

Script generated by TTT

Title: FDS (23.06.2017)

Date: Fri Jun 23 08:30:31 CEST 2017

Duration: 98:07 min

Pages: 53



⑧ Unbalanced BST

⑨ 2-3 Trees

⑩ Red-Black Trees

⑪ Tries and Patricia Tries

93



⑧ Unbalanced BST

⑨ 2-3 Trees

⑩ Red-Black Trees

⑪ Tries and Patricia Tries

Thys/Trie1

Thys/Trie2



Trie

Name: *reTRIEval*



Trie

Name: *reTRIEval*

- Tries are search trees indexed by lists.

96

96



Trie

Name: *reTRIEval*

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



Trie

Name: *reTRIEval*

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

96

96



Patricia trie

Sequences of edges without branching are compressed



Patricia trie

Sequences of edges without branching are compressed

97

97



Patricia trie

Sequences of edges without branching are compressed

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



Patricia trie

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*

97

97



From trie to Patricia trie

and beyond



11 Tries and Patricia Tries

Tries and Patricia Tries

Data Refinement Basics

Trie

Patricia Trie

Patricia Trie for Words

98

99



Data refinement

How to implement an abstract type A by a concrete (“refined”) type C with invariant $\text{inv}_C :: C \Rightarrow \text{bool}$:



Data refinement

How to implement an abstract type A by a concrete (“refined”) type C with invariant $\text{inv}_C :: C \Rightarrow \text{bool}$:

- Define an abstraction function $\text{abs}_C :: C \Rightarrow A$
- Implement each interface function f_A on A by some f_C on C

100

100



Data refinement

How to implement an abstract type A by a concrete (“refined”) type C with invariant $\text{inv}_C :: C \Rightarrow \text{bool}$:

- Define an abstraction function $\text{abs}_C :: C \Rightarrow A$
- Implement each interface function f_A on A by some f_C on C
- Prove that each f_C implements f_A :
 $\text{inv}_C t \implies \text{abs}_C(f_C t) = f_A(\text{abs}_C t)$



Data refinement

How to implement an abstract type A by a concrete (“refined”) type C with invariant $\text{inv}_C :: C \Rightarrow \text{bool}$:

- Define an abstraction function $\text{abs}_C :: C \Rightarrow A$
- Implement each interface function f_A on A by some f_C on C
- Prove that each f_C implements f_A :
 $\text{inv}_C t \implies \text{abs}_C(f_C t) = f_A(\text{abs}_C t)$
 and preserves the invariant:
 $\text{inv}_C t \implies \text{inv}_C(f_C t)$

100

100



Data refinement

How to implement an abstract type A by a concrete (“refined”) type C with invariant $\text{inv}_C :: C \Rightarrow \text{bool}$:

- Define an abstraction function $\text{abs}_C :: C \Rightarrow A$
- Implement each interface function f_A on A by some f_C on C
- Prove that each f_C implements f_A :
 $\text{inv}_C t \implies \text{abs}_C(f_C t) = f_A(\text{abs}_C t)$
 and preserves the invariant:
 $\text{inv}_C t \implies \text{inv}_C(f_C t)$

The f s may take more parameters



Data refinement

How to implement an abstract type A by a concrete (“refined”) type C with invariant $\text{inv}_C :: C \Rightarrow \text{bool}$:

- Define an abstraction function $\text{abs}_C :: C \Rightarrow A$
- Implement each interface function f_A on A by some f_C on C
- Prove that each f_C implements f_A :
 $\text{inv}_C t \implies \text{abs}_C(f_C t) = f_A(\text{abs}_C t)$
 and preserves the invariant:
 $\text{inv}_C t \implies \text{inv}_C(f_C t)$

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

100

100



Data refinement

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



Trie

datatype $trie = Leaf \mid Node\ bool\ (trie \times trie)$

101

103



Trie *isin*

isin Leaf ks = False

*isin (Node b (l, r)) ks =
(case ks of
 [] => b
 | k # x => isin (if k then r else l) x)*



Trie *insert*

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

104

105



Trie insert

$\text{insert } [] \text{ Leaf} = \text{Node True (Leaf, Leaf)}$
 $\text{insert } [] (\text{Node } b \text{ lr}) = \text{Node True lr}$

$\text{insert } (k \# ks) \text{ Leaf} =$
 Node False
 $(\text{if } k \text{ then (Leaf, insert } ks \text{ Leaf)}$
 $\text{else (insert } ks \text{ Leaf, Leaf)})$

$\text{insert } (k \# ks) (\text{Node } b (l, r)) =$
 $\text{Node } b (\text{if } k \text{ then (l, insert } ks \text{ r)} \text{ else (insert } ks \text{ l, r)})$



Correctness of trie w.r.t. set

$\text{isin } (\text{insert as } t) \text{ bs} = (\text{as} = \text{bs} \vee \text{isin } t \text{ bs})$

105

106



Trie insert

$\text{insert } [] \text{ Leaf} = \text{Node True (Leaf, Leaf)}$
 $\text{insert } [] (\text{Node } b \text{ lr}) = \text{Node True lr}$

$\text{insert } (k \# ks) \text{ Leaf} =$
 Node False
 $(\text{if } k \text{ then (Leaf, insert } ks \text{ Leaf)}$
 $\text{else (insert } ks \text{ Leaf, Leaf)})$

$\text{insert } (k \# ks) (\text{Node } b (l, r)) =$
 $\text{Node } b (\text{if } k \text{ then (l, insert } ks \text{ r)} \text{ else (insert } ks \text{ l, r)})$



11 Tries and Patricia Tries

Tries and Patricia Tries

Data Refinement Basics

Trie

Patricia Trie

Patricia Trie for Words

105

102



Patricia trie

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



Splitting lists

$$\text{split } xs \text{ } ys = (zs, \text{ } xs', \text{ } ys')$$

108

110



Splitting lists

$$\text{split } xs \text{ } ys = (zs, \text{ } xs', \text{ } ys')$$

iff zs is the longest common prefix of xs and ys



Splitting lists

$$\text{split } xs \text{ } ys = (zs, \text{ } xs', \text{ } ys')$$

iff zs is the longest common prefix of xs and ys
and xs'/ys' is the remainder of xs/ys

110

110



insertP



Splitting lists

split xs ys = (zs, xs', ys')

iff zs is the longest common prefix of xs and ys

111

110



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



abs_ptrie :: ptrie ⇒ trie

abs_ptrie LeafP = Leaf

abs_ptrie (NodeP ps b (l, r)) =
prefix_trie ps (Node b (abs_ptrie l, abs_ptrie r))

110

112



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

$\text{isinP } t \text{ ks} = \text{isin } (\text{abs_ptrie } t) \text{ ks}$



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

$\text{isinP } t \text{ ks} = \text{isin } (\text{abs_ptrie } t) \text{ ks}$

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

113

113



11 Tries and Patricia Tries

Tries and Patricia Tries

Data Refinement Basics

Trie

Patricia Trie

Patricia Trie for Words



Patricia trie for words

datatype $\text{ptrieW} = \text{LeafW}$
| $\text{NodeW } (32 \text{ word}) \text{ (32 word) bool}$
 $(\text{ptrieW} \times \text{ptrieW})$

114

115



Patricia trie for words

```
datatype ptrieW = LeafW
  | NodeW (32 word) (32 word) bool
    (ptrieW × ptrieW)
```

to_bl :: 'a word \Rightarrow bool list



is_in_W

is_in_W LeafW k = False

115



is_in_W

is_in_W LeafW k = False

```
is_in_W (NodeW p m b (l, r)) k =
(if b then k = p
 else if match_prefix p m k
  then is_in_W (if go_left m k then l else r) k
  else False)
```



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

116

116

117

inv_ptrieW t \implies
is_in_W t k = is_inL (abs_ptrieW t) (rev (to_bl k))



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

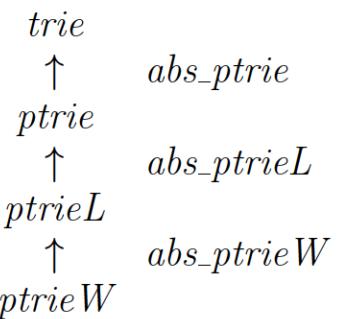
$inv_ptrieW t \implies$

$isinW t k = isinL (abs_ptrieW t) (rev (to_bl k))$

Proof uses intermediate version of Patricia trie for lists where nodes contain complete prefixes ($isinL$).



Refinement hierarchy

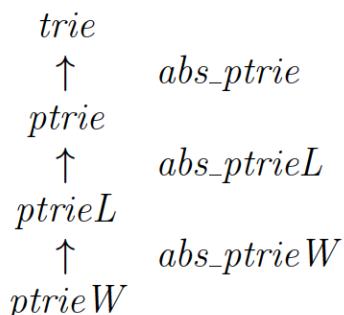


117

118



Refinement hierarchy



where $ptrieL = ptrie$



$inv_ptrieW :: ptrieW \Rightarrow bool$

118

119



Refinement hierarchy

```

trie
  ↑    abs_ptrie
ptrie
  ↑    abs_ptrieL
ptrieL
  ↑    abs_ptrieW
ptrieW

```



inv_ptrieW :: ptrieW \Rightarrow bool

inv_ptrieW LeafW = True

inv_ptrieW (NodeW p m b (l, r)) =
 $(inv_pref\ p\ m \wedge inv_mask\ m \wedge inv_ptrieW\ l \wedge$
 $inv_ptrieW\ r)$

inv_pref p m =
 $(\text{let } ps = to_bl\ p; ms = to_bl\ m;$
 $n = length\ (dropWhile\ Not\ ms);$
 $ps' = take\ (n + 1)\ ps$
 $\text{in set } ps' = \{\text{False}\})$

118

119



inv_ptrieW :: ptrieW \Rightarrow bool
inv_ptrieW LeafW = True

inv_ptrieW (NodeW p m b (l, r)) =
 $(inv_pref\ p\ m \wedge inv_mask\ m \wedge inv_ptrieW\ l \wedge$
 $inv_ptrieW\ r)$

inv_pref p m =
 $(\text{let } ps = to_bl\ p; ms = to_bl\ m;$
 $n = length\ (dropWhile\ Not\ ms);$
 $ps' = take\ (n + 1)\ ps$
 $\text{in set } ps' = \{\text{False}\})$

inv_mask m = ([b \leftarrow to_bl\ m . b] = [True])



abs_ptrieW :: ptrieW \Rightarrow ptrie

abs_ptrieW LeafW = LeafP

119

120



Patricia trie for words

```
datatype ptrieW = LeafW  
| NodeW (32 word) (32 word) bool  
(ptrieW × ptrieW)
```

to_bl :: 'a word ⇒ bool list



Further space optimization?

```
datatype ptrieW2 =  
LeafW2 |  
SingW2 (32 word) |  
NodeW2 (32 word) (32 word) (ptrieW2 × ptrieW2)
```