Executable Transitive Closures of Finite Relations*

Christian Sternagel and René Thiemann

March 17, 2025

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

*Supported by FWF (Austrian Science Fund) project P22767-N13.

[1] http://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 *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 *relpow-impl* that computes all states that are reachable from an initial set of states *new* by at most n steps. The algorithm also stores a set of states that have already been visited *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 ⇒ 'a list) ⇒
      ('a list ⇒ 'b ⇒ 'b) ⇒ ('a ⇒ 'b ⇒ bool) ⇒ 'a list ⇒ 'b ⇒ nat ⇒ '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 (λ n. ¬ 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 ⇒ 'b ⇒ 'b
    and set-of :: 'b ⇒ 'a set
    and memb :: 'a ⇒ 'b ⇒ bool
    and empty :: 'b
  assumes un: set-of (un as bs) = set as ∪ set-of bs
    and memb: memb a bs ⟷ (a ∈ 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* :: $('a \times 'a)$ *set*
  assumes *succ*: *set* (*succ as*) = $\{b.\ \exists\ a \in$ *set as*. $(a, b) \in$ *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*) =
    $\{b \mid a\ b\ m.\ a \in$ *set new* $\wedge\ m \le n \wedge (a, b) \in ($*rel* $\cap \{(a, b).\ b \notin$ *set-of have*$\})$ ^^ $m\} \cup$
    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*) = $\{b \mid a\ b\ m.\ a \in$ *set new* $\wedge\ m \le n \wedge (a, b) \in$ *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 ::
    $(('a \times 'a)$ *list* ⇒ $'a$ *list* ⇒ $'a$ *list*) ⇒
      ($'a$ *list* ⇒ $'b$ ⇒ $'b$) ⇒ ($'a$ ⇒ $'b$ ⇒ *bool*) ⇒ $'b$ ⇒ $('a \times 'a)$ *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

trancl-impl ::
  $(('a \times 'a)\ list \Rightarrow 'a\ list \Rightarrow 'a\ list) \Rightarrow$
   $('a\ list \Rightarrow 'b \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'b \Rightarrow bool) \Rightarrow 'b \Rightarrow ('a \times 'a)\ list \Rightarrow 'a\ list \Rightarrow 'b$
where
 trancl-impl gen-succ un memb emp rel =
  (*let*
   succ = *gen-succ rel*;
   n = *length rel*
  in (λ *as. relpow-impl succ un memb* (*succ as*) *emp n*))

The soundness of both *rtrancl-impl* and *trancl-impl* follows from the soundness of *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.

lemma *rtrancl-finite-relpow*:
 $(a,\ b) \in (set\ rel)^* \longleftrightarrow (\exists\ n \leq length\ rel.\ (a,\ b) \in set\ rel\ \widehat{}\widehat{}\ n)$ (**is** *?l* = *?r*)
⟨*proof*⟩

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

abbreviation *rtrancl-i* $\equiv$ *rtrancl-impl gen-succ un memb empty*
abbreviation *trancl-i* $\equiv$ *trancl-impl gen-succ un memb empty*

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

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

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 (*rtrancl-list-impl*) computes the closure on demand for a given set of initial states. The second one (*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 *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 *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 *filter* in the union operation is preferred over *remdups* since by construction the latter set will not contain duplicates.

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

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

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

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

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, *remdups* is used.

definition *memo-list-rtrancl* :: $('a \times 'a)$ *list* $\Rightarrow$ $('a \Rightarrow 'a\ list)$
where
memo-list-rtrancl r =
(*let*
tr = *rtrancl-list-impl r*;
rm = *map* (λ*a.* (*a*, *tr* [*a*])) ((*remdups* $\circ$ *map fst*) *r*)
in
(λ*a. 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* (λ*a*. (*a*, *tr* [*a*])) ((*remdups* ∘ *map fst*) *r*)
in
(λ*a*. *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 *'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 $r \subseteq s$ we provide two implementations. The first one (*rs-subset*) traverses over r and then performs membership tests $\in s$. Its complexity is $\mathcal{O}(|r| \cdot log(|s|))$. 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-subset* :: $('a :: linorder)$ *rs* $\Rightarrow$ *'a rs* $\Rightarrow$ *'a option*
where
  rs-subset as bs = *rs.iteratei*
    as
    (λ *maybe. case maybe of None* $\Rightarrow$ *True* | *Some* - $\Rightarrow$ *False*)
    (λ *a* -. *if rs.memb a bs then None else Some a*)
    None

lemma *rs-subset* [*simp*]:
  rs-subset as bs = *None* $\longleftrightarrow$ *rs*.α *as* $\subseteq$ *rs*.α *bs*
⟨*proof*⟩

definition *rs-subset-list* :: $('a :: linorder)$ *rs* $\Rightarrow$ *'a rs* $\Rightarrow$ *'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* $\longleftrightarrow$ *rs*.α *as* $\subseteq$ *rs*.α *bs*
  ⟨*proof*⟩

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

lemma *rs-Union* [*simp*]:
  rs.α (*rs-Union qs*) = $\bigcup$ (*rs*.α ‘ *set qs*)
⟨*proof*⟩

3.2 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* :: $('d \Rightarrow 'k :: linorder) \Rightarrow$ *'d list* $\Rightarrow$ *('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 $\Rightarrow$ *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* ⇒ *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 = ⋃ {*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 *rtrancl-impl* and *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 *elem-list-to-rm*. Then, to lift the successor relation for single states to lists of states, all results are united using *rs-Union*. The rest is standard.

interpretation *set-access* λ *as bs. rs.union bs* (*rs.from-list as*) *rs.*α *rs.memb rs.empty* ()
  $\langle proof \rangle$

abbreviation *rm-succ* :: ($'a$ :: *linorder* $\times$ $'a$) *list* $\Rightarrow$ $'a$ *list* $\Rightarrow$ $'a$ *list*
where
  rm-succ $\equiv$ (λ *r. let rm* = *elem-list-to-rm fst r in*
    (λ *as. rs.to-list* (*rs-Union* (*map* (λ *a. rs.from-list* (*map snd* (*rm-set-lookup rm a*))) *as*))))

definition *rtrancl-rbt-impl* :: ($'a$ :: *linorder* $\times$ $'a$) *list* $\Rightarrow$ $'a$ *list* $\Rightarrow$ $'a$ *rs*
where
  rtrancl-rbt-impl = *rtrancl-impl rm-succ*
    (λ *as bs. rs.union bs* (*rs.from-list as*)) *rs.memb* (*rs.empty* ())

definition *trancl-rbt-impl* :: ($'a$ :: *linorder* $\times$ $'a$) *list* $\Rightarrow$ $'a$ *list* $\Rightarrow$ $'a$ *rs*
where
  trancl-rbt-impl = *trancl-impl rm-succ*
    (λ *as bs. rs.union bs* (*rs.from-list as*)) *rs.memb* (*rs.empty* ())

lemma *rtrancl-rbt-impl*:
  rs.α (*rtrancl-rbt-impl r as*) = $\{b.\ \exists\ a \in set\ as.\ (a,b) \in (set\ r)^*\}$
  $\langle proof \rangle$

lemma *trancl-rbt-impl*:
  rs.α (*trancl-rbt-impl r as*) = $\{b.\ \exists\ a \in set\ as.\ (a,b) \in (set\ r)^+\}$
  $\langle proof \rangle$

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.

definition *memo-rbt-rtrancl* :: ($'a$ :: *linorder* $\times$ $'a$) *list* $\Rightarrow$ ($'a$ $\Rightarrow$ $'a$ *rs*)
where
  memo-rbt-rtrancl r =
    (*let*
      tr = *rtrancl-rbt-impl r*;
      rm = *rm.to-map* (*map* (λ *a.* (*a*, *tr* [*a*])) ((*rs.to-list* $\circ$ *rs.from-list* $\circ$ *map fst*) *r*))
    in

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

lemma *memo-rbt-rtrancl*:
$rs.\alpha$ (*memo-rbt-rtrancl r a*) $= \{b.\ (a,\ b) \in (set\ r)^*\}$ (**is** $?l = ?r$)
⟨*proof*⟩

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

lemma *memo-rbt-trancl*:
$rs.\alpha$ (*memo-rbt-trancl r a*) $= \{b.\ (a,\ b) \in (set\ r)^+\}$ (**is** $?l = ?r$)
⟨*proof*⟩

end

5 Computing Images of Finite Transitive Closures

theory *Finite-Transitive-Closure-Simprocs*
imports *Transitive-Closure-List-Impl*
begin

lemma *rtrancl-Image-eq*:
assumes $r = set\ r'$ **and** $x = set\ x'$
shows r^* `` $x = set\ (rtrancl\text{-}list\text{-}impl\ r'\ x')$
⟨*proof*⟩

lemma *trancl-Image-eq*:
assumes $r = set\ r'$ **and** $x = set\ x'$
shows r^+ `` $x = set\ (trancl\text{-}list\text{-}impl\ r'\ x')$
⟨*proof*⟩

5.1 A Simproc for Computing the Images of Finite Transitive Closures

⟨*ML*⟩

5.2 Example

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

lemma
$\{(1::nat,\ 2),\ (2,\ 3),\ (3,\ 4),\ (4,\ 5)\}^*$ `` $\{1\} = \{1,\ 2,\ 3,\ 4,\ 5\}$
$\{(1::nat,\ 2),\ (2,\ 3),\ (3,\ 4),\ (4,\ 5)\}^+$ `` $\{1\} = \{2,\ 3,\ 4,\ 5\}$
$\langle proof \rangle$

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

lemma
$\{(x,\ y)\}^*$ `` $\{x\} = \{x,\ y\}$
$\langle proof \rangle$

end

References

[1] P. Lammich and A. Lochbihler. The Isabelle collections framework. In *Proc. ITP'10*, volume 6172 of *LNCS*, pages 339–354, 2010.

[2] R. Thiemann and C. Sternagel. Certification of termination proofs using CeTA. In *Proc. TPHOLs'09*, volume 5674 of *LNCS*, pages 452–468, 2009.