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

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.

Trie

Name: *reTRIEval*

- Tries are search trees indexed by lists.

- Each node maps the next element in the list to a subtrie.

Trie

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

Patricia trie

Sequences of edges without branching are compressed

Patricia trie

Sequences of edges without branching are compressed

Patricia trie

Sequences of edges without branching are compressed

datatype $ptrie = LeafP$
$\mid NodeP\ (bool\ list)\ bool\ (ptrie \times ptrie)$

Patricia trie

Sequences of edges without branching are compressed

datatype $ptrie = LeafP$
$\mid NodeP\ (bool\ list)\ bool\ (ptrie \times ptrie)$

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

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

Data refinement

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

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 $abs_C :: C \Rightarrow A$
- Implement each interface function f_A on A by some f_C on C

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

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

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

The fs may take more parameters

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

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

Data refinement

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

Trie

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

Trie *isin*

$isin\ Leaf\ ks = False$

$isin\ (Node\ b\ (l,\ r))\ ks =$
(case ks of
$[] \Rightarrow b$
$\mid k\ \#\ x \Rightarrow isin\ (\text{if}\ k\ \text{then}\ r\ \text{else}\ l)\ x$)

Trie *insert*

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

Trie *insert*

$$
\begin{array}{l}
insert\ []\ Leaf = Node\ True\ (Leaf,\ Leaf) \\
insert\ []\ (Node\ b\ lr) = Node\ True\ lr \\
\\
insert\ (k\ \#\ ks)\ Leaf = \\
Node\ False \\
\ (\mathsf{if}\ k\ \mathsf{then}\ (Leaf,\ insert\ ks\ Leaf) \\
\ \ \mathsf{else}\ (insert\ ks\ Leaf,\ Leaf)) \\
\\
insert\ (k\ \#\ ks)\ (Node\ b\ (l,\ r)) = \\
Node\ b\ (\mathsf{if}\ k\ \mathsf{then}\ (l,\ insert\ ks\ r)\ \mathsf{else}\ (insert\ ks\ l,\ r))
\end{array}
$$

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

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

Trie *insert*

$$
\begin{array}{l}
insert\ []\ Leaf = Node\ True\ (Leaf,\ Leaf) \\
insert\ []\ (Node\ b\ lr) = Node\ True\ lr \\
\\
insert\ (k\ \#\ ks)\ Leaf = \\
Node\ False \\
\ (\mathsf{if}\ k\ \mathsf{then}\ (Leaf,\ insert\ ks\ Leaf) \\
\ \ \mathsf{else}\ (insert\ ks\ Leaf,\ Leaf)) \\
\\
insert\ (k\ \#\ ks)\ (Node\ b\ (l,\ r)) = \\
Node\ b\ (\mathsf{if}\ k\ \mathsf{then}\ (l,\ insert\ ks\ r)\ \mathsf{else}\ (insert\ ks\ l,\ r))
\end{array}
$$

105

11 Tries and Patricia Tries
Tries and Patricia Tries
Data Refinement Basics
Trie
Patricia Trie
Patricia Trie for Words

Patricia trie

datatype $ptrie = LeafP$
$\quad | \; NodeP\ (bool\ list)\ bool\ (ptrie \times ptrie)$

Splitting lists

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

Splitting lists

$split\ xs\ ys = (zs,\ xs',\ ys')$
iff zs is the longest common prefix of xs and ys

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

$insertP$

Splitting lists

$split\ xs\ ys = (zs,\ xs',\ ys')$
iff zs is the longest common prefix of xs and ys

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

$abs\_ptrie :: ptrie \Rightarrow trie$

$abs\_ptrie\ LeafP = Leaf$

$abs\_ptrie\ (NodeP\ ps\ b\ (l,\ r)) =$
$prefix\_trie\ ps\ (Node\ b\ (abs\_ptrie\ l,\ abs\_ptrie\ r))$

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

$$isinP\ t\ ks = isin\ (abs\_ptrie\ t)\ ks$$

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

$$isinP\ t\ ks = isin\ (abs\_ptrie\ t)\ ks$$

$$abs\_ptrie\ (insertP\ ks\ t) = insert\ ks\ (abs\_ptrie\ t)$$

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 $ptrieW = LeafW$
$\mid NodeW\ (32\ word)\ (32\ word)\ bool$
$(ptrieW \times ptrieW)$

Patricia trie for words

datatype $ptrie W = LeafW$
| $NodeW\ (32\ word)\ (32\ word)\ bool$
$(ptrie W \times ptrie W)$

$to\_bl :: 'a\ word \Rightarrow bool\ list$

$isin W$

$isin W\ LeafW\ k = False$

$isin W$

$isin W\ LeafW\ k = False$

$isin W\ (NodeW\ p\ m\ b\ (l,\ r))\ k =$
(if b then $k = p$
else if $match\_prefix\ p\ m\ k$
then $isin W$ (if $go\_left\ m\ k$ then l else r) k
else $False$)

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

$inv\_ptrie W\ t \Longrightarrow$
$isin W\ t\ k = isinL\ (abs\_ptrie W\ t)\ (rev\ (to\_bl\ k))$

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

$inv\_ptrie\,W\ t \Longrightarrow$
$isin\,W\ t\ k = isinL\ (abs\_ptrie\,W\ t)\ (rev\ (to\_bl\ k))$

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

Refinement hierarchy

$$
\begin{array}{ll}
trie & \\
\quad\uparrow & abs\_ptrie \\
ptrie & \\
\quad\uparrow & abs\_ptrieL \\
ptrieL & \\
\quad\uparrow & abs\_ptrie\,W \\
ptrie\,W &
\end{array}
$$

Refinement hierarchy

$$
\begin{array}{ll}
trie & \\
\quad\uparrow & abs\_ptrie \\
ptrie & \\
\quad\uparrow & abs\_ptrieL \\
ptrieL & \\
\quad\uparrow & abs\_ptrie\,W \\
ptrie\,W &
\end{array}
$$

where $ptrieL = ptrie$

$inv\_ptrie\,W :: ptrie\,W \Rightarrow bool$

Refinement hierarchy

$$
\begin{array}{ll}
trie & \\
\quad\uparrow & abs\_ptrie \\
ptrie & \\
\quad\uparrow & abs\_ptrieL \\
ptrieL & \\
\quad\uparrow & abs\_ptrieW \\
ptrieW &
\end{array}
$$

$inv\_ptrieW :: ptrieW \Rightarrow bool$

$inv\_ptrieW\ LeafW = True$

$inv\_ptrieW\ (NodeW\ p\ m\ b\ (l,\ r)) =$
$(inv\_pref\ p\ m \wedge inv\_mask\ m \wedge inv\_ptrieW\ l \wedge$
$inv\_ptrieW\ r)$

$inv\_pref\ p\ m =$
$(\mathsf{let}\ ps = to\_bl\ p;\ ms = to\_bl\ m;$
$\quad n = length\ (dropWhile\ Not\ ms);$
$\quad ps' = take\ (n + 1)\ ps$
$\ \mathsf{in}\ set\ ps' = \{False\})$

$inv\_ptrieW :: ptrieW \Rightarrow bool$

$inv\_ptrieW\ LeafW = True$

$inv\_ptrieW\ (NodeW\ p\ m\ b\ (l,\ r)) =$
$(inv\_pref\ p\ m \wedge inv\_mask\ m \wedge inv\_ptrieW\ l \wedge$
$inv\_ptrieW\ r)$

$inv\_pref\ p\ m =$
$(\mathsf{let}\ ps = to\_bl\ p;\ ms = to\_bl\ m;$
$\quad n = length\ (dropWhile\ Not\ ms);$
$\quad ps' = take\ (n + 1)\ ps$
$\ \mathsf{in}\ set\ ps' = \{False\})$

$inv\_mask\ m = ([b \leftarrow to\_bl\ m\ .\ b] = [True])$

$abs\_ptrieW :: ptrieW \Rightarrow ptrie$

$abs\_ptrieW\ LeafW = LeafP$

Patricia trie for words

datatype $ptrieW = LeafW$
$\quad | \; NodeW\ (32\ word)\ (32\ word)\ bool$
$\qquad (ptrieW \times ptrieW)$

$to\_bl :: 'a\ word \Rightarrow bool\ list$

Further space optimization?

datatype $ptrieW2 =$
$\quad LeafW2\ |$
$\quad SingW2\ (32\ word)\ |$
$\quad NodeW2\ (32\ word)\ (32\ word)\ (ptrieW2 \times ptrieW2)$