Executable Transitive Closures of Finite Relations*

Christian Sternagel and René Thiemann

September 19, 2015

Abstract

We provide a generic work-list algorithm to compute the transitive closure of finite relations where only successors of newly detected states are generated. This algorithm is then instantiated for lists over arbitrary carriers and red black trees [1] (which are faster but require a linear order on the carrier), respectively.

Our formalization was performed as part of the IsaFoR/CetA project\(^1\) [2], where reflexive transitive closures of large tree automata have to be computed.

Contents

1 A Generic Work-List Algorithm
   1.1 Bounded Reachability .................................. 2
   1.2 Reflexive Transitive Closure and Transitive closure ....... 3

2 Closure Computation using Lists .......................... 4
   2.1 Computing Closures from Sets On-The-Fly ............... 5
   2.2 Precomputing Closures for Single States .................. 5

3 Utility Functions and Lemmas ................................ 6
   3.1 Miscellaneous .......................................... 6

4 Accessing Values via Keys .................................... 8
   4.1 Subset and Union ........................................ 8
   4.2 Grouping Values via Keys ............................... 9

5 Closure Computation via Red Black Trees .................. 10
   5.1 Computing Closures from Sets On-The-Fly ............... 11
   5.2 Precomputing Closures for Single States ................ 11

---

*Supported by FWF (Austrian Science Fund) project P22767-N13.
\(^1\)http://cl-informatik.uibk.ac.at/software/ceta
6 Computing Images of Finite Transitive Closures

6.1 A Simproc for Computing the Images of Finite Transitive Closures

6.2 Example

1 A Generic Work-List Algorithm

theory Transitive-Closure-Impl
imports Main
begin

Let \( R \) be some finite relation. We start to present a standard work-list algorithm to compute all elements that are reachable from some initial set by at most \( n \) \( R \)-steps. Then, we obtain algorithms for the (reflexive) transitive closure from a given starting set by exploiting the fact that for finite relations we have to iterate at most \( \text{card} \ R \) times. The presented algorithms are generic in the sense that the underlying data structure can freely be chosen, you just have to provide certain operations like union, membership, etc.

1.1 Bounded Reachability

We provide an algorithm \( \text{relpow-impl} \) that computes all states that are reachable from an initial set of states \( \text{new} \) by at most \( n \) steps. The algorithm also stores a set of states that have already been visited \( \text{have} \), and then show, do not have to be expanded a second time. The algorithm is parametric in the underlying data structure, it just requires operations for union and membership as well as a function to compute the successors of a list.

fun relpow-impl ::
(\'a list \Rightarrow \'a list) \Rightarrow
(\'a list \Rightarrow \'b \Rightarrow \'b) \Rightarrow (\'a \Rightarrow \'b \Rightarrow \text{bool}) \Rightarrow \'a list \Rightarrow \'b \Rightarrow \text{nat} \Rightarrow \'b

where
relpow-impl succ un memb new have 0  =  un new have |
relpow-impl succ un memb new have (Suc m)  =
(if new = [] then have
else
let
  maybe = succ new;
  have' = un new have;
  new' = filter (\lambda n. \neg memb n have') maybe
  in relpow-impl succ un memb new' have' m)

We need to know that the provided operations behave correctly.

locale set-access =
fixes un :: \'a list \Rightarrow \'b \Rightarrow \'b
and \texttt{set-of} :: 'b ⇒ 'a set  
and \texttt{memb} :: 'a ⇒ 'b ⇒ bool  
and \texttt{empty} :: 'b

\textbf{assumes} \texttt{un}: \texttt{set-of} (\texttt{un} as \texttt{bs}) = \texttt{set as} ∪ \texttt{set-of} \texttt{bs}  
and \texttt{memb}: \texttt{memb} \texttt{a bs} ⇐⇒ (\texttt{a} ∈ \texttt{set-of} \texttt{bs})  
and \texttt{empty}: \texttt{set-of} \texttt{empty} = \{\}

locale \texttt{set-access-succ} = \texttt{set-access} \texttt{un} 
for \texttt{un} :: 'a list ⇒ 'b ⇒ 'b 
fixes \texttt{succ} :: 'a list ⇒ 'a list  
and \texttt{rel} :: ('a × 'a) set  
\textbf{assumes} \texttt{succ}: \texttt{set} \texttt{(\texttt{succ} as)} = \{ b. \ ∃ a ∈ \texttt{set as}. (a, b) ∈ \texttt{rel}\}

begin

abbreviation \texttt{relpow-i} ≡ \texttt{relpow-impl} \texttt{succ} \texttt{un} \texttt{memb}

What follows is the main technical result of the \texttt{relpow-impl} algorithm: what it computes for arbitrary values of \texttt{new} and \texttt{have}.

\textbf{lemma} \texttt{relpow-impl-main}:
\texttt{set-of} (\texttt{relpow-i new have} \texttt{n}) =
\{ b | a b m. a ∈ \texttt{set new} \land m ≤ n \land (a, b) ∈ (\texttt{rel} ∩ \{ (a, b). b /∈ \texttt{set-of} \texttt{have} \}) \} ^\ast 
\cup
\texttt{set-of} \texttt{have}
\texttt{(is ?l new have n = ?r new have n)} 
⟨\texttt{proof}⟩

From the previous lemma we can directly derive that \texttt{relpow-impl} works correctly if \texttt{have} is initially set to \texttt{empty}

\textbf{lemma} \texttt{relpow-impl}:
\texttt{set-of} (\texttt{relpow-i new empty} \texttt{n}) = \{ b | a b m. a ∈ \texttt{set new} \land m ≤ n \land (a, b) ∈ \texttt{rel} \} ^\ast
⟨\texttt{proof}⟩

end

1.2 Reflexive Transitive Closure and Transitive closure

Using \texttt{relpow-impl} it is now easy to obtain algorithms for the reflexive transitive closure and the transitive closure by restricting the number of steps to the size of the finite relation. Note that \texttt{relpow-impl} will abort the computation as soon as no new states are detected. Hence, there is no penalty in using this large bound.

\textbf{definition}
\texttt{rtrancl-impl} ::
\texttt{(('a × 'a) list ⇒ 'a list ⇒ 'a list) ⇒}
\texttt{('a list ⇒ 'b ⇒ 'b) ⇒ ('a ⇒ 'b ⇒ bool) ⇒ 'b ⇒ ('a × 'a) list ⇒ 'a list ⇒ 'b}

\textbf{where}
\texttt{rtrancl-impl gen-succ} \texttt{un} \texttt{memb} \texttt{emp} \texttt{rel} =
\begin{verbatim}
(let
  succ = gen-succ rel;
  n = length rel
in (\lambda as. relpow-impl succ \text{\textit{un}} \text{\textit{memb}} \text{\textit{as}}} \text{\textit{emp}} n))
\end{verbatim}

\textbf{definition}
\begin{verbatim}
trancl-impl ::
  (('a × 'a) list ⇒ 'a list ⇒ 'a list) ⇒
  ('a list ⇒ 'b ⇒ 'b) ⇒ ('a ⇒ 'b ⇒ bool) ⇒ 'b ⇒ ('a × 'a) list ⇒ 'a list ⇒ 'b
\end{verbatim}

\textbf{where}
\begin{verbatim}
trancl-impl \text{\textit{gen-succ}} \text{\textit{un}} \text{\textit{memb}} \text{\textit{emp}} \text{\textit{rel}} =
  (let
    succ = gen-succ rel;
    n = length rel
in (\lambda \text{\textit{as}}. relpow-impl succ \text{\textit{un}} \text{\textit{memb}}
    (\text{\textit{succ}} \text{\textit{as}}) \text{\textit{emp}} \text{\textit{n}}))
\end{verbatim}

The soundness of both \textit{rtrancl-impl} and \textit{trancl-impl} follows from the soundness of \textit{relpow-impl} and the fact that for finite relations, we can limit the number of steps to explore all elements in the reflexive transitive closure.

\textbf{lemma} \textit{rtrancl-finite-relpow}:
\begin{verbatim}
  \((a, b) \in (set rel)^* \iff (\exists n \leq length rel. (a, b) \in set \text{\textit{rel}}^\ast \text{\textit{n}})) \text{\textit{is}} \text{\textit{l}} = \text{\textit{r}}\)
\end{verbatim}

\textbf{locale} \textit{set-access-gen} = \textit{set-access \textit{un}}
\begin{verbatim}
for \text{\textit{un}} :: 'a list \Rightarrow 'b \Rightarrow 'b +
fixes \text{\textit{gen-succ}} :: ('a × 'a) list \Rightarrow 'a list ⇒ 'a list
assumes \text{\textit{gen-succ}}: set (\text{\textit{gen-succ}} \text{\textit{rel}} \text{\textit{as}}) = \{b. \exists \text{\textit{a}} \in \text{\textit{set as}}. (a, b) \in \text{\textit{set rel}}\}
begin
abbreviation \text{\textit{rtrancl-i}} ≡ \text{\textit{rtrancl-impl}} \text{\textit{gen-succ}} \text{\textit{un}} \text{\textit{memb}} \text{\textit{empty}}
abbreviation \text{\textit{trancl-i}} ≡ \text{\textit{trancl-impl}} \text{\textit{gen-succ}} \text{\textit{un}} \text{\textit{memb}} \text{\textit{empty}}

\textbf{lemma} \textit{rtrancl-impl}:
\begin{verbatim}
  set-of (\text{\textit{rtrancl-i}} \text{\textit{rel}} \text{\textit{as}}) = \{b. (\exists \text{\textit{a}} \in \text{\textit{set as}}. (a, b) \in (set rel)^*)\}
\end{verbatim}

\textbf{lemma} \textit{trancl-impl}:
\begin{verbatim}
  set-of (\text{\textit{trancl-i}} \text{\textit{rel}} \text{\textit{as}}) = \{b. (\exists \text{\textit{a}} \in \text{\textit{set as}}. (a, b) \in (set rel)^*)\}
\end{verbatim}

\end{verbatim}

\section{Closure Computation using Lists}

\textbf{theory} \textit{Transitive-Closure-List-Impl}
\textbf{imports} \textit{Transitive-Closure-Impl}
begin

\end}
We provide two algorithms for the computation of the reflexive transitive closure which internally work on lists. The first one \( rtrancl-list-impl \) computes the closure on demand for a given set of initial states. The second one \( \text{memo-list-rtrancl} \) precomputes the closure for each individual state, stores the result, and then only does a look-up.

For the transitive closure there are the corresponding algorithms \( trancl-list-impl \) and \( \text{memo-list-trancl} \).

### 2.1 Computing Closures from Sets On-The-Fly

The algorithms are based on the generic algorithms \( rtrancl-impl \) and \( trancl-impl \) instantiated by list operations. Here, after computing the successors in a straightforward way, we use \( \text{remdups} \) to not have duplicates in the results. Moreover, also in the union operation we filter to those elements that have not yet been seen. The use of \( \text{filter} \) in the union operation is preferred over \( \text{remdups} \) since by construction the latter set will not contain duplicates.

\[ \text{definition } rtrancl-list-impl :: ('a \times 'a) \text{ list } \Rightarrow 'a \text{ list } \Rightarrow 'a \text{ list } \]
\[ \text{where} rtrancl-list-impl = rtrancl-impl \]
\[ (\lambda r \ as. \ \text{remdups} \ (\text{map} \ \text{snd} \ (\lambda (a, b). \ a \in \text{set as}) \ r))) \]
\[ (\lambda xs \ ys. \ (\text{filter} \ (\lambda x. \ x \notin \text{set ys}) \ xs) @ ys) \]
\[ (\lambda x \ xs. \ x \in \text{set xs}) \]

\[ \text{definition } trancl-list-impl :: ('a \times 'a) \text{ list } \Rightarrow 'a \text{ list } \Rightarrow 'a \text{ list } \]
\[ \text{where} trancl-list-impl = trancl-impl \]
\[ (\lambda r \ as. \ \text{remdups} \ (\text{map} \ \text{snd} \ (\lambda (a, b). \ a \in \text{set as}) \ r))) \]
\[ (\lambda xs \ ys. \ (\text{filter} \ (\lambda x. \ x \notin \text{set ys}) \ xs) @ ys) \]
\[ (\lambda x \ xs. \ x \in \text{set xs}) \]

\[ \text{lemma } rtrancl-list-impl:\]
\[ \text{set} \ (\text{rtrancl-list-impl} \ r \ as) = \{ b. \ \exists \ a \in \text{set as}. \ (a, b) \in (\text{set r})^* \} \]
\[ \langle \text{proof} \rangle \]

\[ \text{lemma } trancl-list-impl:\]
\[ \text{set} \ (\text{trancl-list-impl} \ r \ as) = \{ b. \ \exists \ a \in \text{set as}. \ (a, b) \in (\text{set r})^+ \} \]
\[ \langle \text{proof} \rangle \]

### 2.2 Precomputing Closures for Single States

Storing all relevant entries is done by mapping all left-hand sides of the relation to their closure. To avoid redundant entries, \( \text{remdups} \) is used.

\[ \text{definition } \text{memo-list-rtrancl} :: ('a \times 'a) \text{ list } \Rightarrow ('a \Rightarrow 'a \text{ list}) \]
\[ \text{where} \]
memo-list-rtrancl r =
(let
  tr = rtrancl-list-impl r;
  rm = map (λa. (a, tr a)) ((remdups ◦ map fst) r)
in
  (λa. case map-of rm a of
    None ⇒ [a]
    | Some as ⇒ as))

lemma memo-list-rtrancl:
  set (memo-list-rtrancl r a) = \{ b. (a, b) ∈ (set r)\} (is ?l = ?r)
⟨proof⟩

definition memo-list-trancl :: ('a × 'a) list ⇒ ('a ⇒ 'a list)
where
memo-list-trancl r =
(let
  tr = trancl-list-impl r;
  rm = map (λa. (a, tr a)) ((remdups ◦ map fst) r)
in
  (λa. case map-of rm a of
    None ⇒ []
    | Some as ⇒ as))

lemma memo-list-trancl:
  set (memo-list-trancl r a) = \{ b. (a, b) ∈ (set r)\} (is ?l = ?r)
⟨proof⟩

end

3 Utility Functions and Lemmas

theory Utility
imports Main
begin

3.1 Miscellaneous

lemma infinite-imp-elem: ¬ finite A ⇒ ∃ x. x ∈ A
⟨proof⟩

lemma inf-pigeonhole-principle:
  assumes ∀ k::nat. ∃ i<n::nat. f k i
  shows ∃ i<n. ∀ k. ∃ k'≥k. f k' i
⟨proof⟩

lemma map-upt-Suc: map f [0 ..< Suc n] = f 0 # map (λ i. f (Suc i)) [0 ..< n]
⟨proof⟩
lemma map-upt-add: map f [0..<n + m] = map f [0..<n] @ map (λ i. f (i + n)) [0..<m]
⟨proof⟩

lemma map-upt-split: assumes i: i < n
  shows map f [0..<n] = map f [0..<i] @ f i # map (λ j. f (j + Suc i)) [0..<n - Suc i]
⟨proof⟩

lemma all-Suc-conv:
  (∀ i<Suc n. P i) ↔ P 0 ∧ (∀ i<n. P (Suc i)) (is ?l = ?r)
⟨proof⟩

lemma ex-Suc-conv:
  (∃ i<Suc n. P i) ↔ P 0 ∨ (∃ i<n. P (Suc i)) (is ?l = ?r)
⟨proof⟩

fun sorted-list-subset :: 'a :: linorder list ⇒ 'a list ⇒ 'a option where
  sorted-list-subset (a # as) (b # bs) =
  if a = b then sorted-list-subset as bs
  else if a > b then sorted-list-subset (a # as) bs
  else Some a
| sorted-list-subset [] - = None
| sorted-list-subset (a # -) [] = Some a

lemma sorted-list-subset:
  assumes sorted as and sorted bs
  shows (sorted-list-subset as bs = None) = (set as ⊆ set bs)
⟨proof⟩

lemma zip-nth-conv: length xs = length ys ⇒ zip xs ys = map (λ i. (xs ! i, ys ! i)) [0..<length ys]
⟨proof⟩

lemma nth-map-conv:
  assumes length xs = length ys
  and ∀ i<length xs. f (xs ! i) = g (ys ! i)
  shows map f xs = map g ys
⟨proof⟩

lemma listsum-0: [∀ x. x ∈ set xs ⇒ x = 0] ⇒ listsum xs = 0
⟨proof⟩

lemma foldr-foldr-concat: foldr (foldr f) m a = foldr f (concat m) a
⟨proof⟩

lemma listsum-double-concat:
  fixes f :: 'b ⇒ 'c ⇒ 'a :: comm-monoid-add and g as bs
  shows listsum (concat (map (λ i. map (λ j. f i j + g i j) as) bs))
\begin{verbatim}
  = listsum (concat (map (\lambda i. map (\lambda j. f i j) as) bs)) + 
  listsum (concat (map (\lambda i. map (\lambda j. g i j) as) bs))
  ⟨proof⟩

 fun max-list :: nat list ⇒ nat where
  max-list [] = 0 |
  max-list (x # xs) = max x (max-list xs)

 lemma max-list: x ∈ set xs ⇒ x ≤ max-list xs
  ⟨proof⟩

 lemma max-list-mem: xs ≠ [] ⇒ max-list xs ∈ set xs
  ⟨proof⟩

 lemma max-list-set: max-list xs = (if set xs = {} then 0 else (THE x. x ∈ set xs
  ∧ (∀ y ∈ set xs. y ≤ x)))
  ⟨proof⟩

 lemma max-list-eq-set: set xs = set ys ⇒ max-list xs = max-list ys
  ⟨proof⟩

end

4 Accessing Values via Keys

theory RBT-Map-Set-Extension
imports
  ../Collections/ICF/impl/RBTMapImpl
  ../Collections/ICF/impl/RBTSetImpl
  ../Matrix/Utility
begin
  We provide two extensions of the red black tree implementation.
  The first extension provides two convenience methods on sets which are
  represented by red black trees: a check on subsets and the big union operator.
  The second extension is to provide two operations elem-list-to-rm and
  rm-set-lookup which can be used to index a set of values via keys. More
  precisely, given a list of values of type 'v and a key function of type 'k ⇒ 'v
  ⇒ 'k, elem-list-to-rm will generate a map of type 'k ⇒ 'v set. Then with
  rs-set-lookup we can efficiently access all values which match a given key.

4.1 Subset and Union

For the subset operation \( r \subseteq s \) we provide two implementations. The first
one (rs-subset) traverses over \( r \) and then performs membership tests \( ∈ \) \( s \).
Its complexity is \( O(|r| \cdot \log(|s|)) \). The second one (rs-subset-list) generates
sorted lists for both \( r \) and \( s \) and then linearly checks the subset condition.
Its complexity is \( O(|r| + |s|) \).
\end{verbatim}
As union operator we use the standard fold function. Note that the order of the union is important so that new sets are added to the big union.

**definition rs-subset :: ('a :: linorder) rs ⇒ 'a rs ⇒ 'a option**
where
rs-subset as bs = rs.iteratei
   as
   (λ maybe. case maybe of None ⇒ True | Some - ⇒ False)
   (λ a -. if rs.memb a bs then None else Some a)
None

**lemma rs-subset [simp]:**
rs-subset as bs = None ←→ rs.α as ⊆ rs.α bs
⟨proof⟩

**definition rs-subset-list :: ('a :: linorder) rs ⇒ 'a rs ⇒ 'a option**
where
rs-subset-list as bs = sorted-list-subset (rs.to-sorted-list as) (rs.to-sorted-list bs)

**lemma rs-subset-list [simp]:**
rs-subset-list as bs = None ←→ rs.α as ⊆ rs.α bs
⟨proof⟩

**definition rs-Union :: ('q :: linorder) rs list ⇒ 'q rs**
where
rs-Union = foldl rs.union (rs.empty ()

**lemma rs-Union [simp]:**
rs.α (rs-Union qs) = ∪ (rs.α ' set qs)
⟨proof⟩

### 4.2 Grouping Values via Keys

The functions to produce the index (*elem-list-to-rm*) and the lookup function (*rm-set-lookup*) are straight-forward, however it requires some tedious reasoning that they perform as they should.

**fun elem-list-to-rm :: ('d ⇒ 'k :: linorder) ⇒ 'd list ⇒ ('k, 'd list) rm**
where
elem-list-to-rm key [] = rm.empty () |
  elem-list-to-rm key (d # ds) =
    (let
      rm = elem-list-to-rm key ds;
      k = key d
    in
      (case rm.α rm k of
      None ⇒ rm.update-dj k [d] rm
      | Some data ⇒ rm.update k (d # data) rm))

**definition rm-set-lookup rm = (λ a. (case rm.α rm a of None ⇒ [] | Some rules**
lemma \textit{rm-to-list-empty} [simp]:
\[ \text{rm.to-list (rm.empty ())} = [] \]

locale \textit{rm-set} =
  fixes \textit{rm} :: (‘k :: linorder, ’d list) rm
  and \textit{key} :: ’d ⇒ ’k
  and \textit{data} :: ’d set
  assumes \textit{rm-set-lookup}: \( \{d \in \text{data}. \text{key} d = k\} \)
begin

lemma \textit{data-lookup}:
\[ \text{data} = \bigcup \{ \text{set (rm-set-lookup rm k)} \mid k. \text{True}\} \]

lemma \textit{finite-data}:
finite \text{data}

end

interpretation elem-list-to-rm: \textit{rm-set elem-list-to-rm key ds key set ds}

end

\section{Closure Computation via Red Black Trees}

\textbf{theory} Transitive-Closure-RBT-Impl
\textbf{imports}
  Transitive-Closure-Impl
  RBT-Map-Set-Extension
begin
  We provide two algorithms to compute the reflexive transitive closure
  which internally work on red black trees. Therefore, the carrier has to be
  linear ordered. The first one (\textit{rtrancl-rbt-impl}) computes the closure on
demand for a given set of initial states. The second one (\textit{memo-rbt-rtrancl})
precomputes the closure for each individual state, stores the results, and
then only does a look-up.

  For the transitive closure there are the corresponding algorithms \textit{trancl-rbt-impl}
  and \textit{memo-rbt-trancl}
5.1 Computing Closures from Sets On-The-Fly

The algorithms are based on the generic algorithms \texttt{rtrancl-impl} and \texttt{trancl-impl} using red black trees. To compute the successors efficiently, all successors of a state are collected and stored in a red black tree map by using \textit{elem-list-to-rm}. Then, to lift the successor relation for single states to lists of states, all results are united using \textit{rs-Union}. The rest is standard.

\textbf{interpretation} \texttt{set-access \( \lambda \) as bs. rs.union bs (rs.from-list as) rs.\( \alpha \) rs.memb rs.empty ()}
\texttt{⟨proof⟩}

\textbf{abbreviation} \texttt{rm-succ \( \langle 'a :: \text{linorder} \times 'a \rangle \text{ list} \Rightarrow 'a \text{ list} \Rightarrow 'a \text{ list} \)}
\texttt{where}
\texttt{rm-succ \( \equiv (\lambda r. \text{let rm} = \text{elem-list-to-rm} \text{ fst} r \text{ in}) (\lambda \text{ as. rs.to-list (rs-Union (map (\lambda a. \text{rs.from-list (map \text{snd (rm-set-lookup rm} a)) as)}) as)))} \texttt{⟩}

\textbf{definition} \texttt{rtrancl-rbt-impl \( \langle 'a :: \text{linorder} \times 'a \rangle \text{ list} \Rightarrow 'a \text{ list} \Rightarrow 'a \text{ rs} \)}
\texttt{where}
\texttt{rtrancl-rbt-impl \( = \text{rtrancl-impl} \text{ rm-succ} (\lambda \text{ as bs. rs.union bs (rs.from-list as)}) \text{ rs.memb (rs.empty ()}) \)}

\textbf{definition} \texttt{trancl-rbt-impl \( \langle 'a :: \text{linorder} \times 'a \rangle \text{ list} \Rightarrow 'a \text{ list} \Rightarrow 'a \text{ rs} \)}
\texttt{where}
\texttt{trancl-rbt-impl \( = \text{trancl-impl} \text{ rm-succ} (\lambda \text{ as bs. rs.union bs (rs.from-list as)}) \text{ rs.memb (rs.empty ()}) \)}

\textbf{lemma} \texttt{rtrancl-rbt-impl:}
\texttt{rs.\( \alpha \) (rtrancl-rbt-impl r as) = \{b. \exists a \in \text{set as.} (a,b) \in (\text{set} r)^*\}}
\texttt{⟨proof⟩}

\textbf{lemma} \texttt{trancl-rbt-impl:}
\texttt{rs.\( \alpha \) (trancl-rbt-impl r as) = \{b. \exists a \in \text{set as.} (a,b) \in (\text{set} r)^+\}}
\texttt{⟨proof⟩}

5.2 Precomputing Closures for Single States

Storing all relevant entries is done by mapping all left-hand sides of the relation to their closure. Since we assume a linear order on the carrier, for the lookup we can use maps that are implemented as red black trees.

\textbf{definition} \texttt{memo-rbt-rtrancl \( \langle 'a :: \text{linorder} \times 'a \rangle \text{ list} \Rightarrow ('a \Rightarrow 'a \text{ rs}) \)}
\texttt{where}
\texttt{memo-rbt-rtrancl r =}
\texttt{(let}
\texttt{tr = rtrancl-rbt-impl r;}
\texttt{rm = rm.to-map (map (\lambda a. tr [a]) ((rs.to-list \circ rs.from-list \circ map \text{fst}) r))}
\texttt{in}
\texttt{⟨proof⟩}

(\lambda a. \text{case } rm.\text{lookup } a \text{ of} \\
None \Rightarrow rs.\text{from-list } [a] \\
| \text{Some } as \Rightarrow as))

\text{lemma memo-rbt-rtrancl:} \\
\quad rs.\alpha (\text{memo-rbt-rtrancl } r \ a) = \{ b. \ (a, b) \in (set \ r)^* \} \ (\text{is } ?l = ?r) \\
\langle \text{proof} \rangle

\text{definition memo-rbt-trancl :: } ('a :: \text{linorder } \times \ 'a) \text{ list } \Rightarrow ('a \Rightarrow 'a \ rs) \\
\text{where} \\
\quad \text{memo-rbt-trancl } r = \\
\quad \quad \text{(let} \\
\quad \quad \quad \ tr = \text{trancl-rbt-impl } r; \\
\quad \quad \quad \ rm = \text{rm.to-map } (\text{map } (\lambda a. \ (a, tr \ [a])) ((\text{rs.to-list } \circ \text{rs.from-list } \circ \text{map } \text{fst}) \ r)) \\
\quad \quad \text{in } (\lambda a. \\
\quad \quad \quad \text{(case } rm.\text{lookup } a \text{ of} \\
\quad \quad \quad \quad None \Rightarrow rs.\text{empty } () \\
\quad \quad \quad \quad | \text{Some } as \Rightarrow as)))

\text{lemma memo-rbt-trancl:} \\
\quad rs.\alpha (\text{memo-rbt-trancl } r \ a) = \{ b. \ (a, b) \in (set \ r)^+ \} \ (\text{is } ?l = ?r) \\
\langle \text{proof} \rangle

\text{end}

6 Computing Images of Finite Transitive Closures

definition Finite-Transitive-Closure-Simprocs
\text{imports Transitive-Closure-List-Impl}
begin

\text{lemma rtrancl-Image-eq:} \\
\quad \text{assumes } r = \text{set } r' \text{ and } x = \text{set } x' \\
\quad \text{shows } r^* \ "x = \text{set } (\text{rtrancl-list-impl } r' \ x') \\
\langle \text{proof} \rangle

\text{lemma trancl-Image-eq:} \\
\quad \text{assumes } r = \text{set } r' \text{ and } x = \text{set } x' \\
\quad \text{shows } r^+ \ "x = \text{set } (\text{trancl-list-impl } r' \ x') \\
\langle \text{proof} \rangle

6.1 A Simproc for Computing the Images of Finite Transitive Closures

\langle ML \rangle
6.2 Example

The images of (reflexive) transitive closures are computed by evaluation.

\[
\text{lemma} \quad \{(1::nat, 2), (2, 3), (3, 4), (4, 5)\}^* \cdot 1 \cdot \{1\} = \{1, 2, 3, 4, 5\} \\
\{(1::nat, 2), (2, 3), (3, 4), (4, 5)\}^+ \cdot 1 \cdot \{1\} = \{2, 3, 4, 5\}
\]

(\textit{proof})

Evaluation does not allow for free variables and thus fails in their presence.

\[
\text{lemma} \quad \{(x, y)\}^* \cdot \{x\} = \{x, y\}
\]

(\textit{proof})

end

References
