


Script generated by TTT

Title: FDS (15.06.2018)

Date: Fri Jun 15 08:31:06 CEST 2018

Duration: 70:32 min

Pages: 76



Just join

Given $join :: tree \Rightarrow 'a \Rightarrow tree \Rightarrow tree$
(where $tree$ abbreviates $('a, 'b) tree$), implement

$split :: tree \Rightarrow 'a \Rightarrow tree \times bool \times tree$
 $insert :: 'a \Rightarrow tree \Rightarrow tree$
 $union :: tree \Rightarrow tree \Rightarrow tree$
 $join2 :: tree \Rightarrow tree \Rightarrow tree$
 $delete :: 'a \Rightarrow tree \Rightarrow tree$
 $inter :: tree \Rightarrow tree \Rightarrow tree$
 $diff :: tree \Rightarrow tree \Rightarrow tree$

131



Just join

Given $join :: tree \Rightarrow 'a \Rightarrow tree \Rightarrow tree$
(where $tree$ abbreviates $('a, 'b) tree$), implement

$split :: tree \Rightarrow 'a \Rightarrow tree \times bool \times tree$

$insert :: 'a \Rightarrow tree \Rightarrow tree$

$union :: tree \Rightarrow tree \Rightarrow tree$

$join2 :: tree \Rightarrow tree \Rightarrow tree$

$delete :: 'a \Rightarrow tree \Rightarrow tree$

$inter :: tree \Rightarrow tree \Rightarrow tree$

$diff :: tree \Rightarrow tree \Rightarrow tree$



13 Union, Intersection, Difference on BSTs

Correctness

join for Red-Black Trees



Just *join*

Given $join :: tree \Rightarrow 'a \Rightarrow tree \Rightarrow tree$
 (where $tree$ abbreviates $('a, 'b) tree$), implement

$split :: tree \Rightarrow 'a \Rightarrow tree \times bool \times tree$
 $insert :: 'a \Rightarrow tree \Rightarrow tree$
 $union :: tree \Rightarrow tree \Rightarrow tree$
 $join2 :: tree \Rightarrow tree \Rightarrow tree$
 $delete :: 'a \Rightarrow tree \Rightarrow tree$
 $inter :: tree \Rightarrow tree \Rightarrow tree$
 $diff :: tree \Rightarrow tree \Rightarrow tree$

131



Specification of *join*

- $set_tree (join\ l\ a\ r) = set_tree\ l \cup \{a\} \cup set_tree\ r$

133



Specification of *join*

- $set_tree (join\ l\ a\ r) = set_tree\ l \cup \{a\} \cup set_tree\ r$
- $\llbracket bst\ l; bst\ r; \forall x \in set_tree\ l. x < a; \forall y \in set_tree\ r. a < y \rrbracket \implies bst (join\ l\ a\ r)$

133



Specification of *join* and *inv*

- $set_tree (join\ l\ a\ r) = set_tree\ l \cup \{a\} \cup set_tree\ r$
- $\llbracket bst\ l; bst\ r; \forall x \in set_tree\ l. x < a; \forall y \in set_tree\ r. a < y \rrbracket \implies bst (join\ l\ a\ r)$

Also required: structural invariant *inv*:

133



Specification of *join* and *inv*

- $set_tree (join\ l\ a\ r) = set_tree\ l \cup \{a\} \cup set_tree\ r$
- $\llbracket bst\ l; bst\ r; \forall x \in set_tree\ l. x < a; \forall y \in set_tree\ r. a < y \rrbracket \implies bst\ (join\ l\ a\ r)$

Also required: structural invariant *inv*:

- $inv\ \langle \rangle$

133



Specification of *join* and *inv*

- $set_tree (join\ l\ a\ r) = set_tree\ l \cup \{a\} \cup set_tree\ r$
- $\llbracket bst\ l; bst\ r; \forall x \in set_tree\ l. x < a; \forall y \in set_tree\ r. a < y \rrbracket \implies bst\ (join\ l\ a\ r)$

Also required: structural invariant *inv*:

- $inv\ \langle \rangle$
- $inv\ \langle h, l, x, r \rangle \implies inv\ l \wedge inv\ r$

133



Specification of *join* and *inv*

- $set_tree (join\ l\ a\ r) = set_tree\ l \cup \{a\} \cup set_tree\ r$
- $\llbracket bst\ l; bst\ r; \forall x \in set_tree\ l. x < a; \forall y \in set_tree\ r. a < y \rrbracket \implies bst\ (join\ l\ a\ r)$

Also required: structural invariant *inv*:

- $inv\ \langle \rangle$
- $inv\ \langle h, l, x, r \rangle \implies inv\ l \wedge inv\ r$
- $\llbracket inv\ l; inv\ r \rrbracket \implies inv\ (join\ l\ k\ r)$

133



Specification of *join* and *inv*

- $set_tree (join\ l\ a\ r) = set_tree\ l \cup \{a\} \cup set_tree\ r$
- $\llbracket bst\ l; bst\ r; \forall x \in set_tree\ l. x < a; \forall y \in set_tree\ r. a < y \rrbracket \implies bst\ (join\ l\ a\ r)$

Also required: structural invariant *inv*:

- $inv\ \langle \rangle$
- $inv\ \langle h, l, x, r \rangle \implies inv\ l \wedge inv\ r$
- $\llbracket inv\ l; inv\ r \rrbracket \implies inv\ (join\ l\ k\ r)$

Locale context for def of *union* etc

133



Specification of *union*, *inter*, *diff*

ADT/Locale *Set2* = extension of locale *Set* with

- *union*, *inter*, *diff* :: 's ⇒ 's ⇒ 's

134



Specification of *union*, *inter*, *diff*

ADT/Locale *Set2* = extension of locale *Set* with

- *union*, *inter*, *diff* :: 's ⇒ 's ⇒ 's
- $\llbracket \text{invar } s_1; \text{invar } s_2 \rrbracket$
 $\implies \text{set } (\text{union } s_1 \ s_2) = \text{set } s_1 \cup \text{set } s_2$

134



Specification of *union*, *inter*, *diff*

ADT/Locale *Set2* = extension of locale *Set* with

- *union*, *inter*, *diff* :: 's ⇒ 's ⇒ 's
- $\llbracket \text{invar } s_1; \text{invar } s_2 \rrbracket$
 $\implies \text{set } (\text{union } s_1 \ s_2) = \text{set } s_1 \cup \text{set } s_2$
- $\llbracket \text{invar } s_1; \text{invar } s_2 \rrbracket \implies \text{invar } (\text{union } s_1 \ s_2)$

134



Specification of *union*, *inter*, *diff*

ADT/Locale *Set2* = extension of locale *Set* with

- *union*, *inter*, *diff* :: 's ⇒ 's ⇒ 's
- $\llbracket \text{invar } s_1; \text{invar } s_2 \rrbracket$
 $\implies \text{set } (\text{union } s_1 \ s_2) = \text{set } s_1 \cup \text{set } s_2$
- $\llbracket \text{invar } s_1; \text{invar } s_2 \rrbracket \implies \text{invar } (\text{union } s_1 \ s_2)$
- ... *inter* ...
- ... *diff* ...

We focus on *union*.

See `Thys/Set_Specs.thy`

134



Correctness lemmas for *union* etc code

In the context of *join* specification:

135



Specification of *union*, *inter*, *diff*

ADT/Locale *Set2* = extension of locale *Set* with

- $union, inter, diff :: 's \Rightarrow 's \Rightarrow 's$
- $\llbracket invar\ s_1; invar\ s_2 \rrbracket \implies set\ (union\ s_1\ s_2) = set\ s_1 \cup set\ s_2$
- $\llbracket invar\ s_1; invar\ s_2 \rrbracket \implies invar\ (union\ s_1\ s_2)$
- ... *inter* ...
- ... *diff* ...

We focus on *union*.

134



Correctness lemmas for *union* etc code

In the context of *join* specification:

- $bst\ t_2 \implies set_tree\ (union\ t_1\ t_2) = set_tree\ t_1 \cup set_tree\ t_2$
- $\llbracket bst\ t_1; bst\ t_2 \rrbracket \implies bst\ (union\ t_1\ t_2)$

135



Correctness lemmas for *union* etc code

In the context of *join* specification:

- $bst\ t_2 \implies set_tree\ (union\ t_1\ t_2) = set_tree\ t_1 \cup set_tree\ t_2$
- $\llbracket bst\ t_1; bst\ t_2 \rrbracket \implies bst\ (union\ t_1\ t_2)$
- $\llbracket inv\ t_1; inv\ t_2 \rrbracket \implies inv\ (union\ t_1\ t_2)$

135



Correctness lemmas for *union* etc code

In the context of *join* specification:

- $bst\ t_2 \implies$
 $set_tree\ (union\ t_1\ t_2) = set_tree\ t_1 \cup set_tree\ t_2$
- $\llbracket bst\ t_1; bst\ t_2 \rrbracket \implies bst\ (union\ t_1\ t_2)$
- $\llbracket inv\ t_1; inv\ t_2 \rrbracket \implies inv\ (union\ t_1\ t_2)$

Proofs automatic

135



Correctness lemmas for *union* etc code

In the context of *join* specification:

- $bst\ t_2 \implies$
 $set_tree\ (union\ t_1\ t_2) = set_tree\ t_1 \cup set_tree\ t_2$
- $\llbracket bst\ t_1; bst\ t_2 \rrbracket \implies bst\ (union\ t_1\ t_2)$
- $\llbracket inv\ t_1; inv\ t_2 \rrbracket \implies inv\ (union\ t_1\ t_2)$

Proofs automatic (more complex for *inter* and *diff*)

Implementation of locale *Set2*:

interpretation *Set2* **where** $union = union \dots$

135



Correctness lemmas for *union* etc code

In the context of *join* specification:

- $bst\ t_2 \implies$
 $set_tree\ (union\ t_1\ t_2) = set_tree\ t_1 \cup set_tree\ t_2$
- $\llbracket bst\ t_1; bst\ t_2 \rrbracket \implies bst\ (union\ t_1\ t_2)$
- $\llbracket inv\ t_1; inv\ t_2 \rrbracket \implies inv\ (union\ t_1\ t_2)$

Proofs automatic (more complex for *inter* and *diff*)

Implementation of locale *Set2*:

interpretation *Set2* **where** $union = union \dots$

and $set = set_tree$

135



Correctness lemmas for *union* etc code

In the context of *join* specification:

- $bst\ t_2 \implies$
 $set_tree\ (union\ t_1\ t_2) = set_tree\ t_1 \cup set_tree\ t_2$
- $\llbracket bst\ t_1; bst\ t_2 \rrbracket \implies bst\ (union\ t_1\ t_2)$
- $\llbracket inv\ t_1; inv\ t_2 \rrbracket \implies inv\ (union\ t_1\ t_2)$

Proofs automatic (more complex for *inter* and *diff*)

Implementation of locale *Set2*:

interpretation *Set2* **where** $union = union \dots$

and $set = set_tree$ **and** $invar =$

135



Correctness lemmas for *union* etc code

In the context of *join* specification:

- $bst\ t_2 \implies$
 $set_tree\ (union\ t_1\ t_2) = set_tree\ t_1 \cup set_tree\ t_2$
- $\llbracket bst\ t_1; bst\ t_2 \rrbracket \implies bst\ (union\ t_1\ t_2)$
- $\llbracket inv\ t_1; inv\ t_2 \rrbracket \implies inv\ (union\ t_1\ t_2)$

Proofs automatic (more complex for *inter* and *diff*)

Implementation of locale *Set2*:

interpretation *Set2* **where** $union = union \dots$

135



Specification of *union*, *inter*, *diff*

ADT/Locale *Set2* = extension of locale *Set* with

- $union, inter, diff :: 's \Rightarrow 's \Rightarrow 's$
- $\llbracket invar\ s_1; invar\ s_2 \rrbracket$
 $\implies set\ (union\ s_1\ s_2) = set\ s_1 \cup set\ s_2$
- $\llbracket invar\ s_1; invar\ s_2 \rrbracket \implies invar\ (union\ s_1\ s_2)$
- $\dots inter \dots$
- $\dots diff \dots$

134



Specification of *union*, *inter*, *diff*

ADT/Locale *Set2* = extension of locale *Set* with

- $union, inter, diff :: 's \Rightarrow 's \Rightarrow 's$
- $\llbracket invar\ s_1; invar\ s_2 \rrbracket$
 $\implies set\ (union\ s_1\ s_2) = set\ s_1 \cup set\ s_2$



Correctness lemmas for *union* etc code

In the context of *join* specification:

- $bst\ t_2 \implies$
 $set_tree\ (union\ t_1\ t_2) = set_tree\ t_1 \cup set_tree\ t_2$
- $\llbracket bst\ t_1; bst\ t_2 \rrbracket \implies bst\ (union\ t_1\ t_2)$
- $\llbracket inv\ t_1; inv\ t_2 \rrbracket \implies inv\ (union\ t_1\ t_2)$

Proofs automatic (more complex for *inter* and *diff*)

Implementation of locale *Set2*:

interpretation *Set2* **where** $union = union \dots$
and $set = set_tree$ **and** $invar = (\lambda t. bst\ t \wedge inv\ t)$

134

135



Thys/Set2_Join.thy

136



13 Union, Intersection, Difference on BSTs

Correctness

join for Red-Black Trees

137



join l a r — The idea

138



join l a r — The idea

Assume l is “smaller” than r :

138



join l a r — The idea

Assume l is “smaller” than r :

- Descend along the left spine of r
until you find a subtree t of the same “size” as l .

138



join l a r — The idea

Assume l is “smaller” than r :

- Descend along the left spine of r
until you find a subtree t of the same “size” as l .
- Replace t by $\langle l, a, t \rangle$.

138



join l x r =
 (if *bheight r* < *bheight l*
 then *paint Black (joinR l x r)*
 else if *bheight l* < *bheight r*
 then *paint Black (joinL l x r)* else *B l x r*)

139



join l x r =
 (if *bheight r* < *bheight l*
 then *paint Black (joinR l x r)*
 else if *bheight l* < *bheight r*
 then *paint Black (joinL l x r)* else *B l x r*)

Need to store black height in each node
 for logarithmic complexity

139



Thys/Set2_Join_RBT.thy

140



```

join l x r =
  (if bheight r < bheight l
   then paint Black (joinR l x r)
   else if bheight l < bheight r
        then paint Black (joinL l x r) else B l x r)

```

Need to store black height in each node
for logarithmic complexity

139



join l a r — The idea

Assume l is “smaller” than r :

- Descend along the left spine of r
until you find a subtree t of the same “size” as l .
- Replace t by $\langle l, a, t \rangle$.

138



Adjusting colors

$balL, balR :: 'a\ rbt \Rightarrow 'a \Rightarrow 'a\ rbt \Rightarrow 'a\ rbt$

- Combine arguments $l\ a\ r$ into tree, ideally $\langle l, a, r \rangle$
- Treat invariant violation **Red-Red** in l/r

$$\begin{aligned}
 balL\ (R\ (R\ t_1\ a_1\ t_2)\ a_2\ t_3)\ a_3\ t_4 \\
 &= R\ (B\ t_1\ a_1\ t_2)\ a_2\ (B\ t_3\ a_3\ t_4)
 \end{aligned}$$

109



Adjusting colors

$balil, balir :: 'a\ rbt \Rightarrow 'a \Rightarrow 'a\ rbt \Rightarrow 'a\ rbt$

- Combine arguments $l\ a\ r$ into tree, ideally $\langle l, a, r \rangle$

- Treat invariant violation **Red-Red** in l/r

$balil\ (R\ (R\ t_1\ a_1\ t_2)\ a_2\ t_3)\ a_3\ t_4$

$= R\ (B\ t_1\ a_1\ t_2)\ a_2\ (B\ t_3\ a_3\ t_4)$

$balil\ (R\ t_1\ a_1\ (R\ t_2\ a_2\ t_3))\ a_3\ t_4$

$= R\ (B\ t_1\ a_1\ t_2)\ a_2\ (B\ t_3\ a_3\ t_4)$

- Principle: replace **Red-Red** by **Red-Black**

109



Literature

The idea of “just *join*”:

Stephen Adams. *Efficient Sets — A Balancing Act*.

J. Functional Programming, volume 3, number 4, 1993.

141



Literature

The idea of “just *join*”:

Stephen Adams. *Efficient Sets — A Balancing Act*.

J. Functional Programming, volume 3, number 4, 1993.

The precise analysis:

Guy E. Blelloch, D. Ferizovic, Y. Sun.

Just Join for Parallel Ordered Sets.

ACM Symposium on Parallelism in Algorithms and Architectures 2016.

141



`Thys/Set2_Join_RBT.thy`

140



Uniform *tree* type

129



Thys/Trie1
Thys/Trie2

143



Trie

[Fredkin, CACM 1960]

Name: *reTRIEval*

144



Trie

[Fredkin, CACM 1960]

Name: *reTRIEval*

- Tries are search trees indexed by lists.

144



Trie

[Fredkin, CACM 1960]

Name: *reTRIEval*

- Tries are search trees indexed by lists.
- Each node maps the next element in the list to a subtrie.

144



Trie

[Fredkin, CACM 1960]

Name: *reTRIEval*

- Tries are search trees indexed by lists.
- Each node maps the next element in the list to a subtrie.

144



Trie

[Fredkin, CACM 1960]

Name: *reTRIEval*

- Tries are search trees indexed by lists.
- Each node maps the next element in the list to a subtrie.
- For simplicity we consider only binary tries:
datatype *trie* = *Leaf* | *Node bool (trie × trie)*

144



Patricia trie

[Morrison, JACM 1968]

Sequences of edges without branching are compressed

145



Patricia trie

[Morrison, JACM 1968]

Sequences of edges without branching are compressed

145



Patricia trie

[Morrison, JACM 1968]

Sequences of edges without branching are compressed

```
datatype ptrie = LeafP  
  | NodeP (bool list) bool (ptrie × ptrie)
```

145



Patricia trie

[Morrison, JACM 1968]

Sequences of edges without branching are compressed

```
datatype ptrie = LeafP  
  | NodeP (bool list) bool (ptrie × ptrie)
```

Name: PATRICIA = *Practical Algorithm To Retrieve Information Coded in Alphanumeric*

145



From trie to Patricia trie

and beyond

146



From trie to Patricia trie

and beyond

An exercise in data refinement

146



- 14 Tries and Patricia Tries
 - Data Refinement Basics
 - Trie
 - Patricia Trie
 - Patricia Trie for Words

147



Data refinement

How to implement an abstract type A by a concrete (“refined”) type C with invariant $inv_C :: C \Rightarrow bool$:

148



Data refinement

How to implement an abstract type A by a concrete (“refined”) type C with invariant $inv_C :: C \Rightarrow bool$:

- Define an abstraction function $\alpha_C :: C \Rightarrow A$

148



Data refinement

How to implement an abstract type A by a concrete (“refined”) type C with invariant $inv_C :: C \Rightarrow bool$:

- Define an abstraction function $\alpha_C :: C \Rightarrow A$
- Implement each interface function f_A on A by some f_C on C

148



Data refinement

How to implement an abstract type A by a concrete (“refined”) type C with invariant $inv_C :: C \Rightarrow bool$:

- Define an abstraction function $\alpha_C :: C \Rightarrow A$
- Implement each interface function f_A on A by some f_C on C
- Prove that each f_C implements f_A :
 $inv_C t \Longrightarrow \alpha_C (f_C t) = f_A (\alpha_C t)$

148



Data refinement

How to implement an abstract type A by a concrete (“refined”) type C with invariant $inv_C :: C \Rightarrow bool$:

- Define an abstraction function $\alpha_C :: C \Rightarrow A$
- Implement each interface function f_A on A by some f_C on C
- Prove that each f_C implements f_A :
 $inv_C t \Longrightarrow \alpha_C (f_C t) = f_A (\alpha_C t)$
and preserves the invariant:
 $inv_C t \Longrightarrow inv_C (f_C t)$

148



Data refinement

How to implement an abstract type A by a concrete (“refined”) type C with invariant $inv_C :: C \Rightarrow bool$:

- Define an abstraction function $\alpha_C :: C \Rightarrow A$
- Implement each interface function f_A on A by some f_C on C
- Prove that each f_C implements f_A :
 $inv_C t \Longrightarrow \alpha_C (f_C t) = f_A (\alpha_C t)$
and preserves the invariant:
 $inv_C t \Longrightarrow inv_C (f_C t)$

The f_s may take more parameters
and may return values not of type A/C .

148



Data refinement

Multiple refinement steps can help
to reduce the complexity of correctness proofs

149



- 14 Tries and Patricia Tries
 - Data Refinement Basics
 - Trie
 - Patricia Trie
 - Patricia Trie for Words

150



Correctness of *trie* w.r.t. *set*

$$isin (insert\ as\ t)\ bs = (as = bs \vee isin\ t\ bs)$$

Note: $isin :: trie \Rightarrow bool\ list \Rightarrow bool$
doubles as abstraction function

154



Correctness of *trie* w.r.t. *set*

$$isin (insert\ as\ t)\ bs = (as = bs \vee isin\ t\ bs)$$

Note: $isin :: trie \Rightarrow bool\ list \Rightarrow bool$
doubles as abstraction function
because $'a \Rightarrow bool \approx 'a\ set$

154



Patricia trie

```
datatype ptrie = LeafP  
| NodeP (bool list) bool (ptrie × ptrie)
```

156



$isinP$

$isinP\ LeafP\ ks = False$

157



Splitting lists

```
split xs ys = (zs, xs', ys')  
iff zs is the longest common prefix of xs and ys  
and xs'/ys' is the remainder of xs/ys
```

158



Trie *insert*

$insert []\ Leaf = Node\ True\ (Leaf,\ Leaf)$

153



$abs_ptrie :: ptrie \Rightarrow trie$

$abs_ptrie \text{ LeafP} = \text{Leaf}$

$abs_ptrie (\text{NodeP } ps \ b \ (l, r)) =$
 $prefix_trie \ ps \ (\text{Node } b \ (abs_ptrie \ l, abs_ptrie \ r))$

$prefix_trie :: \text{bool list} \Rightarrow trie \Rightarrow trie$

160



Correctness of $ptrie$ w.r.t. $trie$

$isinP \ t \ ks = isin \ (abs_ptrie \ t) \ ks$

161



Correctness of $ptrie$ w.r.t. $trie$

$isinP \ t \ ks = isin \ (abs_ptrie \ t) \ ks$

$abs_ptrie \ (\text{insertP } ks \ t) = \text{insert } ks \ (abs_ptrie \ t)$

161



14 Tries and Patricia Tries

Data Refinement Basics

Trie

Patricia Trie

Patricia Trie for Words

162



14 Tries and Patricia Tries

Data Refinement Basics

Trie

Patricia Trie

Patricia Trie for Words