Title: Lammich: FDS Tutorial (12.05.2017)
Date: Fri May 12 12:20:33 CEST 2017
Duration: 97:27 min
Pages: 93
Exercise: Given a list `xs`, define a function `contains` that checks whether an element is contained in the list. Define the function directly, not using `@set`.

```ocaml
fun contains :: "a => 'a list => bool"
where
  "contains x [] = False"
```

To define a predicate `idistinct` to characterize `@set@[distinct]` lists, i.e., lists whose elements are pairwise disjoint. Hint: Use the function `contains`.

```ocaml
fun idistinct :: "a list => bool"
where
  "idistinct [] = True"
```

Show that a reversed list is distinct if and only if the original list is distinct. Hint: You may require multiple auxiliary lemmas.

```ocaml
lemma "idistinct (rev xs) == idistinct xs"
apply induction xs
apply auto
```

Consists:
```
fun idistinct :: "a list => bool"
```
Inner syntax error: Unexpected end of inputs
Failed to parse prop

proof (prove)

1. \( \text{distinct } \{x: \text{list} \} = (\text{distinct } \{x\} \land \text{contains } x) \)
fun slice [] : list => n = n @ n = list

where
| slice [()] (Suc n) = slice xs @ n
| slice [x] @ Suc n = x # slice xs @ n

slice [] = []

fun H : list => H = list

where
| H = H @ H = list

let H = H

fun f : list => f = list

where
| f = f @ f = list

text = Show that concatenation of two adjacent slices can be expressed as a single slice:

lemma slice xs @ slice xs @ (Suc n) 12 = slice xs @ (Suc n) 12

apply induction on slice xs @ slice induc

auto

lemma slice xs @ slice xs @ (Suc n) 12 = slice xs @ (Suc n) 12

apply induction on slice xs @ slice induc

auto

proof [prove]

val t at app =

1. Ax u v. slice u v. 0 @ slice u v. (Suc n) 12 = slice u v. (Suc n) 12
2. Ax u v. slice u v. 0 @ slice u v. (Suc n) 12 = slice u v. (Suc n) 12
3. Ax n. slice xs @ n = slice xs @ (Suc n) 12 = slice xs @ (Suc n) 12

apply induction on slice xs @ slice induc
proof (prove)

1. slice [] x y = []
fun slice :: "a list ⇒ nat ⇒ nat ⇒ "a list"

where
"slice _ 0 = []" 
"slice _ _ = []" 
"slice [] 0 = []" 
"slice (Suc n) 0 = []" 
"slice [a] 0 = [a]" 
"slice [a] (Suc n) = [a]" 
"slice (Suc n) 0 = []" 
"slice (Suc n) (Suc m) = slice n m" 

the slice.simps

lemma "slice [a] (Suc n) = slice n 0" by (cases 0) auto

fun slice :: "a list ⇒ nat ⇒ nat ⇒ "a list"

where
"slice _ 0 = []" 
"slice _ _ = []" 
"slice [] 0 = []" 
"slice (Suc n) 0 = []" 
"slice [a] 0 = [a]" 
"slice [a] (Suc n) = [a]" 
"slice (Suc n) 0 = []" 
"slice (Suc n) (Suc m) = slice n m" 

the slice.simps

lemma "slice [a] (Suc n) = slice n 0" by (cases 0) auto
Functional Data Structures
Exercise Sheet 3

Exercise 3.1 Insert with Less Comparisons
- Define a function `ins2` that inserts into a binary search tree, using only one comparison per node. Use the same idea as `ins1`.
- Show that your function is equal to `ins1` on binary search trees. Hint: You may need an auxiliary lemma of the form:
  \[
  \begin{align*}
  \forall t : \forall x \in \text{set}(t) : x < z \Rightarrow \text{ins2}(x)(\text{Some } z) t &= \ldots \\
  \end{align*}
  \]

  \[
  \begin{align*}
  \text{fun ins2} : \forall a : \\text{order} \Rightarrow \forall a' : \\text{tree} \Rightarrow \forall a'' : \\text{tree} \rightarrow \\text{tree} \\
  \text{lemma ins2:} \forall a : \forall a' : \forall a'' : \forall x : \\text{tree} \Rightarrow \text{ins2}(x)(\text{Some } a')(\text{Some } a'') = \text{ins1}(x)(\text{Some } a')(\text{Some } a'') \ldots
  \end{align*}
  \]

Exercise 3.2 Height-Preserving In-Order Join
section "Reducing the Number of Comparisons"

text: Idea: never test for \( \downarrow \) but remember the last value you have tested for \( \downarrow \) but did not. Compare with that value when you reach a \( \uparrow \).

fun isidr \( \text{('#x::invert'), tree = 'a option = 'a = bool'} \) where
isidr Leaf \( d \ a \) \( x = \)
(\( \text{if } x = a \text{ then } isidr \ d \ a \ x \text{ else } isidr \ y \ (\text{Some } a y) \) )

lemma isidr_Some:
\[
\text{isidr}_d a x \text{ is } \text{Some } a y \Rightarrow \text{isidr}_d a y \text{ is } \text{Some } a x
\]

consts:
\[
\text{isidr} : \quad \text{t : tree = 'a option = 'a = bool'}
\]

found termination order: \( \text{size (fst t)} \) = \( \text{hasht} \)
Exercise 3.1 Insert with Less Comparisons

- Define a function \( \text{ins2} \) that inserts into a binary search tree, using only one comparison per node. Use the same idea as \( \text{ins2} \).
- Show that your function is equal to \( \text{ins2} \) on binary search trees. Hint: You may need an auxiliary lemma of the form:

\[
\text{ins2} (\text{Some } y) t = \ldots \quad \text{ins2} (\text{None}) t = \ldots \quad \text{ins2} (\text{Leaf } x) t = \ldots
\]

**Lemma** \( \text{ins2} \text{None} \): \( \text{Ins } t \implies \text{ins2 } \text{None } t = \text{Ins } x t \)

Exercise 3.2 Height-Preserving In-Order Join
Type unification failed: Clash of types "unit" and "tree".
Type error in application: incompatible operand type.
Operator: op :: [sm2 x z :: !] :: tree = bool.
Operator: (!) :: unit.

---

Type unification failed: Clash of types "unit" and "tree".
Type error in application: incompatible operand type.
Operator: op :: [sm2 x z :: !] :: tree = bool.
Operator: (!) :: unit.

---

Type unification failed: Clash of types "unit" and "tree".
Type error in application: incompatible operand type.
Operator: op :: [sm2 x z :: !] :: tree = bool.
Operator: (!) :: unit.

---

Type unification failed: Clash of types "unit" and "tree".
Type error in application: incompatible operand type.
Operator: op :: [sm2 x z :: !] :: tree = bool.
Operator: (!) :: unit.
begin
  fun smz : 'a option => 'a tree => 'a tree
  where
  smz x Leaf = x
  case z of
  None -> (Leaf, x, Leaf)
  Some y = (if y < x then Leaf else Node Leaf x Leaf))
  | "smz x z (Node a r) = undefined" end

inner syntax errors
Failed to parse prop

begin
  fun smz : 'a option => 'a tree => 'a tree
  where
  smz x Leaf = x
  case z of
  None -> (Leaf, x, Leaf)
  Some y = (if y < x then Leaf else Node Leaf x Leaf))
  | "smz x z (Node a r) = undefined" end

cond
smz : "x = 'a option => 'a tree => 'a tree"
Found termination order: "{"}
begin

term ism2

fun iso1 : "(m:l) ordiner => a option => a tree => a tree" where
| ism2 x z Leaf = |
| case x of
| None => (Leaf x, Leaf)
| Some y = (if x > y then Leaf else Node Leaf y Leaf)
| | ism2 x z (Node l y r) = if x > y then Node (ism2 x z l) y (ism2 x z r)
| | else Node (ism2 x z l) y (ism2 x z r)

end

 CONS5: "is = a option => a tree => a tree" Found termination order: "(is, size (and (is))) at
"otherwise []"

proof (prove)

goal (2 subgoals):
1. iso1 x None [] = iso1 x []
2. iso1 x z [1] = iso1 x z [2]

apply induction

end
Functional Data Structures
Exercise Sheet 3

Exercise 3.1 Insert with Less Comparisons

- Define a function insert that inserts into a binary search tree, using only one comparison per node. Use the same idea as insert2.
- Show that your function is equal to me on binary search trees. Hint: You may need an auxiliary lemma of the form:

\[
\begin{align*}
\text{let t = list where } &\exists x, y < x \rightarrow \exists y, z < x \rightarrow \text{insert t x} = \text{insert t y} = \text{insert t z} = \text{Some t} \\
\text{fun insert } &\text{ (a:tree) = a \cdot \text{option} = a \cdot \text{tree} = a \cdot \text{tree}} \\
\text{lemma insert2: } &\text{insert t x None t = insert t x t} \\
\end{align*}
\]

Exercise 3.2 Height-Preserving In-Order Join
Functional Data Structures
Exercise Sheet 3

Exercise 3.1 Insert with Less Comparisons
- Define a function \( \text{insert} \) that inserts into a binary search tree, using only one comparison per node. Use the same idea as \( \text{ins2} \).
- Show that your function is equal to \( \text{ins} \) on binary search trees. Hint: You may need an auxiliary lemma of the form:

\[
\text{ins2} : \langle \text{insert} \rangle \quad \langle \text{insert} \rangle = \text{ins}
\]

```haskell
fun ins2 : "a:disorder ⇒ 'a option ⇒ 'a tree ⇒ 'a tree"
lemma ins2:None : "tht t ⇒ ins2 x None t = ins x t"
```

Exercise 3.2 Height-Preserving In-Order Join
proof [prove]
gol (1 subgoal):
1. bst (fold ins 1 ()

proof [prove]
gol (1 subgoal):
1. As 1. bst (fold ins 1 () — bst (fold ins 1 , a, 0)

proof [prove]
gol (1 subgoal):
1. As 1. bst (fold ins 1 ()

proof [prove]
gol (1 subgoal):
1. As 1. bst (fold ins 1 () — bst (fold ins 1 , a, 0)
unfolding \text{mk\_tree\_def} by \{simp add: aux\}

lemmas aux3: \text{"bst\_i \Rightarrow mk\_tree\_def (fold\_time\_1 \_i) = mk\_tree\_i \_i \_i \_i\"} by \{ injection \text{null\_tree\_def} \}

lemmas aux4: \text{\"bst\_tree\_mk\_tree\_def = \text{mk\_tree\_def}\"}

unfolding \text{mk\_tree\_def} by \{simp add: aux\}

end

\begin{proof}
\begin{goal eclams:1}
\item \text{\texttt{distinct (bst\_sort\_1)}}
\end{goal}
\end{proof}

proof (cases)
\begin{goal eclams:1}
\item \text{distinct (bst\_sort\_1)}
\end{goal}
\end{proof}

lemmas aux: \text{\texttt{\"distinct (bst\_sort\_1)\"}}

unfolding \text{bst\_sort\_def}

by \{ simp add: \text{mk\_tree\_def} \}

lemmas aux2: \text{\texttt{\"sorted (bst\_sort\_1)\"}}

unfolding \text{bst\_sort\_def}

by \{ simp add: \text{mk\_tree\_def} \}

find theorems \text{inorder sorted}

find theorems \text{\"inorder\"}

\text{\"sorted\"}

found 1 theorem(s):
\item \text{Tree.\texttt{inorder\_class.bst\_eq\_impl\_sorted.bst\_eq\_it\_bst \Rightarrow sorted (inorder\_it\_bst)\"}}
\end{proof}

\begin{proof}
\begin{goal eclams:1}
\item \text{\texttt{sorted (bst\_sort\_1)}}
\end{goal}
\end{proof}
def `mk_btree` = "a btree list \Rightarrow \{ a tree \}"
def `mk_btree` = "b btree list \Rightarrow \{ b tree \}"
def `mk_tree` = "c btree list \Rightarrow \{ c tree \}"
def `mk_tree` = "d btree list \Rightarrow \{ d tree \}"
def `mk_tree` = "e btree list \Rightarrow \{ e tree \}"
def `mk_tree` = "f btree list \Rightarrow \{ f tree \}"
def `mk_tree` = "g btree list \Rightarrow \{ g tree \}"
def `mk_tree` = "h btree list \Rightarrow \{ h tree \}"
def `mk_tree` = "i btree list \Rightarrow \{ i tree \}"
def `mk_tree` = "j btree list \Rightarrow \{ j tree \}"
def `mk_tree` = "k btree list \Rightarrow \{ k tree \}"
def `mk_tree` = "l btree list \Rightarrow \{ l tree \}"
def `mk_tree` = "m btree list \Rightarrow \{ m tree \}"
def `mk_tree` = "n btree list \Rightarrow \{ n tree \}"
def `mk_tree` = "o btree list \Rightarrow \{ o tree \}"
def `mk_tree` = "p btree list \Rightarrow \{ p tree \}"
def `mk_tree` = "q btree list \Rightarrow \{ q tree \}"
def `mk_tree` = "r btree list \Rightarrow \{ r tree \}"
def `mk_tree` = "s btree list \Rightarrow \{ s tree \}"
def `mk_tree` = "t btree list \Rightarrow \{ t tree \}"
def `mk_tree` = "u btree list \Rightarrow \{ u tree \}"
def `mk_tree` = "v btree list \Rightarrow \{ v tree \}"
def `mk_tree` = "w btree list \Rightarrow \{ w tree \}"
def `mk_tree` = "x btree list \Rightarrow \{ x tree \}"
def `mk_tree` = "y btree list \Rightarrow \{ y tree \}"
def `mk_tree` = "z btree list \Rightarrow \{ z tree \}"

Homework 3 BSTs with Duplicates

Submission until Friday, May 19, 11:59pm.

- Have a look at `bst_eq` in `/src/IOLib/Tree`, which defines BSTs with duplicate elements.
- Warming: Show that `ins` and `ins` are also correct for `bst_eq`,
  `bst_eq` = `bst_eq` (ins s t) \Rightarrow `bst_eq` (ins s t)
  `bst_eq` = `bst_eq` (ins s t) \Rightarrow `bst_eq` (ins s t)

- Define a function `ins_eq` to insert into a BST with duplicates.

```
fun ins_eq = "\lambda s t. s tree = t tree"
```

- Show that `ins_eq` preserves the invariant `bst_eq`
  `bst_eq` = `bst_eq` (ins s t) \Rightarrow `bst_eq` (ins s t)

- Define a function `count_tree` to count how many a given element occurs in a tree.

```
fun count = "\lambda t. \begin{cases} 0 & \text{if} \# t = 0 \\ 1 + \text{count} (\text{left} t) + \text{count} (\text{right} t) & \text{if} \# t > 0 \end{cases}"
```
fun ins_eq :: "'a => 'a => 'a => 'a"

- Show that ins_eq preserves the invariant bot_eq

lemma bot_eq [ins_eq]: "bot_eq t ⇒ bot_eq (ins_eq x t)"

- Define a function count_tree to count how often a given element occurs in a tree

fun count_tree :: "'a ⇒ 'a tree ⇒ nat"

- Show that the ins_eq function inserts the desired element, and does not affect other elements.

lemma "count_tree x (ins_eq y t) = Suc (count_tree x t)"
lemma "∀y. count_tree y (ins_eq y t) = count_tree y t"

The next exercise is a bonus exercise, yielding bonus points. Bonus points count as achieved points, but not for the maximum achievable points, when computing the percentage of the achieved homework points.

- Bonus [3p]: Use BSTs with duplicates to sort a list (cf. Exercise 3). Prove that the resulting list is sorted, and contains exactly the same number of each element as the original list. Hint: Use a count function for lists, and relate it with the count_tree-function for trees.

Define a function count_tree to count how often a given element occurs in a tree

fun count_tree :: "'a ⇒ 'a tree ⇒ nat"

- Show that the ins_eq function inserts the desired element, and does not affect other elements.

lemma "count_tree x (ins_eq y t) = Suc (count_tree x t)"
lemma "∀y. count_tree y (ins_eq y t) = count_tree y t"

The next exercise is a bonus exercise, yielding bonus points. Bonus points count as achieved points, but not for the maximum achievable points, when computing the percentage of the achieved homework points.

- Bonus [3p]: Use BSTs with duplicates to sort a list (cf. Exercise 3). Prove that the resulting list is sorted, and contains exactly the same number of each element as the original list. Hint: Use a count function for lists, and relate it with the count_tree-function for trees.