Executable Transitive Closures of Finite Relations

Christian Sternagel and René Thiemann

February 23, 2021

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 [2], where reflexive transitive closures of large tree automata have to be computed.

Contents

1 A Generic Work-List Algorithm \hspace{1cm} 2
  1.1 Bounded Reachability \hspace{1cm} 2
  1.2 Reflexive Transitive Closure and Transitive closure \hspace{1cm} 3

2 Closure Computation using Lists \hspace{1cm} 2
  2.1 Computing Closures from Sets On-The-Fly \hspace{1cm} 5
  2.2 Precomputing Closures for Single States \hspace{1cm} 5

3 Accessing Values via Keys \hspace{1cm} 6
  3.1 Subset and Union \hspace{1cm} 6
  3.2 Grouping Values via Keys \hspace{1cm} 7

4 Closure Computation via Red Black Trees \hspace{1cm} 8
  4.1 Computing Closures from Sets On-The-Fly \hspace{1cm} 9
  4.2 Precomputing Closures for Single States \hspace{1cm} 9

5 Computing Images of Finite Transitive Closures \hspace{1cm} 10
  5.1 A Simproc for Computing the Images of Finite Transitive
   Closures \hspace{1cm} 10
  5.2 Example \hspace{1cm} 11

*Supported by FWF (Austrian Science Fund) project P22767-N13.
1http://cl-informatik.uibk.ac.at/software/ceta
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
\[
\text{relpow-impl} :: \\
(\forall a \Rightarrow a \Rightarrow a) \Rightarrow \\
(\forall a \Rightarrow b \Rightarrow b) \Rightarrow (a \Rightarrow b \Rightarrow \text{bool}) \Rightarrow a \Rightarrow b \Rightarrow \text{nat} \Rightarrow b
\]
where
\[
\text{relpow-impl} \text{ succ un memb new have} 0 = un \text{ new have} | \\
\text{relpow-impl} \text{ succ un memb new have} (\text{Suc} \ m) = \\
(\text{if new} = [] \text{ then have} \\
\text{else let} \\
\text{maybe} = \text{succ new;} \\
\text{have'} = \text{un new have;} \\
\text{new'} = \text{filter} (\lambda n. \neg \text{memb n have'}) \text{ maybe} \\
in \text{relpow-impl succ un memb new' have' m})
\]

We need to know that the provided operations behave correctly.

locale set-access =
fixes un :: 'a list \Rightarrow 'b \\
and set-of :: 'b \Rightarrow 'a set \\
and memb :: 'a \Rightarrow 'b \Rightarrow \text{bool} \\
and empty :: 'b 
assumes un: set-of (un as bs) = set as \cup set-of bs \\
and memb: memb a bs \iff (a \in set-of bs) \\
and empty: set-of empty = {\}
locale set-access-succ = set-access un
for un :: 'a list ⇒ 'b ⇒ 'b +
fixes succ :: 'a list ⇒ 'a list
and rel :: ('a × 'a) set
assumes succ: set (succ as) = {b. ∃ a ∈ set as. (a, b) ∈ rel}
begin

abbreviation relpow-i ≡ relpow-impl succ un memb

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

lemma relpow-impl-main:
set-of (relpow-i new have n) =
{b | a b m. a ∈ set new ∧ m ≤ n ∧ (a, b) ∈ (rel ∩ {(a, b). b /∈ set-of have})} ^< m> ∪
set-of have
(is ?l new have n = ?r new have n)
⟨proof⟩

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

lemma relpow-impl:
set-of (relpow-i new empty n) = {b | a b m. a ∈ set new ∧ m ≤ n ∧ (a, b) ∈ rel} ^< m>
⟨proof⟩

end

1.2 Reflexive Transitive Closure and Transitive closure

Using 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 relpow-impl will abort the computation as soon as no new states are detected. Hence, there is no penalty in using this large bound.

definition rtrancl-impl ::
(('a × 'a) list ⇒ 'a list ⇒ 'a list) ⇒
('a list ⇒ 'b ⇒ 'b) ⇒ ('a ⇒ 'b ⇒ bool) ⇒ 'b ⇒ ('a × 'a) list ⇒ 'a list ⇒ 'b
where rtrancl-impl gen-succ un memb emp rel =
(let
succ = gen-succ rel;
n = length rel
in (λ as. relpow-impl succ un memb as emp n))

definition
\( \text{trancl-impl} :: \\
( (a \times a) \text{ list} \Rightarrow a \text{ list} \Rightarrow a \text{ list}) \Rightarrow \\
( a \text{ list} \Rightarrow b \Rightarrow b ) \Rightarrow ( a \Rightarrow b \Rightarrow \text{ bool}) \Rightarrow b \Rightarrow (a \times a) \text{ list} \Rightarrow a \text{ list} \Rightarrow b \\
\)

where

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

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

\[ (a, b) \in (\text{set rel})^* \iff (\exists \ n \leq \text{length rel}. (a, b) \in \text{set rel} \uparrow n) \text{ (is } \forall l = \exists r) \]

\[ \langle \text{proof} \rangle \]

locale set-access-gen = set-access un 
for un :: 'a list \Rightarrow b \Rightarrow b + 
fixes gen-succ :: (a \times a) \text{ list} \Rightarrow a \text{ list} \Rightarrow a \text{ list} 
assumes gen-succ: set (gen-succ rel as) = \{b. \exists a \in \text{set as}. (a, b) \in \text{set rel}\} 
begin 

abbreviation rtrancl-i \equiv \text{rtrancl-impl gen-succ un memb empty} 
abbreviation trancl-i \equiv \text{trancl-impl gen-succ un memb empty} 

lemma rtrancl-impl: 
set-of (rtrancl-i rel as) = \{b. (\exists a \in \text{set as}. (a, b) \in (\text{set rel})^*)\} 
\langle \text{proof} \rangle 

lemma trancl-impl: 
set-of (trancl-i rel as) = \{b. (\exists a \in \text{set as}. (a, b) \in (\text{set rel})^+)\} 
\langle \text{proof} \rangle 

end 

end

2 Closure Computation using Lists

theory Transitive-Closure-List-Impl 
imports Transitive-Closure-Impl 
begin 

We provide two algorithms for the computation of the reflexive transitive closure which internally work on lists. The first one (\( \text{rtrancl-list-impl} \)) 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 \(\text{trancl-list-impl}\) and \(\text{memo-list-trancl}\).

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

The algorithms are based on the generic algorithms \(\text{rtrancl-impl}\) and \(\text{trancl-impl}\) 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.

**Definition**

\[
\text{rtrancl-list-impl} :: (\text{'a} \times \text{'a}) \text{ list} \Rightarrow \text{'a list} \Rightarrow \text{'a list}
\]

where

\[
\text{rtrancl-list-impl} = \text{rtrancl-impl} (\lambda r \, \text{as} \, . \, \text{remdups} (\text{map} \, \text{snd} \, (\text{filter} \, (\lambda (a, b), a \in \text{set as}) \, r)))
\]

\[
\text{let} \, \text{tr} = \text{rtrancl-list-impl} r;
\]

\[
\text{rm} = \text{map} \, (\lambda a. \, (a, \text{tr} \, [a])) \, ((\text{remdups} \circ \text{map} \, \text{fst}) \, r)
\]

\[
\text{in} \, \text{case} \, \text{map-of} \, \text{rm} \, \text{a of}
\]

**Definition**

\[
\text{trancl-list-impl} :: (\text{'a} \times \text{'a}) \text{ list} \Rightarrow \text{'a list} \Rightarrow \text{'a list}
\]

where

\[
\text{trancl-list-impl} = \text{trancl-impl} (\lambda r \, \text{as} \, . \, \text{remdups} (\text{map} \, \text{snd} \, (\text{filter} \, (\lambda (a, b), a \in \text{set as}) \, r)))
\]

\[
\text{let} \, \text{tr} = \text{trancl-list-impl} r;
\]

\[
\text{rm} = \text{map} \, (\lambda a. \, (a, \text{tr} \, [a])) \, ((\text{remdups} \circ \text{map} \, \text{fst}) \, r)
\]

\[
\text{in} \, \text{case} \, \text{map-of} \, \text{rm} \, \text{a of}
\]

**Lemma**

\[
\text{rtrancl-list-impl}:
\]

\[
\text{set} \, (\text{rtrancl-list-impl} \, r \, \text{as}) = \{ b, \exists a \in \text{set as} \, (a, b) \in (\text{set } r^+) \}
\]

**(proof)**

**Lemma**

\[
\text{trancl-list-impl}:
\]

\[
\text{set} \, (\text{trancl-list-impl} \, r \, \text{as}) = \{ b, \exists a \in \text{set as} \, (a, b) \in (\text{set } r^+) \}
\]

**(proof)**

### 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.

**Definition**

\[
\text{memo-list-rtrancl} :: (\text{'a} \times \text{'a}) \text{ list} \Rightarrow (\text{'a} \Rightarrow \text{'a list})
\]

where

\[
\text{let} \, \text{tr} = \text{rtrancl-list-impl} r;
\]

\[
\text{rm} = \text{map} \, (\lambda a. \, (a, \text{tr} \, [a])) \, ((\text{remdups} \circ \text{map} \, \text{fst}) \, r)
\]

\[
\text{in} \, (\lambda a. \, \text{case} \, \text{map-of} \, \text{rm} \, \text{a of}
\]

5
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 o 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 Accessing Values via Keys

theory RBT-Map-Set-Extension
imports
  Collections.RBTMapImpl
  Collections.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.

  3.1 Subset and Union

  For the subset operation r ⊆ 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 \( \mathcal{O}(|r| + |s|) \).

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** \( rs\text{-subset} :: (\text{'a :: linorder}) \Rightarrow \text{'a} \Rightarrow \text{'a} \Rightarrow \text{'a option} \)

where
\[
rs\text{-subset} \; as \; bs = rs.\text{iteratei} \; as \\
\quad \lambda \; \text{maybe}. \; \text{case} \; \text{maybe} \; \text{of} \; \text{None} \Rightarrow \text{True} \; | \; \text{Some} \; - \Rightarrow \text{False} \\
\quad \lambda \; a \; - , \; \text{if} \; rs.\text{memb} \; a \; bs \; \text{then} \; \text{None} \; \text{else} \; \text{Some} \; a \; \\
\quad \text{None}
\]

**lemma** \( rs\text{-subset} [\text{simp}]: \)
\[
rs\text{-subset} \; as \; bs = \text{None} \leftrightarrow rs.\alpha \; as \subseteq rs.\alpha \; bs
\]

**definition** \( rs\text{-subset-list} :: (\text{'a :: linorder}) \Rightarrow \text{'a} \Rightarrow \text{'a} \Rightarrow \text{'a option} \)

where
\[
rs\text{-subset-list} \; as \; bs = \text{sorted-list-subset} \; (rs.\text{to-sorted-list} \; as) \; (rs.\text{to-sorted-list} \; bs)
\]

**lemma** \( rs\text{-subset-list} [\text{simp}]: \)
\[
rs\text{-subset-list} \; as \; bs = \text{None} \leftrightarrow rs.\alpha \; as \subseteq rs.\alpha \; bs
\]

**definition** \( rs\text{-Union} :: (\text{'q :: linorder}) \Rightarrow \text{list} \Rightarrow \text{'q} \Rightarrow \text{'q} \Rightarrow \text{'q} \)

where
\[
rs\text{-Union} = \text{foldl} \; rs.\text{union} \; (rs.\text{empty} \; ())
\]

**lemma** \( rs\text{-Union} [\text{simp}]: \)
\[
rs.\alpha \; (rs\text{-Union} \; qs) = \bigcup \; (rs.\alpha \; \text{° set} \; qs)
\]

### 3.2 Grouping Values via Keys

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

**fun** \( \text{elem-list-to-rm} :: (\text{'d} \Rightarrow \text{'k :: linorder}) \Rightarrow \text{'d} \; \text{list} \Rightarrow \text{('k, 'd list) rm} \)

where
\[
\text{elem-list-to-rm} \; \text{key} \; [] = \text{rm.} \text{empty} \; () \; | \; \\
\text{elem-list-to-rm} \; \text{key} \; (d \neq ds) = \\
\quad \text{let} \; \text{rm} = \text{elem-list-to-rm} \; \text{key} \; ds; \ \\
\quad k = \text{key} \; d \; \\
\quad \text{in} \\
\quad (\text{case} \; \text{rm} \; \text{°} \; \text{rm} \; k \; \text{af} \\
\quad \quad \text{None} \Rightarrow \text{rm.} \text{update-dj} \; k \; [d] \; \text{rm}
\]

7
Some data ⇒ rm.update k (d ≠ data) rm)

definition rm-set-lookup rm = (λ a. (case rm.α rm a of None ⇒ [] | Some rules ⇒ rules))

lemma rm-to-list-empty [simp]:
rm.to-list (rm.empty ()) = []
⟨proof⟩

locale rm-set =
fixes rm :: ('k :: linorder, 'd list) rm
and key :: 'd ⇒ 'k
and data :: 'd set
assumes rm-set-lookup: \ k. set (rm-set-lookup rm k) = {d ∈ data. key d = k}
begin

lemma data-lookup: data = \ k. set (rm-set-lookup rm k) \ k. True (is = ?R)
⟨proof⟩

lemma finite-data:
finite data
⟨proof⟩

end

interpretation elem-list-to-rm: rm-set elem-list-to-rm key ds key set ds
⟨proof⟩

end

4 Closure Computation via Red Black Trees

theory Transitive-Closure-RBT-Impl
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 (rtrancl-rbt-impl) computes the closure on demand for a given set of initial states. The second one (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 trancl-rbt-impl and memo-rbt-trancl.
4.1 Computing Closures from Sets On-The-Fly

The algorithms are based on the generic algorithms \( \text{rtrancl-impl} \) and \( \text{trancl-impl} \) 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 \( \text{elem-list-to-rm} \). Then, to lift the successor relation for single states to lists of states, all results are united using \( \text{rs-Union} \). The rest is standard.

\[
\text{interpretation set-access } \lambda \text{ as bs. } \text{rs.union bs (rs.from-list as) rs.\alpha rs.memb rs.empty ()}
\]

\[
\text{abbreviation rm-succ :: ('}a::\text{linorder } \times \text{'a list)} \Rightarrow \text{'a list } \Rightarrow \text{'a list}
\]

\[
\text{where rm-succ } \equiv (\lambda \text{ r. let rm = elem-list-to-rm fst r in (\lambda as. rs.to-list ((rs-Union (map (\lambda a. rs.from-list (map snd (rm-set-lookup rm a)))) as)))})
\]

\[
\text{definition rtrancl-rbt-impl :: ('}a::\text{linorder } \times \text{'a list)} \Rightarrow \text{'a list } \Rightarrow \text{'a rs}
\]

\[
\text{where rtrancl-rbt-impl } = \text{ rtrancl-impl rm-succ}
\]

\[
(\lambda as. \text{rs.union bs (rs.from-list as) rs.memb (rs.empty ())})
\]

\[
\text{definition trancl-rbt-impl :: ('}a::\text{linorder } \times \text{'a list)} \Rightarrow \text{'a list } \Rightarrow \text{'a rs}
\]

\[
\text{where trancl-rbt-impl } = \text{ trancl-impl rm-succ}
\]

\[
(\lambda as. \text{rs.union bs (rs.from-list as) rs.memb (rs.empty ())})
\]

\[
\text{lemma rtrancl-rbt-impl:}
\]

\[
\text{rs.\alpha (rtrancl-rbt-impl r as) } = \{b. \exists a \in \text{set as. } (a,b) \in (\text{set r})^+\}
\]

\[
\langle \text{proof} \rangle
\]

\[
\text{lemma trancl-rbt-impl:}
\]

\[
\text{rs.\alpha (trancl-rbt-impl r as) } = \{b. \exists a \in \text{set as. } (a,b) \in (\text{set r})^+\}
\]

\[
\langle \text{proof} \rangle
\]

4.2 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.

\[
\text{definition memo-rbt-rtrancl :: ('}a::\text{linorder } \times \text{'a list)} \Rightarrow ('}a \Rightarrow \text{'a rs})
\]

\[
\text{where memo-rbt-rtrancl r = (let tr = rtrancl-rbt-impl r; rm = rm.to-map (map (\lambda a. (a, tr [a])) ((rs.to-list \circ rs.from-list \circ map fst) r)) in}
\]

9
\(\lambda a. \text{case rm.lookup a rm of}
\quad \text{None} \Rightarrow rs\text{.from-list [a]}
\quad | \text{Some as} \Rightarrow as)\)

**lemma** memo-rbt-rtrancl:

\(rs.a (\text{memo-rbt-rtrancl } r \ a) = \{ b. (a, b) \in (\text{set } r)^* \} \ (\text{is } ?l = ?r)\)

\(\langle\text{proof}\rangle\)

**definition** memo-rbt-trancl :: \((a :: \text{linorder} \times a)\ \text{list} \Rightarrow (a \Rightarrow a)\) rs

**where**

memo-rbt-trancl \( r = \)

(let

\( tr = \text{trancl-rbt-impl } r; \)

\( rm = \text{rm.to-map (map (\lambda } a. (a, tr \ [a])) ((rs.to-list o rs.from-list o \text{map } \text{fst}) r))} \)

in (\( \lambda \ a. \)

(case rm.lookup a rm of

\( \text{None} \Rightarrow rs\text{.empty ()}
\quad | \text{Some as} \Rightarrow as))\)

**lemma** memo-rbt-trancl:

\(rs.a (\text{memo-rbt-trancl } r \ a) = \{ b. (a, b) \in (\text{set } r)^+ \} \ (\text{is } ?l = ?r)\)

\(\langle\text{proof}\rangle\)

end

5 Computing Images of Finite Transitive Closures

**theory** Finite-Transitive-Closure-Simprocs

**imports** Transitive-Closure-List-Impl

**begin**

**lemma** rtrancl-Image-eq:

**assumes** \( r = \text{set } r' \ and \ x = \text{set } x' \)

**shows** \( r^{*} \ x' = \text{set } (\text{rtrancl-list-impl } r' x') \)

\(\langle\text{proof}\rangle\)

**lemma** trancl-Image-eq:

**assumes** \( r = \text{set } r' \ and \ x = \text{set } x' \)

**shows** \( r^{+} \ x' = \text{set } (\text{trancl-list-impl } r' x') \)

\(\langle\text{proof}\rangle\)

5.1 A Simproc for Computing the Images of Finite Transitive Closures

\(\langle\text{ML}\rangle\)
5.2 Example

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

lemma

\{(1::nat, 2), (2, 3), (3, 4), (4, 5)\}^* \{1\} = \{1, 2, 3, 4, 5\}
\{(1::nat, 2), (2, 3), (3, 4), (4, 5)\}^+ \{1\} = \{2, 3, 4, 5\}

⟨proof⟩

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

lemma

\{(x, y)\}^* \{x\} = \{x, y\}

⟨proof⟩

end

References
