrow [d] \rightarrow [[d]] \\
\text{concat} &::= [[d]] \rightarrow [d]
\end{align*}

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 \( \tau' \) that is a copy of \( \tau \) 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 $\tau'$ that is a copy of $\tau$ with fresh type variables.

3. For each subexpression $f \ e_1 \ldots e_n$ of $e$ where $f :: \tau_1 \rightarrow \cdots \rightarrow \tau_n \rightarrow \tau$ and where $e_i$ has type $\sigma_i$, 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 $\tau'$ that is a copy of $\tau$ 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 \( \tau' \) that is a copy of \( \tau \) with fresh type variables.
3. For each subexpression \( f e_1 \ldots e_n \) of \( e \) where \( f :: \tau_1 \rightarrow \cdots \rightarrow \tau_n \rightarrow \tau \) and where \( e_i \) has type \( \sigma_i \) 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 \( \tau \) everywhere (if \( a \) does not occur in \( \tau \))
   - \( T \sigma_1 \ldots \sigma_n = T \tau_1 \ldots \tau_n \rightarrow \sigma_1 = \tau_1, \ldots, \sigma_n = \tau_n \)
     (where \( T \) is a type constructor, e.g. \( [\cdot], \text{->}, \text{etc} \))

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 \( \tau' \) that is a copy of \( \tau \) with fresh type variables.
3. For each subexpression \( f e_1 \ldots e_n \) of \( e \) where \( f :: \tau_1 \rightarrow \cdots \rightarrow \tau_n \rightarrow \tau \) and where \( e_i \) has type \( \sigma_i \) 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 \( \tau \) everywhere (if \( a \) does not occur in \( \tau \))
   - \( T \sigma_1 \ldots \sigma_n = T \tau_1 \ldots \tau_n \rightarrow \sigma_1 = \tau_1, \ldots, \sigma_n = \tau_n \)
     (where \( T \) is a type constructor, e.g. \( [\cdot], \text{->}, \text{etc} \))
   - \( a = T \ldots a \ldots \) 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 $\tau'$ that is a copy of $\tau$ with fresh type variables.
3. For each subexpression $f \ e_1 \ldots e_n$ of $e$ where $f :: \tau_1 \rightarrow \cdots \rightarrow \tau_n \rightarrow \tau$ and where $e_i$ has type $\sigma_i$ 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 $\tau$ everywhere (if $a$ does not occur in $\tau$)
   - $T \sigma_1 \ldots \sigma_n = T \tau_1 \ldots \tau_n \leadsto \sigma_1 = \tau_1, \ldots, \sigma_n = \tau_n$
     (where $T$ is a type constructor, e.g. [], ->, etc)
   - $a = T \ldots a \ldots$ or $T \ldots a \ldots = a$: type error!
   - $T \ldots = T \ldots$ where $T \neq T'$: type error!

Some properties of filter

- 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

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)