Executable Transitive Closures of Finite Relations*

Christian Sternagel and René Thiemann

April 20, 2020

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 ::
    $('a\ list \Rightarrow 'a\ list) \Rightarrow$
      $('a\ list \Rightarrow 'b \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'b \Rightarrow bool) \Rightarrow 'a\ list \Rightarrow 'b \Rightarrow nat \Rightarrow 'b$
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* (λ *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 *set-of* :: $'b \Rightarrow 'a\ set$
    and *memb* :: $'a \Rightarrow 'b \Rightarrow bool$
    and *empty* :: $'b$
  assumes *un*: *set-of* (*un as bs*) = *set as* $\cup$ *set-of bs*
    and *memb*: *memb a bs* $\longleftrightarrow$ ($a \in$ *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 (*induction n arbitrary*: *have new*)
  case (*Suc n hhave nnew*)
  show *?case*
  proof (*cases nnew* = [])
    case *True*
    then show *?thesis* **by** *auto*
  next
    case *False*
    let *?have* = *set-of hhave*
    let *?new* = *set nnew*
    obtain *have new* **where** *hav*: *have* = *?have* **and** *new*: *new* = *?new* **by** *auto*
    let *?reln* = λ *m*. (*rel* ∩ {(*a*, *b*). *b* ∉ *new* ∧ *b* ∉ *have*}) ^^ *m*
    let *?rel* = λ *m*. (*rel* ∩ {(*a*, *b*). *b* ∉ *have*}) ^^ *m*
    have *idl*: *?l nnew hhave* (*Suc n*) =
      {*uu*. ∃ *a*. (∃ *aa*∈ *new*. (*aa*,*a*) ∈ *rel*) ∧ *a* ∉ *new* ∧ *a* ∉ *have* ∧ (∃ *m* ≤ *n*. (*a*, *uu*) ∈ *?reln m*)} ∪
      (*new* ∪ *have*)
      (**is** - = *?l1* ∪ (*?l2* ∪ *?l3*))
      by (*simp add*: *hav new False Let-def Suc*, *simp add*: *memb un succ*)
    let *?l* = *?l1* ∪ (*?l2* ∪ *?l3*)
    have *idr*: *?r nnew hhave* (*Suc n*) = {*b*. ∃ *a m*. *a* ∈ *new* ∧ *m* ≤ *Suc n* ∧ (*a*, *b*) ∈ *?rel m*} ∪ *have*
      (**is** - = (*?r1* ∪ *?r2*)) **by** (*simp add*: *hav new*)
    let *?r* = *?r1* ∪ *?r2*
    {
      fix *b*
      assume *b*: *b* ∈ *?l*
      have *b* ∈ *?r*
      proof (*cases b* ∈ *new* ∨ *b* ∈ *have*)
        case *True* **then show** *?thesis*
        proof

```
      assume b ∈ have then show ?thesis by auto
    next
      assume b: b ∈ new
      have b ∈ ?r1
        by (intro CollectI, rule exI, rule exI [of - 0], intro conjI, rule b, auto)
      then show ?thesis by auto
    qed
  next
    case False
    with b have b ∈ ?l1 by auto
    then obtain a2 a1 m where a2n: a2 ∉ new and a2h: a2 ∉ have and a1: a1 ∈ new
      and a1a2: (a1,a2) ∈ rel and m: m ≤ n and a2b: (a2,b) ∈ ?reln m by auto
    have b ∈ ?r1
      by (rule CollectI, rule exI, rule exI [of - Suc m], intro conjI, rule a1, simp add: m, rule relpow-Suc-I2, rule, rule a1a2, simp add: a2h, insert a2b, induct m arbitrary: a2 b, auto)
    then show ?thesis by auto
  qed
}
moreover
{
  fix b
  assume b: b ∈ ?r
  then have b ∈ ?l
  proof (cases b ∈ have)
    case True then show ?thesis by auto
  next
    case False
    with b have b ∈ ?r1 by auto
    then obtain a m where a: a ∈ new and m: m ≤ Suc n and ab: (a, b) ∈ ?rel m by auto
    have seq: ∃ a ∈ new. (a, b) ∈ ?rel m
      using a ab by auto
    obtain l where l: l = (LEAST m. (∃ a ∈ new. (a, b) ∈ ?rel m)) by auto
    have least: (∃ a ∈ new. (a, b) ∈ ?rel l)
      by (unfold l, rule LeastI, rule seq)
    have lm: l ≤ m unfolding l
      by (rule Least-le, rule seq)
    with m have ln: l ≤ Suc n by auto
    from least obtain a where a: a ∈ new
      and ab: (a, b) ∈ ?rel l by auto
    from ab [unfolded relpow-fun-conv]
    obtain f where fa: f 0 = a and fb: b = f l
      and steps: ⋀ i. i < l ⟹ (f i, f (Suc i)) ∈ ?rel 1 by auto
    {
      fix i
      assume i: i < l
```

```
          have main: f (Suc i) ∉ new
          proof
            assume new: f (Suc i) ∈ new
            let ?f = λ j. f (Suc i + j)
            have seq: (f (Suc i), b) ∈ ?rel (l − Suc i)
              unfolding relpow-fun-conv
            proof (rule exI[of - ?f], intro conjI allI impI)
              from i show f (Suc i + (l − Suc i)) = b
                unfolding fb by auto
            next
              fix j
              assume j < l − Suc i
              then have small: Suc i + j < l by auto
               show (?f j, ?f (Suc j)) ∈ rel ∩ {(a, b). b ∉ have} using steps [OF
small] by auto
            qed simp
            from i have small: l − Suc i < l by auto
            from seq new have ∃ a ∈ new. (a, b) ∈ ?rel (l − Suc i) by auto
            with not-less-Least [OF small [unfolded l]]
            show False unfolding l by auto
          qed
          then have (f i, f (Suc i)) ∈ ?reln 1
            using steps [OF i] by auto
        } note steps = this
        have ab: (a, b) ∈ ?reln l unfolding relpow-fun-conv
          by (intro exI conjI, insert fa fb steps, auto)
        have b ∈ ?l1 ∪ ?l2
        proof (cases l)
          case 0
          with ab a show ?thesis by auto
        next
          case (Suc ll)
          from relpow-Suc-D2 [OF ab [unfolded Suc]] a ln Suc
          show ?thesis by auto
        qed
        then show ?thesis by auto
      qed
    }
    ultimately show ?thesis
      unfolding idl idr by blast
  qed
qed (simp add: un)
```

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 −
```

```
  have id: rel ∩ {(a ,b). True} = rel by auto
  show ?thesis unfolding relpow-impl-main empty by (simp add: id)
qed

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
  trancl-impl ::
    (('a × 'a) list ⇒ 'a list ⇒ 'a list) ⇒
      ('a list ⇒ 'b ⇒ 'b) ⇒ ('a ⇒ 'b ⇒ bool) ⇒ 'b ⇒ ('a × 'a) list ⇒ 'a list ⇒ '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) ∈ (set rel)* ⟷ (∃ n ≤ length rel. (a, b) ∈ set rel ^^ n) (is ?l = ?r)
proof
  assume ?r
  then show ?l
    unfolding rtrancl-power by auto
next
  assume ?l
  from this [unfolded rtrancl-power]
    obtain n where ab: (a,b) ∈ set rel ^^ n ..
```

```
  obtain l where l: l = (LEAST n. (a,b) ∈ set rel ^^ n) by auto
  have ab: (a, b) ∈ set rel ^^ l unfolding l
    by (intro LeastI, rule ab)
  from this [unfolded relpow-fun-conv]
  obtain f where a: f 0 = a and b: f l = b
    and steps: ⋀ i. i < l ⟹ (f i, f (Suc i)) ∈ set rel by auto
  let ?hits = map (λ i. f (Suc i)) [0 ..< l]
  from steps have subset: set ?hits ⊆ snd ‘ set rel by force
  have l ≤ length rel
  proof (cases distinct ?hits)
    case True
    have l = length ?hits by simp
    also have ... = card (set ?hits) unfolding distinct-card [OF True] ..
    also have ... ≤ card (snd ‘ set rel) by (rule card-mono [OF - subset], auto)
    also have ... = card (set (map snd rel)) by auto
    also have ... ≤ length (map snd rel) by (rule card-length)
    finally show ?thesis by simp
  next
    case False
    from this [unfolded distinct-conv-nth]
    obtain i j where i: i < l and j: j < l and ij: i ≠ j and fij: f (Suc i) = f
(Suc j) by auto
    let ?i = min i j
    let ?j = max i j
    have i: ?i < l and j: ?j < l and fij: f (Suc ?i) = f (Suc ?j)
      and ij: ?i < ?j
      using i j ij fij unfolding min-def max-def by (cases i ≤ j, auto)
    from i j fij ij obtain i j where i: i < l and j: j < l and ij: i < j and fij: f
(Suc i) = f (Suc j) by blast
    let ?g = λ n. if n ≤ i then f n else f (n + (j − i))
    let ?l = l − (j − i)
    have abl: (a,b) ∈ set rel ^^ ?l
      unfolding relpow-fun-conv
    proof (rule exI [of - ?g], intro conjI impI allI)
      show ?g ?l = b unfolding b [symmetric] using j ij by auto
    next
      fix k
      assume k: k < ?l
      show (?g k, ?g (Suc k)) ∈ set rel
      proof (cases k < i)
        case True
        with i have k < l by auto
        from steps [OF this] show ?thesis using True by simp
      next
        case False
        then have ik: i ≤ k by auto
        show ?thesis
        proof (cases k = i)
          case True
```

then show *?thesis* **using** *ij fij steps* [*OF i*] **by** *simp*
next
case *False*
with *ik* **have** *ik*: $i < k$ **by** *auto*
then have *small*: $k + (j - i) < l$ **using** *k* **by** *auto*
show *?thesis* **using** *steps*[*OF small*] *ik* **by** *auto*
qed
qed
qed (*simp add*: *a*)
from *ij i* **have** *ll*: $?l < l$ **by** *auto*
have $l \leq ?l$ **unfolding** *l*
by (*rule Least-le*, *rule abl* [*unfolded l*])
with *ll* **have** *False* **by** *simp*
then show *?thesis* **by** *simp*
qed
with *ab* **show** *?r* **by** *auto*
qed

locale *set-access-gen* = *set-access un*
for *un* :: *'a list* ⇒ *'b* ⇒ *'b* +
fixes *gen-succ* :: (*'a* × *'a*) *list* ⇒ *'a list* ⇒ *'a list*
assumes *gen-succ*: *set* (*gen-succ rel as*) = {*b*. ∃ *a* ∈ *set as*. (*a*, *b*) ∈ *set rel*}
begin

abbreviation *rtrancl-i* ≡ *rtrancl-impl gen-succ un memb empty*
abbreviation *trancl-i* ≡ *trancl-impl gen-succ un memb empty*

lemma *rtrancl-impl*:
set-of (*rtrancl-i rel as*) = {*b*. (∃ *a* ∈ *set as*. (*a*, *b*) ∈ (*set rel*)*)}
proof −
interpret *set-access-succ set-of memb empty un gen-succ rel set rel*
by (*unfold-locales*, *insert gen-succ*, *auto*)
show *?thesis* **unfolding** *rtrancl-impl-def Let-def relpow-impl*
by (*auto simp*: *rtrancl-finite-relpow*)
qed

lemma *trancl-impl*:
set-of (*trancl-i rel as*) = {*b*. (∃ *a* ∈ *set as*. (*a*, *b*) ∈ (*set rel*)$^+$)}
proof −
interpret *set-access-succ set-of memb empty un gen-succ rel set rel*
by (*unfold-locales*, *insert gen-succ*, *auto*)
show *?thesis*
unfolding *trancl-impl-def Let-def relpow-impl trancl-unfold-left relcomp-unfold rtrancl-finite-relpow succ* **by** *auto*
qed

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* × '*a*) *list* ⇒ '*a list* ⇒ '*a list*
where
  rtrancl-list-impl = *rtrancl-impl*
    (λ *r as*. *remdups* (*map snd* (*filter* (λ (*a*, *b*). *a* ∈ *set as*) *r*)))
    (λ *xs ys*. (*filter* (λ *x*. *x* ∉ *set ys*) *xs*) @ *ys*)
    (λ *x xs*. *x* ∈ *set xs*)
    []

definition *trancl-list-impl* :: ('*a* × '*a*) *list* ⇒ '*a list* ⇒ '*a list*
where
  trancl-list-impl = *trancl-impl*
    (λ *r as*. *remdups* (*map snd* (*filter* (λ (*a*, *b*). *a* ∈ *set as*) *r*)))
    (λ *xs ys*. (*filter* (λ *x*. *x* ∉ *set ys*) *xs*) @ *ys*)
    (λ *x xs*. *x* ∈ *set xs*)
    []

lemma *rtrancl-list-impl*:
  set (*rtrancl-list-impl r as*) = {*b*. ∃ *a* ∈ *set as*. (*a*, *b*) ∈ (*set r*)*}
  unfolding *rtrancl-list-impl-def*
  by (*rule set-access-gen.rtrancl-impl*, *unfold-locales*, *force*+)

lemma *trancl-list-impl*:
  set (*trancl-list-impl r as*) = {*b*. ∃ *a* ∈ *set as*. (*a*, *b*) ∈ (*set r*)$^+$}
  unfolding *trancl-list-impl-def*
  by (*rule set-access-gen.trancl-impl*, *unfold-locales*, *force*+)

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 × 'a) list ⇒ ('a ⇒ 'a list)
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 −
  let ?rm = map (λ a. (a, rtrancl-list-impl r [a])) ((remdups ∘ map fst) r)
  show ?thesis
  proof (cases map-of ?rm a)
    case None
    have one: ?l = {a}
      unfolding memo-list-rtrancl-def Let-def None
      by auto
    from None [unfolded map-of-eq-None-iff]
    have a: a ∉ fst ‘ set r by force
    {
      fix b
      assume b ∈ ?r
      from this [unfolded rtrancl-power relpow-fun-conv] obtain n f where
        ab: f 0 = a ∧ f n = b and steps: ⋀ i. i < n ⟹ (f i, f (Suc i)) ∈ set r by
auto
      from ab steps [of 0] a have a = b
        by (cases n, force+)
    }
    then have ?r = {a} by auto
    then show ?thesis unfolding one by simp
  next
    case (Some as)
    have as: set as = {b. (a, b) ∈ (set r) ^*}
      using map-of-SomeD [OF Some]
        rtrancl-list-impl [of r [a]] by force
    then show ?thesis unfolding memo-list-rtrancl-def Let-def Some by simp
  qed
qed

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 −
  let ?rm = map (λ a. (a, trancl-list-impl r [a])) ((remdups ∘ map fst) r)
  show ?thesis
  proof (cases map-of ?rm a)
    case None
    have one: ?l = {}
      unfolding memo-list-trancl-def Let-def None
      by auto
    from None [unfolded map-of-eq-None-iff]
      have a: a ∉ fst ‘ set r by force
    {
      fix b
      assume b ∈ ?r
      from this [unfolded trancl-unfold-left] a have False by force
    }
    then have ?r = {} by auto
    then show ?thesis unfolding one by simp
  next
    case (Some as)
    have as: set as = {b. (a, b) ∈ (set r)⁺}
      using map-of-SomeD [OF Some]
        trancl-list-impl[of r [a]] by force
    then show ?thesis unfolding memo-list-trancl-def Let-def Some by simp
  qed
qed

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 $'v \Rightarrow 'k$, *elem-list-to-rm* will generate a map of type $'k \Rightarrow 'v$ *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* ⇒ *'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 −
  let *?abort* = λ *maybe*. *case maybe of None* ⇒ *True* | *Some* - ⇒ *False*
  let *?I* = λ *aas maybe*. *maybe* = *None* ⟷ (∀ *a*. *a* ∈ *rs.α as* − *aas* ⟶ *a* ∈ *rs.α bs*)
  let *?it* = *rs-subset as bs*
  have *?I* {} *?it* ∨ (∃ *it* ⊆ *rs.α as*. *it* ≠ {} ∧ ¬ *?abort ?it* ∧ *?I it ?it*)
    unfolding *rs-subset-def*
    by (*rule rs.iteratei-rule-P* [**where** *I*=*?I*]) (*auto simp*: *rs.correct*)
  then show *?thesis* **by** *auto*
qed

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*
  unfolding *rs-subset-list-def*
    sorted-list-subset[*OF rs.to-sorted-list-correct*(*3*)[*OF rs.invar*, *of as*]
    rs.to-sorted-list-correct(*3*)[*OF rs.invar*, *of bs*]]

```
  by (simp add: rs.to-sorted-list-correct)

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 −
  {
    fix start
    have rs.α (foldl rs.union start qs) = rs.α start ∪ ⋃ (rs.α ‘ set qs)
      by (induct qs arbitrary: start, auto simp: rs.correct)
  } from this[of rs.empty ()]
  show ?thesis unfolding rs-Union-def
    by (auto simp: rs.correct)
qed
```

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 ⇒ '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
⇒ rules))

lemma rm-to-list-empty [simp]:
  rm.to-list (rm.empty ()) = []
proof −
  have map-of (rm.to-list (rm.empty ())) = Map.empty
    by (simp add: rm.correct)
  moreover have map-of-empty-iff: ⋀l. map-of l = Map.empty ⟷ l = []
    by (case-tac l) auto
  ultimately show ?thesis by metis
qed

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 −
  {
    fix d
    assume d: d ∈ data
    then have d: d ∈ {d' ∈ data. key d' = key d} by auto
    have d ∈ ?R
     by (rule UnionI[OF - d], rule CollectI, rule exI[of - key d], unfold rm-set-lookup[of
key d], simp)
  }
  moreover
  {
    fix d
    assume d ∈ ?R
    from this[unfolded rm-set-lookup]
    have d ∈ data by auto
  }
  ultimately show ?thesis by blast
qed

lemma finite-data:
  finite data
  unfolding data-lookup
proof
  show finite {set (rm-set-lookup rm k) | k. True} (is finite ?L)
  proof −
    let ?rmset = rm.α rm
    let ?M = ?rmset ‘ Map.dom ?rmset
    let ?N = ((λ e. set (case e of None ⇒ [] | Some ds ⇒ ds)) ‘ ?M)
    let ?K = ?N ∪ {{}}
    from rm.finite[of rm] have fin: finite ?K by auto
    show ?thesis
    proof (rule finite-subset[OF - fin], rule)
      fix ds
      assume ds ∈ ?L
      from this[unfolded rm-set-lookup-def]
      obtain fn where ds: ds = set (case rm.α rm fn of None ⇒ []
          | Some ds ⇒ ds) by auto
      show ds ∈ ?K
      proof (cases rm.α rm fn)
        case None
        then show ?thesis unfolding ds by auto
```

```
      next
        case (Some rules)
        from Some have fn: fn ∈ Map.dom ?rmset by auto
        have ds ∈ ?N
          unfolding ds
          by (rule, rule refl, rule, rule refl, rule fn)
        then show ?thesis by auto
      qed
    qed
  qed
qed (force simp: rm-set-lookup-def)

end

interpretation elem-list-to-rm: rm-set elem-list-to-rm key ds key set ds
proof
  fix k
  show set (rm-set-lookup (elem-list-to-rm key ds) k) = {d ∈ set ds. key d = k}
  proof (induct ds arbitrary: k)
    case Nil
    then show ?case unfolding rm-set-lookup-def
      by (simp add: rm.correct)
  next
    case (Cons d ds k)
    let ?el = elem-list-to-rm key
    let ?l = λk ds. set (rm-set-lookup (?el ds) k)
    let ?r = λk ds. {d ∈ set ds. key d = k}
    from Cons have ind:
      ⋀ k. ?l k ds = ?r k ds by auto
    show ?l k (d # ds) = ?r k (d # ds)
    proof (cases rm.α (?el ds) (key d))
      case None
      from None ind[of key d] have r: {da ∈ set ds. key da = key d} = {}
        unfolding rm-set-lookup-def by auto
      from None have el: ?el (d # ds) = rm.update-dj (key d) [d] (?el ds)
        by simp
      from None have ndom: key d ∉ Map.dom (rm.α (?el ds)) by auto
      have r: ?r k (d # ds) = ?r k ds ∩ {da. key da ≠ key d} ∪ {da . key da = k
∧ da = d} (is - = ?r1 ∪ ?r2) using r by auto
      from ndom have l: ?l k (d # ds) =
        set (case (rm.α (elem-list-to-rm key ds)(key d ↦ [d])) k of None ⇒ []
        | Some rules ⇒ rules) (is - = ?l) unfolding el rm-set-lookup-def
        by (simp add: rm.correct)
      {
        fix da
        assume da ∈ ?r1 ∪ ?r2
        then have da ∈ ?l
        proof
          assume da ∈ ?r2
```

then have *da*: *da* = *d* **and** *k*: *key d* = *k* **by** *auto*
show *?thesis* **unfolding** *da k* **by** *auto*
next
assume *da* ∈ *?r1*
from *this*[*unfolded ind*[*symmetric*] *rm-set-lookup-def*]
obtain *das* **where** *rm*: *rm*.α (*?el ds*) *k* = *Some das* **and** *da*: *da* ∈ *set das* **and** *k*: *key da* ≠ *key d* **by** (*cases rm*.α (*?el ds*) *k*, *auto*)
from *ind*[*of k*, *unfolded rm-set-lookup-def*] *rm da k* **have** *k*: *key d* ≠ *k* **by** *auto*
have *rm*: (*rm*.α (*elem-list-to-rm key ds*)(*key d* ↦ [*d*])) *k* = *Some das*
unfolding *rm*[*symmetric*] **using** *k* **by** *auto*
show *?thesis* **unfolding** *rm* **using** *da* **by** *auto*
qed
}
moreover
{
fix *da*
assume *l*: *da* ∈ *?l*
let *?rm* = ((*rm*.α (*elem-list-to-rm key ds*))(*key d* ↦ [*d*])) *k*
from *l* **obtain** *das* **where** *rm*: *?rm* = *Some das* **and** *da*: *da* ∈ *set das*
by (*cases ?rm*, *auto*)
have *da* ∈ *?r1* ∪ *?r2*
proof (*cases k* = *key d*)
case *True*
with *rm da* **have** *da*: *da* = *d* **by** *auto*
then show *?thesis* **using** *True* **by** *auto*
next
case *False*
with *rm* **have** *rm*.α (*?el ds*) *k* = *Some das* **by** *auto*
from *ind*[*of k*, *unfolded rm-set-lookup-def this*] *da False*
show *?thesis* **by** *auto*
qed
}
ultimately have *?l* = *?r1* ∪ *?r2* **by** *blast*
then show *?thesis* **unfolding** *l r* **.**
next
case (*Some das*)
from *Some ind*[*of key d*] **have** *das*: {*da* ∈ *set ds*. *key da* = *key d*} = *set das*
unfolding *rm-set-lookup-def* **by** *auto*
from *Some* **have** *el*: *?el* (*d* # *ds*) = *rm.update* (*key d*) (*d* # *das*) (*?el ds*)
by *simp*
from *Some* **have** *dom*: *key d* ∈ *Map.dom* (*rm*.α (*?el ds*)) **by** *auto*
from *dom* **have** *l*: *?l k* (*d* # *ds*) =
set (*case* (*rm*.α (*elem-list-to-rm key ds*)(*key d* ↦ (*d* # *das*))) *k of None* ⇒ []
| *Some rules* ⇒ *rules*) (**is** - = *?l*) **unfolding** *el rm-set-lookup-def*
by (*simp add*: *rm.correct*)
have *r*: *?r k* (*d* # *ds*) = *?r k ds* ∪ {*da*. *key da* = *k* ∧ *da* = *d*} (**is** - = *?r1* ∪ *?r2*) **by** *auto*

{
  fix *da*
  assume $da \in ?r1 \cup ?r2$
  then have $da \in ?l$
  proof
    assume $da \in ?r2$
    then have *da*: $da = d$ **and** *k*: $key\ d = k$ **by** *auto*
    show *?thesis* **unfolding** *da k* **by** *auto*
  next
    assume $da \in ?r1$
    from *this*[*unfolded ind*[*symmetric*] *rm-set-lookup-def*]
    obtain *das'* **where** *rm*: $rm.\alpha\ (?el\ ds)\ k = Some\ das'$ **and** *da*: $da \in set\ das'$ **by** (*cases* $rm.\alpha\ (?el\ ds)\ k$, *auto*)
    from *ind*[*of k*, *unfolded rm-set-lookup-def rm*] **have** *das'*: $set\ das' = \{d \in set\ ds.\ key\ d = k\}$ **by** *auto*
    show *?thesis*
    proof (*cases* $k = key\ d$)
      case *True*
      show *?thesis* **using** *das' das da* **unfolding** *True* **by** *simp*
    next
      case *False*
      then show *?thesis* **using** *das' da rm* **by** *auto*
    qed
  qed
}
moreover
{
  fix *da*
  assume *l*: $da \in ?l$
  let $?rm = ((rm.\alpha\ (elem\text{-}list\text{-}to\text{-}rm\ key\ ds))(key\ d \mapsto d\ \#\ das))\ k$
  from *l* **obtain** *das'* **where** *rm*: $?rm = Some\ das'$ **and** *da*: $da \in set\ das'$
    by (*cases ?rm*, *auto*)
  have $da \in ?r1 \cup ?r2$
  proof (*cases* $k = key\ d$)
    case *True*
    with *rm da das* **have** *da*: $da \in set\ (d\ \#\ das)$ **by** *auto*
    then have $da = d \vee da \in set\ das$ **by** *auto*
    then have *k*: $key\ da = k$
    proof
      assume $da = d$
      then show *?thesis* **using** *True* **by** *simp*
    next
      assume $da \in set\ das$
      with *das True* **show** *?thesis* **by** *auto*
    qed
    from *da k* **show** *?thesis* **using** *das* **by** *auto*
  next
    case *False*
    with *rm* **have** $rm.\alpha\ (?el\ ds)\ k = Some\ das'$ **by** *auto*

```
      from ind[of k, unfolded rm-set-lookup-def this] da False
      show ?thesis by auto
     qed
    }
    ultimately have ?l = ?r1 ∪ ?r2 by blast
    then show ?thesis unfolding l r .
   qed
  qed
qed

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 ()
  by (unfold-locales, auto simp: rs.correct)

abbreviation rm-succ :: ('a :: linorder × 'a) list ⇒ 'a list ⇒ 'a list
where
  rm-succ ≡ (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 × 'a) list ⇒ 'a list ⇒ '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* × *'a*) *list* ⇒ *'a list* ⇒ *'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*. ∃ *a* ∈ *set as*. (*a*,*b*) ∈ (*set r*)*}
unfolding *rtrancl-rbt-impl-def*
by (*rule set-access-gen.rtrancl-impl*, *unfold-locales*, *unfold Let-def*, *simp add*: *rs.correct elem-list-to-rm.rm-set-lookup*, *force*)

lemma *trancl-rbt-impl*:
rs.α (*trancl-rbt-impl r as*) = {*b*. ∃ *a* ∈ *set as*. (*a*,*b*) ∈ (*set r*)$^+$}
unfolding *trancl-rbt-impl-def*
by (*rule set-access-gen.trancl-impl*, *unfold-locales*, *unfold Let-def*, *simp add*: *rs.correct elem-list-to-rm.rm-set-lookup*, *force*)

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* × *'a*) *list* ⇒ (*'a* ⇒ *'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* ∘ *rs.from-list* ∘ *map fst*) *r*))
in
(λ*a*. *case rm.lookup a rm of*
None ⇒ *rs.from-list* [*a*]
| *Some as* ⇒ *as*))

lemma *memo-rbt-rtrancl*:
rs.α (*memo-rbt-rtrancl r a*) = {*b*. (*a*, *b*) ∈ (*set r*)*} (**is** *?l* = *?r*)
proof −
let *?rm* = *rm.to-map*
(*map* (λ*a*. (*a*, *rtrancl-rbt-impl r* [*a*])) ((*rs.to-list* ∘ *rs.from-list* ∘ *map fst*) *r*))
show *?thesis*
proof (*cases rm.lookup a ?rm*)
case *None*
have *one*: *?l* = {*a*}
unfolding *memo-rbt-rtrancl-def Let-def None*
by (*simp add*: *rs.correct*)

```
    from None [unfolded rm.lookup-correct [OF rm.invar], simplified rm.correct
map-of-eq-None-iff]
    have a: a ∉ fst ‘ set r by (simp add: rs.correct, force)
    {
      fix b
      assume b ∈ ?r
      from this [unfolded rtrancl-power relpow-fun-conv] obtain n f where
        ab: f 0 = a ∧ f n = b and steps: ⋀ i. i < n ⟹ (f i, f (Suc i)) ∈ set r by
auto
      from ab steps [of 0] a have b = a
        by (cases n, force+)
    }
    then have ?r = {a} by auto
    then show ?thesis unfolding one by simp
  next
    case (Some as)
    have as: rs.α as = {b. (a,b) ∈ (set r)*}
      using map-of-SomeD [OF Some [unfolded rm.lookup-correct [OF rm.invar],
simplified rm.correct]]
        rtrancl-rbt-impl [of r [a]] by force
    then show ?thesis unfolding memo-rbt-rtrancl-def Let-def Some by simp
  qed
qed

definition memo-rbt-trancl :: ('a :: linorder × 'a) list ⇒ ('a ⇒ '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 ∘ rs.from-list ∘ map fst)
r))
    in (λ a.
      (case rm.lookup a rm of
        None ⇒ rs.empty ()
      | Some as ⇒ as)))

lemma memo-rbt-trancl:
  rs.α (memo-rbt-trancl r a) = {b. (a, b) ∈ (set r)+} (is ?l = ?r)
proof –
  let ?rm = rm.to-map
    (map (λ a. (a, trancl-rbt-impl r [a])) ((rs.to-list ∘ rs.from-list ∘ map fst) r))
  show ?thesis
  proof (cases rm.lookup a ?rm)
    case None
    have one: ?l = {}
      unfolding memo-rbt-trancl-def Let-def None
      by (simp add: rs.correct)
    from None [unfolded rm.lookup-correct [OF rm.invar], simplified rm.correct
map-of-eq-None-iff]
```

have *a*: $a \notin$ *fst* ‘ *set r* **by** (*simp add*: *rs.correct*, *force*)
{
fix *b*
assume $b \in$ *?r*
from *this* [*unfolded trancl-unfold-left*] *a* **have** *False* **by** *force*
}
then have *?r* = {} **by** *auto*
then show *?thesis* **unfolding** *one* **by** *simp*
next
case (*Some as*)
have *as*: *rs.α as* = {*b*. (*a*,*b*) ∈ (*set r*)$^+$}
using *map-of-SomeD* [*OF Some* [*unfolded rm.lookup-correct* [*OF rm.invar*], *simplified rm.correct*]]
trancl-rbt-impl [*of r* [*a*]] **by** *force*
then show *?thesis* **unfolding** *memo-rbt-trancl-def Let-def Some* **by** *simp*
qed
qed

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-list-impl r′ x′*)
using *assms* **by** (*auto simp*: *rtrancl-list-impl*)

lemma *trancl-Image-eq*:
assumes *r* = *set r′* **and** *x* = *set x′*
shows r^+ “ *x* = *set* (*trancl-list-impl r′ x′*)
using *assms* **by** (*auto simp*: *trancl-list-impl*)

5.1 A Simproc for Computing the Images of Finite Transitive Closures

ML ‹

```
signature FINITE-TRANCL-IMAGE =
sig
  val trancl-simproc : Proof.context −> cterm −> thm option
  val rtrancl-simproc : Proof.context −> cterm −> thm option
end

structure Finite-Trancl-Image : FINITE-TRANCL-IMAGE =
struct
```

```
fun eval-tac ctxt =
  let val conv = Code-Runtime.dynamic-holds-conv ctxt
  in CONVERSION (Conv.params-conv ~1 (K (Conv.concl-conv ~1 conv)) ctxt)
THEN' resolve-tac ctxt [TrueI] end

fun mk-rtrancl T = Const (@{const-name rtrancl-list-impl}, T);

fun mk-trancl T = Const (@{const-name trancl-list-impl}, T);

fun dest-rtrancl-Image
      (Const (@{const-name Image}, T) $ (Const (@{const-name rtrancl}, -) $ r)
$ x) = (T, r, x)
  | dest-rtrancl-Image - = raise Match

fun dest-trancl-Image
      (Const (@{const-name Image}, T) $ (Const (@{const-name trancl}, -) $ r)
$ x) = (T, r, x)
  | dest-trancl-Image - = raise Match

fun gen-simproc dest mk-const eq-thm ctxt ct =
  let
    val t = Thm.term-of ct;
    val (T, r, x) = t |> dest;
  in
    (*make sure that the relation as well as the given domain are finite sets*)
    (case (try HOLogic.dest-set r, try HOLogic.dest-set x) of
      (SOME xs, SOME ys) =>
        let
          (*types*)
          val setT = T |> dest-funT |> snd |> dest-funT |> fst;
          val eltT = setT |> HOLogic.dest-setT;
          val prodT = HOLogic.mk-prodT (eltT, eltT);
          val prod-listT = HOLogic.listT prodT;
          val listT = HOLogic.listT eltT;

          (*terms*)
          val set = Const (@{const-name List.set}, listT --> setT);
          val const = mk-const (prod-listT --> listT --> listT);
          val r' = HOLogic.mk-list prodT xs;
          val x' = HOLogic.mk-list eltT ys;
          val t' = set $ (const $ r' $ x')
          val u = Value-Command.value ctxt t';
          val eval = (t', u) |> HOLogic.mk-eq |> HOLogic.mk-Trueprop;

          val maybe-rule =
            try (Goal.prove ctxt [] [] eval) (fn {context, ...} => eval-tac context 1);
        in
          (case maybe-rule of
            SOME rule =>
```

```
      let
        val conv = (t, t') |> HOLogic.mk-eq |> HOLogic.mk-Trueprop;
        val eq-thm' = Goal.prove ctxt [] [] conv (fn {context = ctxt', ...} =>
          resolve-tac ctxt' [eq-thm] 1 THEN REPEAT (simp-tac ctxt' 1));
      in
      SOME (@{thm HOL.trans} OF [eq-thm', rule] RS @{thm eq-reflection})
      end
    | NONE => NONE)
   end
 | - => NONE)
end


val rtrancl-simproc = gen-simproc dest-rtrancl-Image mk-rtrancl @{thm rtrancl-Image-eq}
val trancl-simproc = gen-simproc dest-trancl-Image mk-trancl @{thm trancl-Image-eq}

end
›

simproc-setup rtrancl-Image (r* `` x) = ‹K Finite-Trancl-Image.rtrancl-simproc›
simproc-setup trancl-Image  (r+ `` x) = ‹K Finite-Trancl-Image.trancl-simproc›
```

5.2 Example

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

lemma
 $\{(1::nat,\ 2),\ (2,\ 3),\ (3,\ 4),\ (4,\ 5)\}^*$ `` $\{1\} = \{1,\ 2,\ 3,\ 4,\ 5\}$
 $\{(1::nat,\ 2),\ (2,\ 3),\ (3,\ 4),\ (4,\ 5)\}^+$ `` $\{1\} = \{2,\ 3,\ 4,\ 5\}$
 apply *simp-all*
 apply *auto*
done

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

lemma
 $\{(x,\ y)\}^*$ `` $\{x\} = \{x,\ y\}$
 oops

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.