Transitive Closure Implementation with Meta data

Nils Cremer

September 30, 2022

Abstract

In this work we provide algorithms to calculate the transitive closure of a relation. In addition to the transitive closure the algirhtms also provides meta data for each element such as corresponsing proof term. We prove correctness of the algorithms in the sense that the transitive closure is produced and that the meta data are valid with respect to a user specified predicate.

Contents

1 Relpow Implementation

theory *Relpow-Meta-Impl*
  imports *Transitive−Closure.Transitive-Closure-Impl*
begin

fun
  $\mathit{relpow\text{-}meta\text{-}impl} :: (('a * 'b)\ \mathit{list} \Rightarrow ('a * 'b)\ \mathit{list}\ ) \Rightarrow$
$(('a * 'b)\ \mathit{list} \Rightarrow 'c \Rightarrow 'c) \Rightarrow ('a \Rightarrow 'c \Rightarrow \mathit{bool}) \Rightarrow ('a * 'b)\ \mathit{list} \Rightarrow 'c \Rightarrow \mathit{nat} \Rightarrow 'c$
where
  $\mathit{relpow\text{-}meta\text{-}impl}\ \mathit{succ}\ \mathit{un}\ \mathit{memb}\ \mathit{new}\ \mathit{have}\ 0 = \mathit{un}\ \mathit{new}\ \mathit{have}\ |$
  $\mathit{relpow\text{-}meta\text{-}impl}\ \mathit{succ}\ \mathit{un}\ \mathit{memb}\ \mathit{new}\ \mathit{have}\ (\mathit{Suc}\ m) =$
    $(\mathit{if}\ \mathit{new} = []\ \mathit{then}\ \mathit{have}$
    else
      let
        $\mathit{maybe} = \mathit{succ}\ \mathit{new};$
        $\mathit{have}' = \mathit{un}\ \mathit{new}\ \mathit{have};$
        $\mathit{new}' = \mathit{filter}\ (\lambda x.\ \neg\ \mathit{memb}\ (\mathit{fst}\ x)\ \mathit{have}')\ \mathit{maybe}$

in relpow-meta-impl succ un memb new′ have′ m)

abbreviation *keys* ≡ *map fst*

definition *valid-metas-set valid-meta as* ≡ (∀ (*k*, *v*) ∈ *as*. *valid-meta k v*)

locale *relpow-data-structure* =
  fixes *un* :: (′*a* ∗ ′*b*) *list* ⇒ ′*c* ⇒ ′*c*
    and *set-of* :: ′*c* ⇒ (′*a* ∗ ′*b*) *set*
    and *memb* :: ′*a* ⇒ ′*c* ⇒ *bool*
    and *empty* :: ′*c*
    and *valid-meta* :: ′*a* ⇒ ′*b* ⇒ *bool*
  assumes *un*: *fst* ‘ *set-of* (*un as m*) = *fst* ‘ (*set as* ∪ *set-of m*)
    and *memb*: *memb a m* ⟷ (*a* ∈ *fst* ‘ (*set-of m*))
    and *empty*: *set-of empty* = {}
    and *un-valid-metas*: *valid-metas-set valid-meta* (*set as*) ⟹ *valid-metas-set valid-meta* (*set-of m*)
      ⟹ *valid-metas-set valid-meta* (*set-of* (*un as m*))
begin

abbreviation *valid-metas* ≡ *valid-metas-set valid-meta*

abbreviation *keys-of m* ≡ *fst* ‘ *set-of m*

lemma *valid-metas-empty*: *valid-metas* (*set-of empty*)
  using *valid-metas-set-def empty* **by** *auto*

end

locale *relpow-meta-succ* =
  relpow-data-structure un set-of memb empty valid-meta
  for *un* :: (′*a* ∗ ′*b*) *list* ⇒ ′*c* ⇒ ′*c* **and** *set-of memb empty valid-meta* +
  fixes *succ* :: (′*a* ∗ ′*b*) *list* ⇒ (′*a* ∗ ′*b*) *list*
    and *succ-rel* :: (′*a* ∗ ′*a*) *set*
  assumes *succ-rel*: *fst* ‘ (*set* (*succ as*)) = {*b*. ∃ *a* ∈ *fst* ‘ (*set as*). (*a*, *b*) ∈ *succ-rel*}
    and *succ-valid-metas*: *valid-metas-set valid-meta* (*set as*) ⟹ *valid-metas-set valid-meta* (*set* (*succ as*))
begin

abbreviation *add-undef-meta* ≡ *map* (λ*a*. (*a*,*undefined*))

definition *succ′ as* ≡ *keys* (*succ* (*add-undef-meta as*))

definition *un′ as* ≡ *un* (*add-undef-meta as*)

sublocale *relpow′*: *set-access-succ keys-of memb empty un′ succ′ succ-rel*
  by (*unfold-locales*,
    auto simp: *un memb empty succ-rel succ′-def un′-def*,

force)

definition *eq-new new new'* ≡ *set* (*keys new*) = *set new'*

lemma *succ-eq*: *eq-new new new'* ⟹ *eq-new* (*succ new*) (*succ' new'*)
  using *succ-rel* **by** (*auto simp add*: *eq-new-def succ'-def*) *fastforce*

lemma *un-eq*: *eq-new as as'* ⟹ *keys-of bs* = *keys-of bs'*
  ⟹ *keys-of* (*un as bs*) = *keys-of* (*un' as' bs'*)
  using *relpow'.un* **by** (*auto simp add*: *eq-new-def un'-def un*)

lemma *relpow-relpow'-aux*: *eq-new new new'* ⟹ *keys-of have* = *keys-of have'*
  ⟹ *keys-of* (*relpow-meta-impl succ un memb new have n*)
    = *keys-of* (*relpow-impl succ' un' memb new' have' n*)
proof (*induction n arbitrary*: *have have' new new'*)
  case *0*
  then show *?case* **by** (*simp add*: *un-eq*)
next
  case (*Suc n*)
  then show *?case*
  proof (*cases new* = [])
    case *True*
    then have *new'* = [] **using** *Suc.prems*(*1*) **by** (*auto simp*: *eq-new-def*)
    then show *?thesis* **using** *True Suc.prems*(*2*) **by** *simp*
  next
    case *False*
    then have *new'*: *new'* ≠ [] **using** *Suc.prems*(*1*) **by** (*auto simp*: *eq-new-def*)
    let *?maybe1* = *succ new*
    let *?maybe2* = *succ' new'*
    have *maybe-eq*: *eq-new ?maybe1 ?maybe2* **using** *Suc.prems*(*1*) **by** (*simp add*: *succ-eq*)
    let *?have1* = *un new have*
    let *?have2* = *un' new' have'*
    have *have'-eq*: *keys-of ?have1* = *keys-of ?have2* **using** *Suc.prems* **by** (*auto simp*: *un-eq*)
    let *?new1* = *filter* (*λx*. ¬ *memb* (*fst x*) *?have1*) *?maybe1*
    let *?new2* = *filter* (*λx*. ¬ *memb x ?have2*) *?maybe2*
    have *new'-eq*: *eq-new ?new1 ?new2* **using** *maybe-eq have'-eq eq-new-def relpow'.memb* **by** (*auto simp*: *eq-new-def*)
    show *?thesis* **using** *Suc.IH*[*OF new'-eq have'-eq*] *False new'* **by** (*simp add*: *Let-def*)
  qed
qed

lemma *relpow-relpow'*: *keys-of* (*relpow-meta-impl succ un memb new have n*)
    = *keys-of* (*relpow-impl succ' un' memb* (*keys new*) *have n*)
  by (*simp add*: *eq-new-def relpow-relpow'-aux*)

theorem *trancl-meta-impl*: *keys-of* (*relpow-meta-impl succ un memb new empty*

n)
  $= \{b \mid a\ b\ m.\ a \in \mathit{fst}\ `\ \mathit{set\ new} \land m \le n \land (a,\ b) \in \mathit{succ\text{-}rel}\ \widehat{}\widehat{}\ m\}$
  by (*simp add*: *relpow-relpow′ relpow′.relpow-impl*)

lemma *relpow-valid-metas-gen*: *valid-metas* (*set-of have*) $\Longrightarrow$ *valid-metas* (*set new*)
  $\Longrightarrow$ *valid-metas* (*set-of* (*relpow-meta-impl succ un memb new have n*))
proof (*induction n arbitrary*: *new have*)
  case *0*
  then show *?case* **by** (*simp add*: *un-valid-metas*)
next
  case (*Suc n*)
  let *?have′* = *un new have*
  have *valid-have′*: *valid-metas* (*set-of ?have′*) **by** (*simp add*: *un-valid-metas Suc.prems*)
  let *?new′* = *filter* ($\lambda x.$ $\neg$ *memb* (*fst x*) *?have′*) (*succ new*)
  have *valid-metas* (*set* (*succ new*)) **using** *succ-valid-metas Suc.prems* **by** *auto*
  hence *valid-new′*: *valid-metas* (*set ?new′*) **using** *filter-is-subset* **by** (*auto simp*: *valid-metas-set-def*)
  then show *?case* **using** *Suc.IH Suc valid-have′ valid-new′* **by** (*simp add*: *Let-def*)
qed

theorem *relpow-valid-metas*: *valid-metas* (*set new*) $\Longrightarrow$ *valid-metas* (*set-of* (*relpow-meta-impl succ un memb new empty n*))
  using *relpow-valid-metas-gen valid-metas-empty* **by** *blast*

end

end

2 Transitive Closure Implementation

theory *Transitive-Closure-Meta-Impl*
  imports *HOL−Library.Mapping HOL−Library.Product-Lexorder Relpow-Meta-Impl*
begin

2.1 Data structure

definition *un-map* $\equiv$ *fold* ($\lambda(u,b).$ *Mapping.update u b*)

definition *memb-map u m* $\equiv$ $\neg$ *Option.is-none* (*Mapping.lookup m u*)

lemma *un-keys*: *Mapping.keys* (*un-map as m*) = *fst* ‘ *set as* $\cup$ *Mapping.keys m*
  by (*induction as arbitrary*: *m*) (*auto simp*: *un-map-def*)

lemma *update-valid-metas*: *valid-meta k v* $\Longrightarrow$ *valid-metas-set valid-meta* (*Mapping.entries m*) $\Longrightarrow$
  valid-metas-set valid-meta (*Mapping.entries* (*Mapping.update k v m*))
  unfolding *valid-metas-set-def* **by** (*simp add*: *entries-delete entries-update*)

lemma *un-map-valid-metas*:
assumes *valid-metas-set valid-meta* (*set as*)
and *valid-metas-set valid-meta* (*Mapping.entries m*)
shows *valid-metas-set valid-meta* (*Mapping.entries* (*un-map as m*))
using *assms*
unfolding *un-map-def*
proof (*induction as arbitrary*: *m*)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons a as*)
then have *as-valid*: *valid-metas-set valid-meta* (*set as*) **unfolding** *valid-metas-set-def* **by** *simp*
obtain *k v* **where** *kv*: $a = (k,v)$ **by** *fastforce*
then have *valid-meta k v* **using** *Cons*(*2*) **unfolding** *valid-metas-set-def* **by** *simp*
then have *valid-metas-set valid-meta* (*Mapping.entries* (*Mapping.update k v m*)) **using** *update-valid-metas Cons*(*3*) **by** *fast*
then show *?case* **using** *kv as-valid Cons update-valid-metas* **by** *simp*
qed

interpretation *trancl-data-structure-mapping*: *relpow-data-structure un-map Mapping.entries memb-map Mapping.empty*
unfolding *memb-map-def*
by (*unfold-locales*, *auto simp*: *image-Un un-keys keys-is-none-rep un-map-valid-metas*)

2.2 Transitive Closure with successor function

locale *trancl-meta-succ* =
relpow-data-structure un set-of memb empty valid-meta
for *un* :: $(('a * 'a) * 'b)$ *list* $\Rightarrow$ $'c \Rightarrow 'c$
and *set-of memb empty valid-meta* +
fixes *succ* :: $(('a * 'a) * 'b)$ *list* $\Rightarrow$ $(('a * 'a) * 'b)$ *list*
and *rel-meta* :: $(('a * 'a) * 'b)$ *list*
assumes *succ-rel-step*: *fst* ‘ *set* (*succ as*) = $\{(x,z) \mid x\ y\ z.\ (x,y) \in$ *fst* ‘ *set as* $\wedge$ $(y,z) \in$ *fst* ‘ *set rel-meta*$\}$
and *succ-valid-metas*: *valid-metas* (*set as*) $\Longrightarrow$ *valid-metas* (*set* (*succ as*))
begin

definition *rel* $\equiv$ *fst* ‘ *set rel-meta*

definition *succ-rel* $\equiv$ $\{((x,y),(x,z)) \mid x\ y\ z.\ (y,z) \in rel\}$

lemma *succ-succ-rel*: *fst* ‘ *set* (*succ as*) = $\{b.\ \exists\, a \in$ *fst* ‘ *set as*. $(a,\ b) \in$ *succ-rel*$\}$ (**is** *?l* = *?r*)
proof
show *?l* $\subseteq$ *?r* **unfolding** *succ-rel-def rel-def succ-rel-step* **by** *blast*
show *?r* $\subseteq$ *?l*

proof
  fix *b*
  assume $b \in ?r$
  then obtain *a* **where** *a*: $a \in fst$ ` *set as* **and** $(a,b) \in$ *succ-rel* **by** *blast*
  then obtain *x y z* **where** $a = (x,y) \wedge b = (x,z) \wedge (y,z) \in rel$ **unfolding** *succ-rel-def* **by** *auto*
  then show $b \in ?l$ **unfolding** *succ-rel-step rel-def* **using** *a* **by** *auto*
qed
qed

sublocale *relpow-meta-succ*: *relpow-meta-succ un set-of memb empty valid-meta succ succ-rel*
using *succ-succ-rel succ-valid-metas* **by** *unfold-locales*

lemma *rel-succ-comp-gen*: $\{b.\ \exists a.\ a \in as \wedge (a,b) \in succ\text{-}rel\ \widehat{\ }\widehat{\ }\ m\} = as\ O\ rel\ \widehat{\ }\widehat{\ }\ m$ (**is** $?l\ m = ?r\ m$)
proof
  show $?l\ m \subseteq ?r\ m$
  proof (*induction m*)
    case *0*
    then show *?case* **by** *auto*
  next
    case (*Suc m*)
    show *?case*
    proof
      fix *b*
      assume *b-in-l*: $b \in ?l\ (Suc\ m)$
      then obtain *a z*
        where $a \in as$
          and $(a,z) \in succ\text{-}rel\ \widehat{\ }\widehat{\ }\ m$
          and *zb*: $(z,b) \in$ *succ-rel* **by** *auto*
      then have $z \in as\ O\ rel\ \widehat{\ }\widehat{\ }\ m$ **using** *Suc* **by** *blast*
      then show $b \in ?r\ (Suc\ m)$ **using** *zb* **unfolding** *succ-rel-def* **by** *auto*
    qed
  qed
next
  show $?r\ m \subseteq ?l\ m$
  proof (*induction m*)
    case *0*
    then show *?case* **by** *auto*
  next
    case (*Suc m*)
    show *?case*
    proof
      fix *b*
      assume $b \in ?r\ (Suc\ m)$
      then obtain *z* **where** *y-in-r*: $z \in ?r\ m$ **and** $b \in \{z\}\ O\ rel$ **by** *auto*

      then have *zb-relpow-rel*: $(z, b) \in$ *succ-rel* **using** *succ-rel-def* **by** *blast*
      have $z \in$ *?l m* **using** *y-in-r Suc* **by** *auto*
      then show $b \in$ *?l (Suc m)* **using** *zb-relpow-rel* **by** *auto*
    qed
  qed
qed

lemma *rel-succ-comp*: $\{b.\ \exists a.\ a \in rel \wedge (a,b) \in succ\text{-}rel\ \hat{}\hat{}\ m\} = rel\ \hat{}\hat{}\ Suc\ m$
  unfolding *rel-succ-comp-gen* **using** *relpow-commute* **by** *simp*

lemma *ntrancl-Suc*: $ntrancl\ (Suc\ n)\ rel = ntrancl\ n\ rel \cup rel\ \hat{}\hat{}\ Suc\ (Suc\ n)$ (**is** *?l = ?r*)
proof−
  have $\{i.\ 0 < i \wedge i \leq Suc\ (Suc\ n)\} = \{i.\ 0 < i \wedge i \leq Suc\ n\} \cup \{Suc\ (Suc\ n)\}$ **by** *auto*
  then have $ntrancl\ (Suc\ n)\ rel = \bigcup\ ((\hat{}\hat{})\ rel\ `\ (\{i.\ 0 < i \wedge i \leq Suc\ n\} \cup \{Suc\ (Suc\ n)\}))$ **unfolding** *ntrancl-def* **by** *auto*
  also have $\ldots = ntrancl\ n\ rel \cup rel\ \hat{}\hat{}\ Suc\ (Suc\ n)$ **unfolding** *ntrancl-def* **by** *auto*
  finally show *?thesis* **.**
qed

lemma *relpow-impl-ntrancl*: $\{b.\ \exists a\ m.\ a \in rel \wedge m \leq n \wedge (a, b) \in succ\text{-}rel\ \hat{}\hat{}\ m\}$
    $= ntrancl\ n\ rel$ (**is** *?l n = ?r n*)
proof (*induction n*)
  case *0*
  then show *?case* **by** *auto*
next
  case (*Suc n*)
  have $?l\ (Suc\ n) = \{b.\ \exists a\ m.\ a \in rel \wedge (m \leq n \vee m = Suc\ n) \wedge (a, b) \in succ\text{-}rel\ \hat{}\hat{}\ m\}$ **by** (*simp add*: *le-Suc-eq*)
  then have $?l\ (Suc\ n) = ?l\ n \cup \{b.\ \exists a.\ a \in rel \wedge (a, b) \in succ\text{-}rel\ \hat{}\hat{}\ Suc\ n\}$ **by** *force*
  also have $\ldots = ?l\ n \cup rel\ \hat{}\hat{}\ Suc\ (Suc\ n)$ **using** *rel-succ-comp* **by** *blast*
  also have $\ldots = ntrancl\ n\ rel \cup rel\ \hat{}\hat{}\ Suc\ (Suc\ n)$ **using** *Suc* **by** *simp*
  also have $\ldots = ntrancl\ (Suc\ n)\ rel$ **using** *ntrancl-Suc* **by** *simp*
  finally show *?case* **.**
qed

lemma *ntrancl-mono*: $n \leq m \Longrightarrow ntrancl\ n\ rel \subseteq ntrancl\ m\ rel$ **unfolding** *ntrancl-def* **by** *force*

lemma *ntrancl-bounded*: $finite\ r \Longrightarrow ntrancl\ (card\ r - 1 + n)\ r = r^+$
  by (*induction n*, *simp add*: *finite-trancl-ntranl*, *fastforce*)

theorem *relpow-meta-trancl*: *fst ` set-of (relpow-meta-impl succ un memb rel-meta empty (length rel-meta)) = trancl rel*
proof −
  have *ntrancl (length rel-meta) rel = trancl rel*

proof
have *card-leq*: *card rel* $-$ *1* $\leq$ *length rel-meta*
by (*metis List.finite-set card-image-le card-length diff-le-self le-trans rel-def*)
have *finite rel* **unfolding** *rel-def* **by** *simp*
then have *trancl rel* = *ntrancl* (*card rel* $-$ *1*) *rel* **by** (*simp add*: *finite-trancl-ntranl*)
then show *trancl rel* $\subseteq$ *ntrancl* (*length rel-meta*) *rel* **using** *card-leq ntrancl-mono*
by *simp*
next
show *ntrancl* (*length rel-meta*) *rel* $\subseteq$ *trancl rel* **using** *ntrancl-bounded*
by (*metis List.finite-set le-add2 list.set-map ntrancl-mono rel-def*)
qed
then show *?thesis* **using** *relpow-meta-succ.trancl-meta-impl rel-def relpow-impl-ntrancl*
by *simp*
qed

end

fun *to-rel-map* :: (($'a * 'a$) $* 'b$) *list* $\Rightarrow$ ($'a$, ($'a * 'b$) *list*) *mapping* **where**
to-rel-map [] = *Mapping.empty*
| *to-rel-map* (((*x*,*y*),*b*)#*xs*) = (*let m* = *to-rel-map xs in*
Mapping.update x ((*y*,*b*) # *Mapping.lookup-default* [] *m x*) *m*)

fun *succ-map* :: ($'b \Rightarrow 'b \Rightarrow 'b$) $\Rightarrow$ ($'a$, ($'a * 'b$) *list*) *mapping* $\Rightarrow$ (($'a * 'a$) $* 'b$) *list* $\Rightarrow$ (($'a * 'a$) $* 'b$) *list* **where**
succ-map combine-meta rel-map [] = []
| *succ-map combine-meta rel-map* (((*x*,*y*),*b*)#*xs*) =
map (λ(*z*,b'). ((*x*,*z*), *combine-meta b* b')) (*Mapping.lookup-default* [] *rel-map y*)
@ *succ-map combine-meta rel-map xs*

locale *trancl-meta-map* =
relpow-data-structure un set-of memb empty valid-meta
for *un* :: (($'a$::*linorder* $\times$ $'a$) $\times$ $'b$) *list* $\Rightarrow$ $'c \Rightarrow 'c$
and *set-of memb empty valid-meta* +
fixes *combine-meta* :: $'b \Rightarrow 'b \Rightarrow 'b$
and *rel-meta* :: (($'a \times 'a$) $\times$ $'b$) *list*
assumes *combine-meta-valid*: *valid-meta* (*x*,*y*) *b1* $\Longrightarrow$ *valid-meta* (*y*,*z*) *b2* $\Longrightarrow$ *valid-meta* (*x*,*z*) (*combine-meta b1 b2*)
and *valid-rel-meta*: *valid-metas-set valid-meta* (*set rel-meta*)
begin

abbreviation *succ* $\equiv$ *succ-map combine-meta* (*to-rel-map rel-meta*)

lemma *rel-map-default-lookup*: *set* (*Mapping.lookup-default* [] (*to-rel-map rel*) *y*) = {(*z*,*b*). ((*y*,*z*),*b*) $\in$ *set rel*}
by (*induction rel*) (*auto simp*: *lookup-default-empty lookup-default-update'* *Let-def split*: *if-splits*)

lemma *set-succ*: *set* (*succ as*)

= {((*x*,*z*),*combine-meta p q*) | *x y z p q*. ((*x*,*y*),*p*) ∈ *set as* ∧ ((*y*,*z*),*q*) ∈ *set rel-meta*} (**is** *?l as* = *?r as*)
proof (*induction as*)
  case *Nil*
  then show *?case* **by** *simp*
next
  case (*Cons a as*)
  then obtain *x y p* **where** *a-xyb*: *a* = ((*x*,*y*),*p*) **by** (*metis surj-pair*)
  then have *?l* (*a#as*) = {((*x*,*z*),*combine-meta p q*) | *z q*. ((*y*,*z*),*q*) ∈ *set rel-meta*} ∪ *?l as* **using** *rel-map-default-lookup* **by** *fastforce*
  also have ... = *?r* (*a#as*) **using** *a-xyb Cons* **by** *fastforce*
  finally show *?case* **.**
qed

lemma *succ-rel-meta*: *fst* ‘ *set* (*succ as*) = {(*x*,*z*) | *x y z*. (*x*, *y*) ∈ *fst* ‘ *set as* ∧ (*y*, *z*) ∈ *fst* ‘ *set rel-meta*}
  unfolding *set-succ* **by** *force*

lemma *succ-valid-meta*:
  assumes *valid-as*: *valid-metas-set valid-meta* (*set as*)
    shows *valid-metas-set valid-meta* (*set* (*succ as*))
  unfolding *valid-metas-set-def*
proof
  fix *b*
  assume *b* ∈ *set* (*succ as*)
  then obtain *x y z p q*
    where *b* = ((*x*,*z*),*combine-meta p q*)
      and ((*x*,*y*),*p*) ∈ *set as*
      and ((*y*,*z*),*q*) ∈ *set rel-meta* **using** *set-succ* **by** *auto*
  then show *case b of* (*u*,*r*) ⇒ *valid-meta u r* **using** *combine-meta-valid valid-rel-meta valid-as* **by** (*auto simp*: *valid-metas-set-def split*: *prod.splits*)
qed

sublocale *trancl-meta-succ un set-of memb empty valid-meta succ rel-meta*
  by *unfold-locales* (*auto simp*: *succ-rel-meta succ-valid-meta*)

end

locale *trancl-meta-rbt* =
  fixes *combine-meta* :: ′*b* ⇒ ′*b* ⇒ ′*b*
    and *valid-meta* :: ((′*a*::*linorder*) × ′*a*) ⇒ ′*b* ⇒ *bool*
    and *rel-meta* :: ((′*a* × ′*a*) × ′*b*) *list*
  assumes *combine-meta-valid*: *valid-meta* (*x*,*y*) *b1* ⟹ *valid-meta* (*y*,*z*) *b2* ⟹ *valid-meta* (*x*,*z*) (*combine-meta b1 b2*)
      and *valid-rel-meta*: *valid-metas-set valid-meta* (*set rel-meta*)
begin

sublocale *trancl-meta-map un-map Mapping.entries memb-map Mapping.empty valid-meta combine-meta rel-meta*

by *unfold-locales* (*auto simp*: *combine-meta-valid valid-rel-meta*)

abbreviation *trancl-meta-impl* ≡ (*relpow-meta-impl succ un-map memb-map rel-meta Mapping.empty* (*length rel-meta*))

thm *relpow-meta-trancl*

thm *relpow-meta-succ.relpow-valid-metas*

end

2.3 Transitive Closure with can_combine function

locale *relpow-meta-combine* =
relpow-data-structure un set-of memb empty valid-meta
for *un* :: $(('a::linorder) * 'b)$ *list* ⇒ *'c* ⇒ *'c* **and** *set-of memb empty valid-meta* +
fixes *can-combine* :: *'a* ⇒ *'a* ⇒ *bool*
and *combine* :: *'a* ⇒ *'a* ⇒ *'a*
and *combine-meta* :: *'a* ⇒ *'b* ⇒ *'a* ⇒ *'b* ⇒ *'b*
and *as* :: $('a \times 'b)$ *list*
assumes *valid-meta-combine*: *can-combine k1 k2* ⟹ *valid-meta k1 v1* ⟹ *valid-meta k2 v2*
⟹ *valid-meta* (*combine k1 k2*) (*combine-meta k1 v1 k2 v2*)
and *as-valid-metas*: *valid-metas* (*set as*)
begin

fun *succ* :: $(('a * 'b)$ *list* ⇒ $('a * 'b)$ *list*) **where**
succ [] = []
| *succ* ((*a*,*p*)#*xs*) =
map (λ(*b*,*q*). (*combine a b*, *combine-meta a p b q*)) (*filter* (λ(*b*,*q*). *can-combine a b*) *as*)
@ *succ xs*

definition *relpow-meta-combine-impl* :: *'c* **where**
relpow-meta-combine-impl = *relpow-meta-impl succ un memb as empty* (*length as*)

definition *rel-succ* ≡ {(*a*,*combine a b*) |*a b*. *b* ∈ *fst* ‘ *set as* ∧ *can-combine a b*}

lemma *succ-aux*: *set* (*map* (λ(*b*,*q*). (*combine a b*, *combine-meta a p b q*)) (*filter* (λ(*b*,*q*). *can-combine a b*) *as*))
= {(*combine a b*, *combine-meta a p b q*) | *b q*. (*b*,*q*) ∈ *set as* ∧ *can-combine a b*} **by** *auto*

lemma *succ-set*: *set* (*succ xs*) = {(*combine a b*, *combine-meta a p b q*) |*a p b q*. (*a*,*p*) ∈ *set xs* ∧ (*b*,*q*) ∈ *set as* ∧ *can-combine a b*}
by (*induction xs*) (*simp*,*fastforce*)

lemma *succ-rel-succ*: *fst* ‘ *set* (*succ xs*) = {*b*. ∃ *a*∈*fst* ‘ *set xs*. (*a*, *b*) ∈ *rel-succ*}
proof −
have *fst* ‘ *set* (*succ xs*) = *fst* ‘ {(*combine a b*, *combine-meta a p b q*) | *a p b q*. (*a*,*p*) ∈ *set xs* ∧(*b*,*q*) ∈ *set as* ∧ *can-combine a b*} **using** *succ-set* **by** *simp*
also have ... = {*combine a b* | *a p b q*. (*a*,*p*) ∈ *set xs* ∧(*b*,*q*) ∈ *set as* ∧ *can-combine a b*} **by** *force*
also have ... = {*b*. ∃ *a*∈*fst* ‘ *set xs*. (*a*, *b*) ∈ *rel-succ*} **unfolding** *rel-succ-def* **by** *force*
finally show *?thesis* **.**
qed

lemma *succ-valid-metas*: *valid-metas* (*set as*) ⟹ *valid-metas* (*set xs*) ⟹ *valid-metas* (*set* (*succ xs*))
unfolding *valid-metas-set-def*
using *succ-set valid-meta-combine* **by** (*auto split*: *prod.splits*)

sublocale *relpow-meta-succ un set-of memb empty valid-meta succ rel-succ*
using *succ-rel-succ succ-valid-metas as-valid-metas* **by** *unfold-locales*

end

locale *trancl-meta-combine-impl* =
relpow-data-structure un set-of memb empty valid-meta
for *un* :: ((′*a*::*linorder* ∗ ′*a*) ∗ ′*b*) *list* ⇒ ′*c* ⇒ ′*c* **and** *set-of memb empty valid-meta* +
fixes *rel-meta* :: ((′*a* × ′*a*) × ′*b*) *list*
and *combine-meta* :: ′*b* ⇒ ′*b* ⇒ ′*b*
assumes *rel-valid-metas*: *valid-metas* (*set rel-meta*)
and *valid-meta-combine-meta*: *valid-meta* (*x*, *y*) *pxy* ⟹ *valid-meta* (*y*, *z*) *pyz* ⟹ *valid-meta* (*x*, *z*) (*combine-meta pxy pyz*)
begin

definition *can-combine* :: (′*a* ∗ ′*a*) ⇒ (′*a* ∗ ′*a*) ⇒ *bool* **where**
can-combine = (λ(*x*, *y*) (*y*′, *z*). *y* = *y*′)

definition *combine* :: (′*a* ∗ ′*a*) ⇒ (′*a* ∗ ′*a*) ⇒ (′*a* ∗ ′*a*) **where**
combine = (λ(*x*, *y*) (*y*′, *z*). (*x*, *z*))

sublocale *relpow-meta-combine un set-of memb empty valid-meta can-combine combine* λ*u b1 v b2*. *combine-meta b1 b2 rel-meta*
by *unfold-locales* (*auto simp*: *valid-meta-combine-meta combine-def can-combine-def rel-valid-metas*)

lemma *succ-rel-meta*: *fst* ‘ *set* (*succ as*) = {(*x*,*z*) | *x y z*. (*x*,*y*) ∈ *fst* ‘ *set as* ∧ (*y*,*z*) ∈ *fst* ‘ *set rel-meta*} (**is** *?l* = *?r*)
proof
show *?l* ⊆ *?r*
proof
fix *c*

assume $c \in ?l$
then obtain $a\ p\ b\ q$ **where** c: $c = combine\ a\ b$ **and** a-as: $(a,\ p) \in set\ as$ **and** b-rel-$meta$: $(b,\ q) \in set\ rel\text{-}meta \wedge can\text{-}combine\ a\ b$ **using** *succ-set* **by** *auto*
then obtain $x\ y\ z$ **where** a: $a = (x,y)$ **and** b: $b = (y,z)$ **unfolding** *can-combine-def* **by** *blast*
have *fst-set-ab*: $a \in fst\ `\ set\ as \wedge b \in fst\ `\ set\ rel\text{-}meta$ **using** *a-as b-rel-meta* **by** *force*
have *c-xy*: $c = (x,z)$ **by** (*simp add*: *c combine-def a b*)
then show $c \in ?r$ **using** *fst-set-ab* **by** (*auto simp add*: *a b c combine-def*)
qed
show $?l \supseteq ?r$
proof
fix c
assume $c \in ?r$
then obtain $x\ y\ z$ **where** c: $c = (x,z)$ **and** *xy-as*: $(x,y) \in fst\ `\ set\ as$ **and** *yz-rel-meta*: $(y,z) \in fst\ `\ set\ rel\text{-}meta$ **by** *auto*
then obtain $p\ q$ **where** *xy-p-as*: $((x,y),p) \in set\ as$ **and** *yz-q-rel-meta*: $((y,z),q) \in set\ rel\text{-}meta$ **by** *auto*
have *c-combine*: $c = combine\ (x,y)\ (y,z)$ **by** (*simp add*: *c combine-def*)
have *can-combine*: $can\text{-}combine\ (x,y)\ (y,z)$ **by** (*simp add*: *can-combine-def*)
then show $c \in ?l$ **unfolding** *succ-set* **using** *xy-p-as yz-q-rel-meta c-combine* **by** *force*
qed
qed

sublocale *trancl-meta-succ*: *trancl-meta-succ un set-of memb empty valid-meta succ rel-meta*
by *unfold-locales* (*auto simp add*: *succ-rel-meta succ-valid-metas rel-valid-metas*)

theorem *trancl-meta-combine*: $keys\text{-}of\ (relpow\text{-}meta\text{-}impl\ succ\ un\ memb\ rel\text{-}meta\ empty\ (length\ rel\text{-}meta)) = trancl\text{-}meta\text{-}succ.rel^+$
using *trancl-meta-succ.relpow-meta-trancl* **.**

end

end