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 $\mathit{FiniteSets}, \mathit{Integers}, \mathit{Sequences}, \mathit{TLC}$

INSTANCE $\mathit{RadixTrees}$

Set of characters to use for the alphabet of generated strings.

CONSTANT $\mathit{Alphabet}$

Length of input strings generated

CONSTANT $\mathit{MinLength}, \mathit{MaxLength}$

ASSUME

$\quad \land \{\mathit{MinLength}, \mathit{MaxLength}\} \subseteq \mathit{Nat}$

$\quad \land \mathit{MinLength} \leq \mathit{MaxLength}$

$\quad \land \mathit{MinLength} > 0$

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 $\mathit{ElementCounts}$

ASSUME $\mathit{ElementCounts} \subseteq \mathit{Nat}$

Inputs is the set of input strings valid for the tree.

$\mathit{Inputs} \triangleq \text{UNION } \{[1 \,..\, n \rightarrow \mathit{Alphabet}] : n \in \mathit{MinLength} \,..\, \mathit{MaxLength}\}$

InputSets is the full set of possible inputs we can send to the radix tree.

$\mathit{InputSets} \triangleq \{T \in \text{SUBSET } \mathit{Inputs} : \mathit{Cardinality}(T) \in \mathit{ElementCounts}\}$

TRUE iff *seq* is prefixed with prefix.

$\mathit{HasPrefix}(\mathit{seq}, \mathit{prefix}) \triangleq$

$\quad \land \mathit{Len}(\mathit{seq}) \geq \mathit{Len}(\mathit{prefix})$

$\quad \land \forall i \in 1 \,..\, \mathit{Len}(\mathit{prefix}) : \mathit{seq}[i] = \mathit{prefix}[i]$

Remove prefix from *seq*.

$\mathit{TrimPrefix}(\mathit{seq}, \mathit{prefix}) \triangleq [i \in 1 \,..\, (\mathit{Len}(\mathit{seq}) - \mathit{Len}(\mathit{prefix})) \mapsto \mathit{seq}[i + \mathit{Len}(\mathit{prefix})]]$

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.

$\mathit{ExpectedTree}(\mathit{input}, \mathit{prefix}) \triangleq \mathit{RadixTree}(\{\mathit{value} \in \mathit{input} : \neg \mathit{HasPrefix}(\mathit{value}, \mathit{prefix})\})$

```
  --algorithm delete_prefix
variables
  input ∈ InputSets,
  prefix ∈ Inputs,
  $root = RadixTree(input)$,
  $newChild = \langle\rangle$,
  $search = \{\}$,
  $result\ = \{\}$ ;

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.
  $NewChildNull \triangleq \text{DOMAIN } newChild = \{\}$
end define ;

  Precondition: $Len(n.Edges) = 1$
procedure $mergeChild(n = \langle\rangle)$
begin
$MergeChild$:
  with
    $label\ = \text{CHOOSE } x \in \text{DOMAIN } n.Edges : \text{TRUE}$,    we know we have only one edge
    $child = n.Edges[label]$
   do
    $n.Prefix := n.Prefix \circ child.Prefix$ ||
    $n.Value := child.Value$ ||
    $n.Edges := child.Edges$ ;
  end with ;

$ExitMergeChild$:
  return ;
end procedure ;

procedure $deletePrefix(n = \langle\rangle, nRoot = \text{FALSE})$
variables $searchLabel = \langle\rangle$ ;
begin
$DeletePrefix$:
    Check for key exhaustion
  if $Len(search) = 0$ then
    $newChild := [$
      $Prefix \mapsto n.Prefix$,
      $Value\ \ \mapsto \langle\rangle$,
      $Edges \mapsto \langle\rangle$
    $]$ ;
    return ;
  end if ;

$FindEdge$:
    Look for an edge
  $searchLabel := search[1]$ ;
  if $\neg searchLabel \in \text{DOMAIN } n.Edges$ then
  $NoEdge$:
    $newChild := \langle\rangle$ ;
    return ;
  end if ;

$ConsumeAndRecurse$:
  with $child = n.Edges[searchLabel]$ do
    if $\neg HasPrefix(child.Prefix, search) \land \neg HasPrefix(search, child.Prefix)$ then
      $newChild := \langle\rangle$ ;
      return ;
     else
        Consume the search prefix
      if $Len(child.Prefix) > Len(search)$ then
        $search := \langle\rangle$ ;
       else
        $search := SubSeq(search, Len(child.Prefix) + 1, Len(search))$
      end if ;

      call $deletePrefix(child, \textsc{false})$ ;
    end if ;
  end with ;

$ExitIfNoChild$:
  if $NewChildNull$ then
    $newChild := \langle\rangle$ ;
    return ;
  end if ;

$ModifyNode$:
  if $Len(newChild.Value) = 0 \land Cardinality(\textsc{domain}\ newChild.Edges) = 0$ then
    $n.Edges := [label \in \textsc{domain}\ n.Edges \setminus \{searchLabel\} \mapsto n.Edges[label]]$ ;

    if $\neg nRoot \land Cardinality(\textsc{domain}\ n.Edges) = 1 \land Len(n.Value) = 0$ 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.
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" ∧ chksum(tla) = "8366b16")
 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 ≜ DOMAIN newChild = {}

VARIABLES n_, n, nRoot, searchLabel

vars ≜ ⟨input, prefix, root, newChild, search, result, pc, stack, n_, n,
          nRoot, searchLabel⟩
```

```
Init ≜ Global variables
        ∧ input ∈ InputSets
        ∧ prefix ∈ Inputs
        ∧ root = RadixTree(input)
        ∧ newChild = ⟨⟩
        ∧ search = {}
        ∧ result  = {}
        Procedure mergeChild
        ∧ n_ = ⟨⟩
        Procedure deletePrefix
        ∧ n = ⟨⟩
        ∧ nRoot = FALSE
        ∧ searchLabel = ⟨⟩
        ∧ stack = ⟨⟩
        ∧ pc = "Begin"

MergeChild ≜ ∧ pc = "MergeChild"
             ∧ LET label ≜ CHOOSE x ∈ DOMAIN n_.Edges : TRUEIN
                 LET child ≜ n_.Edges[label]IN
                   n_′ = [n_ EXCEPT !.Prefix = n_.Prefix ∘ child.Prefix,
                                    !.Value = child.Value,
                                    !.Edges = child.Edges]
             ∧ pc′ = "ExitMergeChild"
             ∧ UNCHANGED ⟨input, prefix, root, newChild, search, result,
                            stack, n, nRoot, searchLabel⟩

ExitMergeChild ≜ ∧ pc = "ExitMergeChild"
                 ∧ pc′ = Head(stack).pc
                 ∧ n_′ = Head(stack).n_
                 ∧ stack′ = Tail(stack)
                 ∧ UNCHANGED ⟨input, prefix, root, newChild, search,
                                result, n, nRoot, searchLabel⟩

mergeChild ≜ MergeChild ∨ ExitMergeChild

DeletePrefix ≜ ∧ pc = "DeletePrefix"
               ∧ IF Len(search) = 0
                    THEN ∧ newChild′ =           [
                                          Prefix ↦ n.Prefix,
                                          Value  ↦ ⟨⟩,
                                          Edges ↦ ⟨⟩
                                        ]
                         ∧ pc′ = Head(stack).pc
                         ∧ searchLabel′ = Head(stack).searchLabel
                         ∧ n′ = Head(stack).n
                         ∧ 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",
                                                       $searchLabel \mapsto searchLabel,$
                                                       $n \mapsto n,$
                                                       $nRoot \mapsto nRoot]\rangle$
                                                       $\circ\ stack$
                                     $\land$ $searchLabel' = \langle\rangle$
                                     $\land$ $pc' =$ "DeletePrefix"
```

```
                        $\land$ UNCHANGED $newChild$
                    $\land$ UNCHANGED $\langle input, prefix, root, result, n\_ \rangle$

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

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

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

$deletePrefix \triangleq$ $DeletePrefix \lor FindEdge \lor NoEdge \lor ConsumeAndRecurse$
                   $\lor ExitIfNoChild \lor ModifyNode \lor ReturnDeletePrefix$
```

```
$Begin \triangleq \land pc = \text{"Begin"}$
          $\land search' = prefix$
          $\land \land n' = root$
             $\land nRoot' = \text{TRUE}$
             $\land stack' = \langle [procedure \mapsto \text{"deletePrefix"},$
                              $pc \mapsto \text{"SetNewRoot"},$
                              $searchLabel \mapsto searchLabel,$
                              $n \mapsto n,$
                              $nRoot \mapsto nRoot] \rangle$
                              $\circ\ stack$
          $\land searchLabel' = \langle\rangle$
          $\land pc' = \text{"DeletePrefix"}$
          $\land \text{UNCHANGED } \langle input, prefix, root, newChild, result, n\_ \rangle$

$SetNewRoot \triangleq \land pc = \text{"SetNewRoot"}$
               $\land \text{IF } \neg NewChildNull$
                     $\text{THEN } \land root' = newChild$
                     $\text{ELSE } \land \text{TRUE}$
                           $\land root' = root$
               $\land pc' = \text{"AssertExpected"}$
               $\land \text{UNCHANGED } \langle input, prefix, newChild, search, result, stack,$
                               $n\_, n, nRoot, searchLabel \rangle$

$AssertExpected \triangleq \land pc = \text{"AssertExpected"}$
                   $\land \text{LET } actual \triangleq Range(root) \text{IN}$
                        $\text{LET } expected \triangleq Range(ExpectedTree(input, prefix)) \text{IN}$
                          $\text{IF } actual \neq expected$
                             $\text{THEN } \land PrintT(\langle \text{"value check"}, \text{"actual"}, actual, \text{"expected"}, expected \rangle)$
                                  $\land Assert(\text{FALSE},$
                                               $\text{"Failure of assertion at line 162, column 7."})$
                             $\text{ELSE } \land \text{TRUE}$
                   $\land pc' = \text{"Done"}$
                   $\land \text{UNCHANGED } \langle input, prefix, root, newChild, search,$
                                   $result, stack, n\_, n, nRoot, searchLabel \rangle$

Allow infinite stuttering to prevent deadlock on termination.
$Terminating \triangleq pc = \text{"Done"} \land \text{UNCHANGED } vars$

$Next \triangleq mergeChild \lor deletePrefix \lor Begin \lor SetNewRoot \lor AssertExpected$
           $\lor Terminating$

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

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

END TRANSLATION
```

* Modification History
* Last modified *Fri Jul* 02 11:44:17 *PDT* 2021 by *mitchellh*
* Created *Wed Jun* 30 10:05:52 *PDT* 2021 by *mitchellh*