This module verifies the correctness of the algorith used to implement *DeletePrefix* in the go-immutable-radix project. (https://*github.com*/hashicorp/go-immutable-radix)

MODULE *RadixDeletePrefix*

EXTENDS *FiniteSets*, *Integers*, *Sequences*, *TLC*
INSTANCE *RadixTrees*

Set of characters to use for the alphabet of generated strings.
CONSTANT *Alphabet*

Length of input strings generated
CONSTANT *MinLength*, *MaxLength*
ASSUME
$\land \{MinLength, MaxLength\} \subseteq Nat$
$\land MinLength \leq MaxLength$
$\land MinLength > 0$

Number of unique elements to construct the radix tree with. This
is a set of numbers so you can test with inputs of multiple sizes.
CONSTANT *ElementCounts*
ASSUME $ElementCounts \subseteq Nat$

Inputs is the set of input strings valid for the tree.
$Inputs \triangleq \text{UNION } \{[1 \,..\, n \to Alphabet] : n \in MinLength \,..\, MaxLength\}$

InputSets is the full set of possible inputs we can send to the radix tree.
$InputSets \triangleq \{T \in \text{SUBSET } Inputs : Cardinality(T) \in ElementCounts\}$

TRUE iff *seq* is prefixed with prefix.
$HasPrefix(seq, prefix) \triangleq$
$\land Len(seq) \geq Len(prefix)$
$\land \forall\, i \in 1 \,..\, Len(prefix) : seq[i] = prefix[i]$

Remove prefix from *seq*.
$TrimPrefix(seq, prefix) \triangleq [i \in 1 \,..\, (Len(seq) - Len(prefix)) \mapsto seq[i + Len(prefix)]]$

DeletePrefix should be equivalent to the tree without inputs that have that prefix.
This purposely doesn't model the "delete" algorithm at all: only the end result
of what the tree should contain.
$ExpectedTree(input, prefix) \triangleq RadixTree(\{value \in input : \neg HasPrefix(value, prefix)\})$

--algorithm *delete_prefix*
variables
$input \in InputSets$,
$prefix \in Inputs$,

```
  root = RadixTree(input),
  newChild = ⟨⟩,
  search = {},
  result  = {} ;

define
    We determine if newChild is "nil" by checking if it has the empty domain,
    since a non-null child will be a function with domains prefix, value, etc.
  NewChildNull ≜ DOMAIN newChild = {}
end define ;

 Precondition: Len(n.Edges) = 1
procedure mergeChild(n = ⟨⟩)
begin
MergeChild:
  with
    label = CHOOSE x ∈ DOMAIN n.Edges : TRUE,   we know we have only one edge
    child = n.Edges[label]
   do
    n.Prefix := n.Prefix ∘ child.Prefix ||
    n.Value := child.Value ||
    n.Edges := child.Edges ;
  end with ;

ExitMergeChild:
  return ;
end procedure ;

procedure deletePrefix(n = ⟨⟩, nRoot = FALSE)
variables searchLabel = ⟨⟩ ;
begin
DeletePrefix:
    Check for key exhaustion
  if Len(search) = 0 then
    newChild := [
      Prefix ↦ n.Prefix,
      Value  ↦ ⟨⟩,
      Edges ↦ ⟨⟩
    ] ;
    return ;
  end if ;

FindEdge:
    Look for an edge
  searchLabel := search[1] ;
  if ¬searchLabel ∈ DOMAIN n.Edges then
  NoEdge:
```

```
    newChild := ⟨⟩ ;
    return ;
  end if ;

ConsumeAndRecurse:
  with child = n.Edges[searchLabel] do
    if ¬HasPrefix(child.Prefix, search) ∧ ¬HasPrefix(search, child.Prefix) then
      newChild := ⟨⟩ ;
      return ;
     else
       Consume the search prefix
      if Len(child.Prefix) > Len(search) then
        search := ⟨⟩ ;
       else
        search := SubSeq(search, Len(child.Prefix) + 1, Len(search))
      end if ;

      call deletePrefix(child, FALSE) ;
    end if   ;
  end with ;

ExitIfNoChild:
  if NewChildNull then
    newChild := ⟨⟩ ;
    return ;
  end if ;

ModifyNode:
  if Len(newChild.Value) = 0 ∧ Cardinality(DOMAIN newChild.Edges) = 0 then
    n.Edges := [label ∈ DOMAIN n.Edges \ {searchLabel} ↦ n.Edges[label]] ;

    if ¬nRoot ∧ Cardinality(DOMAIN n.Edges) = 1 ∧ Len(n.Value) = 0 then
      call mergeChild(n) ;
    end if ;
   else
    n.Edges[searchLabel] := newChild
  end if ;

ReturnDeletePrefix:
  newChild := n ;
  return ;
end procedure ;
```

This entire algorith is almost 1:1 translated where possible from the actual implementation in *iter.go*. That's the point: we're trying to verify our algorithm is correct for all inputs.

```
Source: https://github.com/hashicorp/go-immutable-radix/blob/f63f49c0b598a5ead21c5015fb4d08fe7e3c21ea/iter.go ≠ L16
begin
Begin:
  search := prefix ;
  call deletePrefix(root, TRUE) ;

SetNewRoot:
  if ¬NewChildNull then
    root := newChild ;
  end if ;

AssertExpected:
    check our expected values
  with
    actual = Range(root),
    expected = Range(ExpectedTree(input, prefix))
   do
    if actual ≠ expected then
      print ⟨"value check", "actual", actual, "expected", expected⟩ ;
      assert FALSE ;
    end if   ;
  end with ;

    check our expected tree structure for an optimal structure
 with actual = root,
  expected = ExpectedTree(input, prefix)
 do if actual ≠ expected then
    print ⟨"tree check", "actual", actual, "expected", expected⟩; assert FALSE;
  end if;
 end with;
end algorithm   ;
```

!!! *NOTE* !!! The rest of the file is auto-generated based on the *PlusCal* above. For those who are reading this to learn TLA+/*PlusCal*, you can stop reading here.

BEGIN TRANSLATION ($chksum(pcal) =$ "$c97935ab$" $\land$ $chksum(tla) =$ "$b2f124d$")
Parameter n of procedure *mergeChild* at line 63 col 22 changed to $n\_$

VARIABLES *input*, *prefix*, *root*, *newChild*, *search*, *result*, *pc*, *stack*

define statement

$NewChildNull \triangleq$ DOMAIN $newChild = \{\}$

VARIABLES $n\_$, n, *nRoot*, *searchLabel*

$vars \triangleq \langle input, prefix, root, newChild, search, result, pc, stack, n\_, n,$

$nRoot,\ searchLabel\rangle$

$Init \triangleq$ Global variables
$\land\ input \in InputSets$
$\land\ prefix \in Inputs$
$\land\ root = RadixTree(input)$
$\land\ newChild = \langle\rangle$
$\land\ search = \{\}$
$\land\ result\ = \{\}$
Procedure *mergeChild*
$\land\ n\_ = \langle\rangle$
Procedure *deletePrefix*
$\land\ n = \langle\rangle$
$\land\ nRoot = \text{FALSE}$
$\land\ searchLabel = \langle\rangle$
$\land\ stack = \langle\rangle$
$\land\ pc = \text{“Begin”}$

$MergeChild \triangleq \land\ pc = \text{“MergeChild”}$
$\land\ \text{LET}\ label \triangleq \text{CHOOSE}\ x \in \text{DOMAIN}\ n\_.Edges : \text{TRUEIN}$
$\text{LET}\ child \triangleq n\_.Edges[label]\text{IN}$
$n\_' = [n\_\ \text{EXCEPT}\ !.Prefix = n\_.Prefix \circ child.Prefix,$
$!.Value = child.Value,$
$!.Edges = child.Edges]$
$\land\ pc' = \text{“ExitMergeChild”}$
$\land\ \text{UNCHANGED}\ \langle input,\ prefix,\ root,\ newChild,\ search,\ result,$
$stack,\ n,\ nRoot,\ searchLabel\rangle$

$ExitMergeChild \triangleq \land\ pc = \text{“ExitMergeChild”}$
$\land\ pc' = Head(stack).pc$
$\land\ n\_' = Head(stack).n\_$
$\land\ stack' = Tail(stack)$
$\land\ \text{UNCHANGED}\ \langle input,\ prefix,\ root,\ newChild,\ search,$
$result,\ n,\ nRoot,\ searchLabel\rangle$

$mergeChild \triangleq MergeChild \lor ExitMergeChild$

$DeletePrefix \triangleq \land\ pc = \text{“DeletePrefix”}$
$\land\ \text{IF}\ Len(search) = 0$
$\text{THEN}\ \land\ newChild' = [$
$Prefix \mapsto n.Prefix,$
$Value\ \mapsto \langle\rangle,$
$Edges \mapsto \langle\rangle$
$]$
$\land\ pc' = Head(stack).pc$
$\land\ searchLabel' = Head(stack).searchLabel$

$\land\ n' = Head(stack).n$
$\land\ nRoot' = Head(stack).nRoot$
$\land\ stack' = Tail(stack)$
ELSE $\land\ pc' =$ "FindEdge"
$\land$ UNCHANGED $\langle newChild,\ stack,\ n,\ nRoot,\ searchLabel\rangle$
$\land$ UNCHANGED $\langle input,\ prefix,\ root,\ search,\ result,\ n\_\rangle$

$FindEdge \triangleq$ $\land\ pc =$ "FindEdge"
$\land\ searchLabel' = search[1]$
$\land$ IF $\neg searchLabel' \in$ DOMAIN $n.Edges$
THEN $\land\ pc' =$ "NoEdge"
ELSE $\land\ pc' =$ "ConsumeAndRecurse"
$\land$ UNCHANGED $\langle input,\ prefix,\ root,\ newChild,\ search,\ result,\ stack,\ n\_,\ n,\ nRoot\rangle$

$NoEdge \triangleq$ $\land\ pc =$ "NoEdge"
$\land\ newChild' = \langle\rangle$
$\land\ pc' = Head(stack).pc$
$\land\ searchLabel' = Head(stack).searchLabel$
$\land\ n' = Head(stack).n$
$\land\ nRoot' = Head(stack).nRoot$
$\land\ stack' = Tail(stack)$
$\land$ UNCHANGED $\langle input,\ prefix,\ root,\ search,\ result,\ n\_\rangle$

$ConsumeAndRecurse \triangleq$ $\land\ pc =$ "ConsumeAndRecurse"
$\land$ LET $child \triangleq n.Edges[searchLabel]$ IN
IF $\neg HasPrefix(child.Prefix,\ search) \land \neg HasPrefix(search,\ child.Prefix)$
THEN $\land\ newChild' = \langle\rangle$
$\land\ pc' = Head(stack).pc$
$\land\ searchLabel' = Head(stack).searchLabel$
$\land\ n' = Head(stack).n$
$\land\ nRoot' = Head(stack).nRoot$
$\land\ stack' = Tail(stack)$
$\land$ UNCHANGED $search$
ELSE $\land$ IF $Len(child.Prefix) > Len(search)$
THEN $\land\ search' = \langle\rangle$
ELSE $\land\ search' = SubSeq(search,\ Len(child.Prefix) + 1,\ Len(se$
$\land\ \land\ n' = child$
$\land\ nRoot' =$ FALSE
$\land\ stack' = \langle[procedure \mapsto$ "deletePrefix",
$pc \mapsto$ "ExitIfNoChild",
$searchLabel \mapsto\ searchLabel,$
$n \mapsto\ n,$
$nRoot \mapsto\ nRoot]\rangle$
$\circ\ stack$

$\wedge\ searchLabel' = \langle\rangle$
$\wedge\ pc' =$ "DeletePrefix"
$\wedge$ UNCHANGED $newChild$
$\wedge$ UNCHANGED $\langle input,\ prefix,\ root,\ result,\ n\_\rangle$

$ExitIfNoChild \triangleq$ $\wedge\ pc =$ "ExitIfNoChild"
$\wedge$ IF $NewChildNull$
THEN $\wedge\ newChild' = \langle\rangle$
$\wedge\ pc' = Head(stack).pc$
$\wedge\ searchLabel' = Head(stack).searchLabel$
$\wedge\ n' = Head(stack).n$
$\wedge\ nRoot' = Head(stack).nRoot$
$\wedge\ stack' = Tail(stack)$
ELSE $\wedge\ pc' =$ "ModifyNode"
$\wedge$ UNCHANGED $\langle newChild,\ stack,\ n,\ nRoot,\ searchLabel\rangle$
$\wedge$ UNCHANGED $\langle input,\ prefix,\ root,\ search,\ result,\ n\_\rangle$

$ModifyNode \triangleq$ $\wedge\ pc =$ "ModifyNode"
$\wedge$ IF $Len(newChild.Value) = 0 \wedge Cardinality(\text{DOMAIN}\ newChild.Edges) = 0$
THEN $\wedge\ n' = [n$ EXCEPT $!.Edges = [label \in \text{DOMAIN}\ n.Edges \setminus \{searchLabel\} \mapsto n.Edge$
$\wedge$ IF $\neg nRoot \wedge Cardinality(\text{DOMAIN}\ n'.Edges) = 1 \wedge Len(n'.Value) = 0$
THEN $\wedge\ \wedge\ n\_' = n'$
$\wedge\ stack' = \langle[procedure \mapsto$ "mergeChild",
$pc \mapsto$ "ReturnDeletePrefix",
$n\_ \mapsto n\_]\rangle$
$\circ\ stack$
$\wedge\ pc' =$ "MergeChild"
ELSE $\wedge\ pc' =$ "ReturnDeletePrefix"
$\wedge$ UNCHANGED $\langle stack,\ n\_\rangle$
ELSE $\wedge\ n' = [n$ EXCEPT $!.Edges[searchLabel] = newChild]$
$\wedge\ pc' =$ "ReturnDeletePrefix"
$\wedge$ UNCHANGED $\langle stack,\ n\_\rangle$
$\wedge$ UNCHANGED $\langle input,\ prefix,\ root,\ newChild,\ search,\ result,\ nRoot,\ searchLabel\rangle$

$ReturnDeletePrefix \triangleq$ $\wedge\ pc =$ "ReturnDeletePrefix"
$\wedge\ newChild' = n$
$\wedge\ pc' = Head(stack).pc$
$\wedge\ searchLabel' = Head(stack).searchLabel$
$\wedge\ n' = Head(stack).n$
$\wedge\ nRoot' = Head(stack).nRoot$
$\wedge\ stack' = Tail(stack)$
$\wedge$ UNCHANGED $\langle input,\ prefix,\ root,\ search,\ result,\ n\_\rangle$

$deletePrefix \triangleq DeletePrefix \vee FindEdge \vee NoEdge \vee ConsumeAndRecurse$

$\lor\ ExitIfNoChild \lor ModifyNode \lor ReturnDeletePrefix$

$$
\begin{array}{l}
Begin \triangleq \land\ pc = \text{“Begin”} \\
\qquad \land\ search' = prefix \\
\qquad \land\ \land\ n' = root \\
\qquad\quad \land\ nRoot' = \text{TRUE} \\
\qquad\quad \land\ stack' = \langle [procedure \mapsto \text{“deletePrefix”}, \\
\qquad\qquad\qquad\qquad pc \mapsto \text{“SetNewRoot”}, \\
\qquad\qquad\qquad\qquad searchLabel \mapsto searchLabel, \\
\qquad\qquad\qquad\qquad n \mapsto n, \\
\qquad\qquad\qquad\qquad nRoot \mapsto nRoot] \rangle \\
\qquad\qquad\qquad\qquad \circ\ stack \\
\qquad \land\ searchLabel' = \langle\rangle \\
\qquad \land\ pc' = \text{“DeletePrefix”} \\
\qquad \land\ \text{UNCHANGED}\ \langle input,\ prefix,\ root,\ newChild,\ result,\ n\_ \rangle
\end{array}
$$

$$
\begin{array}{l}
SetNewRoot \triangleq \land\ pc = \text{“SetNewRoot”} \\
\qquad\qquad \land\ \text{IF}\ \neg NewChildNull \\
\qquad\qquad\qquad\quad \text{THEN}\ \land\ root' = newChild \\
\qquad\qquad\qquad\quad \text{ELSE}\ \land\ \text{TRUE} \\
\qquad\qquad\qquad\qquad\qquad\ \land\ root' = root \\
\qquad\qquad \land\ pc' = \text{“AssertExpected”} \\
\qquad\qquad \land\ \text{UNCHANGED}\ \langle input,\ prefix,\ newChild,\ search,\ result,\ stack, \\
\qquad\qquad\qquad\qquad\qquad\qquad n\_,\ n,\ nRoot,\ searchLabel \rangle
\end{array}
$$

$$
\begin{array}{l}
AssertExpected \triangleq \land\ pc = \text{“AssertExpected”} \\
\qquad\qquad \land\ \text{LET}\ actual \triangleq Range(root)\ \text{IN} \\
\qquad\qquad\qquad \text{LET}\ expected \triangleq Range(ExpectedTree(input,\ prefix))\ \text{IN} \\
\qquad\qquad\qquad\quad \text{IF}\ actual \neq expected \\
\qquad\qquad\qquad\qquad\quad \text{THEN}\ \land\ PrintT(\langle \text{“value check”},\ \text{“actual”},\ actual,\ \text{“expected”},\ expected \rangle) \\
\qquad\qquad\qquad\qquad\qquad\qquad\ \land\ Assert(\text{FALSE}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{“Failure of assertion at line 164, column 7.”}) \\
\qquad\qquad\qquad\qquad\quad \text{ELSE}\ \land\ \text{TRUE} \\
\qquad\qquad \land\ pc' = \text{“Done”} \\
\qquad\qquad \land\ \text{UNCHANGED}\ \langle input,\ prefix,\ root,\ newChild,\ search, \\
\qquad\qquad\qquad\qquad\qquad\qquad result,\ stack,\ n\_,\ n,\ nRoot,\ searchLabel \rangle
\end{array}
$$

Allow infinite stuttering to prevent deadlock on termination.

$$Terminating \triangleq pc = \text{“Done”} \land \text{UNCHANGED}\ vars$$

$$
\begin{array}{l}
Next \triangleq mergeChild \lor deletePrefix \lor Begin \lor SetNewRoot \lor AssertExpected \\
\qquad\qquad \lor\ Terminating
\end{array}
$$

$$Spec \triangleq Init \land \Box[Next]_{vars}$$

$$Termination \triangleq \Diamond(pc = \text{“Done”})$$

END TRANSLATION

\ * Modification History
\ * Last modified *Fri Jul* 02 11:33:15 *PDT* 2021 by *mitchellh*
\ * Created *Wed Jun* 30 10:05:52 *PDT* 2021 by *mitchellh*